

## 第六章 键盘

在 Microsoft Windows 98 中，键盘和滑鼠是两个标准的使用者输入来源，在一些连贯操作中常产生互补作用。当然，滑鼠在今天的應用程式中比十年前使用得更为广泛。甚至在一些應用程式中，我们更习惯於使用滑鼠，例如在游戏、画图程式、音乐程式以及 Web 浏览器等程式中就是这样。然而，我们可以不使用滑鼠，但绝对不能从一般的 PC 中把键盘拆掉。

相对于个人电脑的其他元件，键盘有非常久远的历史，它起源於 1874 年的第一台 Remington 打字机。早期的电脑程式员用键盘在 Hollerith 卡片上打孔，后来在终端机上用键盘直接与大型主机沟通。PC 上的键盘在某些方面进行了扩充，加上了功能键、游标移动键和单独的数字键盘，但它们的输入原理基本相同。

### 键盘基础

您大概已经猜到 Windows 程式是如何获得键盘输入的：键盘输入以讯息的形式传递给程式的视窗讯息处理程式。实际上，第一次学习讯息时，键盘事件就是一个讯息如何将不同型态资讯传递给應用程式的显例。

Windows 用八种不同的讯息来传递不同的键盘事件。这好像太多了，但是（就像我们所看到的一样）程式可以忽略其中至少一半的讯息而不会有任何问题。并且，在大多数情况下，这些讯息中包含的键盘资讯会多於程式所需要的。处理键盘的部分工作就是识别出哪些讯息是重要的，哪些是不重要的。

### 忽略键盘

虽然键盘是 Windows 程式中使用者的主要来源，但是程式不必对它接收的所有讯息都作出回应。Windows 本身也能处理许多键盘功能。

例如，您可以忽略那些属于系统功能的按键，它们通常用到 Alt 键。程式不必监视这些按键，因为 Windows 会将按键的作用通知程式（当然，如果程式想这么做，它也能监视这些按键）。虽然呼叫程式功能表的按键将通过视窗的视窗讯息处理程式，但通常内定的处理方式是将按键传递给 DefWindowProc。最终，视窗讯息处理程式将获得一个讯息，表示一个功能表项被选择了。通常，这是所有视窗讯息处理程式需要知道的（在第十章将介绍功能表）。

有些 Windows 程式使用「键盘加速键」来启动通用功能表项。加速键通常是功能键或字母同 Ctrl 键的组合（例如，Ctrl-S 用於保存档案）。这些键盘加

速键与程式功能表一起在程式的资源描述档案中定义（我们可以在第十章看到）。Windows 将这些键盘加速键转换为功能表命令讯息，您不必自己去进行转换。

对话方块也有键盘介面，但是当对话方块处于活动状态时，應用程式通常不必监视键盘。键盘介面由 Windows 处理，Windows 把关于按键作用的讯息发送给程式。对话方块可以包含用于输入文字的编辑控制项。它们一般是小方框，使用者可以在框中键入字符串。Windows 处理所有编辑控制项逻辑，并在输入完毕后，将编辑控制项的最终内容传送给程式。关于对话方块的详细资讯，请参见第十一章。

编辑控制项不必局限于单独一行，而且也不限于只在对话方块中。一个在程式主视窗内的多行编辑控制项就能够作为一个简单的文字编辑器了（参见第九、十、十一和十三章的 POPPAD 程式）。Windows 甚至有一个 Rich Text 文字编辑控制项，允许您编辑和显示格式化的文字（请参见 Platform SDK/User Interface Services/Controls/Rich Edit Controls）。

您将会发现，在开发 Windows 程式时，可以使用处理键盘和鼠标输入的子视窗控制项来将较高层的资讯传递回父视窗。只要这样的控制项用得够多，您就不会因处理键盘讯息而烦恼了。

## 谁获得了焦点

与所有的个人电脑硬体一样，键盘必须由在 Windows 下执行的所有應用程式共用。有些應用程式可能有多个视窗，键盘必须由该應用程式内的所有视窗共用。

回想一下，程式用来从讯息佇列中检索讯息的 MSG 结构包括 hwnd 栏位。此栏位指出接收讯息的视窗控制项码。讯息回圈中的 DispatchMessage 函式向视窗讯息处理程式发送该讯息，此视窗讯息处理程式与需要讯息的视窗相联系。在按下键盘上的键时，只有一个视窗讯息处理程式接收键盘讯息，并且此讯息包括接收讯息的视窗控制项码。

接收特定键盘事件的视窗具有输入焦点。输入焦点的概念与活动视窗的概念很相近。有输入焦点的视窗是活动视窗或活动视窗的衍生视窗（活动视窗的子视窗，或者活动视窗子视窗的子视窗等等）。

通常很容易辨别活动视窗。它通常是顶层视窗——也就是说，它的父视窗代号是 NULL。如果活动视窗有标题列，Windows 将突出显示标题列。如果活动视窗具有对话方块架（对话方块中很常见的格式）而不是标题列，Windows 将突出显示框架。如果活动视窗目前是最小化的，Windows 将在工作列中突出显示该

项，其显示就像一个按下的按钮。

如果活动视窗有子视窗，那么有输入焦点的视窗既可以是活动视窗也可以是其子视窗。最常见的子视窗有类似以下控制项：出现在对话方块中的下压按钮、单选钮、核取方块、卷动列、编辑方块和清单方块。子视窗不能自己成为活动视窗。只有当它是活动视窗的衍生视窗时，子视窗才能有输入焦点。子视窗控制项一般通过显示一个闪烁的插入符号或虚线来表示它具有输入焦点。

有时输入焦点不在任何视窗中。这种情况发生在所有程式都是最小化的时候。这时，Windows 将继续向活动视窗发送键盘讯息，但是这些讯息与发送给非最小化的活动视窗的键盘讯息有不同的形式。

视窗讯息处理程式通过拦截 WM\_SETFOCUS 和 WM\_KILLFOCUS 讯息来判定它的视窗何时拥有输入焦点。WM\_SETFOCUS 指示视窗正在得到输入焦点，WM\_KILLFOCUS 表示视窗正在失去输入焦点。我将在本章的后面详细说明这些讯息。

## 伫列和同步

当使用者按下并释放键盘上的键时，Windows 和键盘驱动程序将硬体扫描码转换为格式讯息。然而，这些讯息并不保存在讯息伫列中。实际上，Windows 在所谓的「系统讯息伫列」中保存这些讯息。系统讯息伫列是独立的讯息伫列，它由 Windows 维护，用於初步保存使用者从键盘和滑鼠输入的资讯。只有当 Windows 應用程式处理完前一个使用者输入讯息时，Windows 才会从系统讯息伫列中取出下一个讯息，并将其放入應用程式的讯息伫列中。

此过程分为两步：首先在系统讯息伫列中保存讯息，然後将它们放入應用程式的讯息伫列，其原因是需要同步。就像我们刚才所学的，假定接收键盘输入的视窗就是有输入焦点的视窗。使用者的输入速度可能比應用程式处理按键的速度快，并且特定的按键可能会使焦点从一个视窗切换到另一个视窗，後来的按键就输入到了另一个视窗。但如果後来的按键已经记下了目标视窗的位址，并放入了應用程式讯息伫列，那么後来的按键就不能输入到另一个视窗。

## 按键和字元

應用程式从 Windows 接收的关于键盘事件的讯息可以分为按键和字元两类，这与您看待键盘的两种方式一致。

首先，您可以将键盘看作是键的集合。键盘只有唯一的 A 键，按下该键是一次按键，释放该键也是一次按键。但是键盘也是能产生可显示字元或控制字元的输入设备。根据 Ctrl、Shift 和 Caps Lock 键的状态，A 键能产生几个字

元。通常情况下，此字元为小写 a。如果按下 Shift 键或者打开了 Caps Lock，则该字元就变成大写 A。如果按下了 Ctrl，则该字元为 Ctrl-A（它在 ASCII 中有意义，但在 Windows 中可能是某事件的键盘加速键）。在一些键盘上，A 按键之前可能有「死字元键(dead-character key)」或者 Shift、Ctrl 或者 Alt 的不同组合，这些组合可以产生带有音调标记的小写或者大写，例如，à、á、â、Ä、或 Å。

对产生可显示字元的按键组合，Windows 不仅给程式发送按键讯息，而且还发送字元讯息。有些键不产生字元，这些键包括 shift 键、功能键、光标移动键和特殊字元键如 Insert 和 Delete。对于这些键，Windows 只产生按键讯息。

## 按键讯息

当您按下一个键时，Windows 把 WM\_KEYDOWN 或者 WM\_SYSKEYDOWN 讯息放入有输入焦点的视窗的讯息伫列；当您释放一个键时，Windows 把 WM\_KEYUP 或者 WM\_SYSKEYUP 讯息放入讯息伫列中。

表 6-1

	键按下	键释放
非系统键	WM_KEYDOWN	WM_KEYUP
系统键	WM_SYSKEYDOWN	WM_SYSKEYUP

通常「down（按下）」和「up（放开）」讯息是成对出现的。不过，如果您按住一个键使得自动重复功能生效，那么当该键最后被释放时，Windows 会给视窗讯息处理程式发送一系列 WM\_KEYDOWN（或者 WM\_SYSKEYDOWN）讯息和一个 WM\_KEYUP（或者 WM\_SYSKEYUP）讯息。像所有放入伫列的讯息一样，按键讯息也有时间资讯。通过呼叫 GetMessageTime，您可以获得按下或者释放键的相对时间。

## 系统按键与非系统按键

WM\_SYSKEYDOWN 和 WM\_SYSKEYUP 中的「SYS」代表「系统」，它表示该按键对 Windows 比对 Windows 應用程式更加重要。WM\_SYSKEYDOWN 和 WM\_SYSKEYUP 讯息经常由与 Alt 相组合的按键产生，这些按键启动程式功能表或者系统功能表上的选项，或者用于切换活动视窗等系统功能（Alt-Tab 或者 Alt-Esc），也可以用作系统功能表加速键（Alt 键与一个功能键相结合，例如 Alt-F4 用于关闭應用程式）。程式通常忽略 WM\_SYSKEYUP 和 WM\_SYSKEYDOWN 讯息，并将它们传送到 DefWindowProc。由于 Windows 要处理所有 Alt 键的功能，所以您无需拦截

这些讯息。您的视窗讯息处理程式将最後收到关于这些按键结果（如功能表选择）的其他讯息。如果您想在自己的视窗讯息处理程式中加上拦截系统按键的程式码（如本章後面的 KEYVIEW1 和 KEYVIEW2 程式所作的那样），那么在处理这些讯息之後再传送到 DefWindowProc，Windows 就仍然可以将它们用於通常的目的。

但是，请再考虑一下，几乎所有会影响使用者程式视窗的讯息都会先通过使用者视窗讯息处理程式。只有使用者把讯息传送到 DefWindowProc，Windows 才会对讯息进行处理。例如，如果您将下面几行叙述：

```
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
    return 0 ;
```

加入到一个视窗讯息处理程式中，那么当您的程式主视窗拥有输入焦点时，就可以有效地阻止所有 Alt 键操作（我将在本章的後面讨论 WM\_SYSCHAR），其中包括 Alt-Tab、Alt-Esc 以及功能表操作。虽然我怀疑您会这么做，但是，我相信您会感到视窗讯息处理程式的强大功能。

WM\_KEYDOWN 和 WM\_KEYUP 讯息通常是在按下或者释放不带 Alt 键的键时产生的，您的程式可以使用或者忽略这些讯息，Windows 本身并不处理这些讯息。

对所有四类按键讯息，wParam 是虚拟键代码，表示按下或释放的键，而 lParam 则包含属于按键的其他资料。

## 虚拟键码

虚拟键码保存在 WM\_KEYDOWN、WM\_KEYUP、WM\_SYSKEYDOWN 和 WM\_SYSKEYUP 讯息的 wParam 参数中。此代码标识按下或释放的键。

哈，又是「虚拟」，您喜欢这个词吗？虚拟指的是假定存在於思想中而不是现实世界中的一些事物，也只有熟练使用 DOS 组合语言编写应用程式的程式写作者才有可能指出，为什么对 Windows 键盘处理如此基本的键码是虚拟的而不是真实的。

对于早期的程式写作者来说，真实的键码由实际键盘硬体产生。在 Windows 文件中将这些键码称为「扫描码(scan codes)」。在 IBM 相容机种上，扫描码 16 是 Q 键，17 是 W 键，18 是 E、19 是 R，20 是 T，21 是 Y 等等。这时您会发现，扫描码是依据键盘的实际布局的。Windows 开发者认为这些代码过於与设备相关了，於是他们试图通过定义所谓的虚拟键码，以便经由与装置无关的方式处理键盘。其中一些虚拟键码不能在 IBM 相容机种上产生，但可能会在其他制造商生产的键盘中找到，或者在未来的键盘上找到。

您使用的大多数虚拟键码的名称在 WINUSER.H 表头档案中都定义为以 VK\_ 开头。表 6-2 列出了这些名称和数值（十进位和十六进位），以及与虚拟键相对应的 IBM 相容机种键盘上的键。下表也标出了 Windows 执行时是否需要这些键。下表还按数位顺序列出了虚拟键码。

前四个虚拟键码中有三个指的是滑鼠键：

表 6-2

十进位	十六进位	WINUSER.H 识别字	必需?	IBM 相容键盘
1	01	VK_LBUTTON		滑鼠左键
2	02	VK_RBUTTON		滑鼠右键
3	03	VK_CANCEL	√	Ctrl-Break
4	04	VK_MBUTTON		滑鼠中键

您永远都不会从键盘讯息中获得这些滑鼠键代码。在下一章可以看到，我们能够从滑鼠讯息中获得它们。VK\_CANCEL 代码是一个虚拟键码，它包括同时按下两个键 (Ctrl-Break)。Windows 應用程式通常不使用此键。

表 6-3 中的键——Backspace、Tab、Enter、Escape 和 Spacebar——通常用於 Windows 程式。不过，Windows 一般用字元讯息（而不是键盘讯息）来处理这些键。

表 6-3

十进位	十六进位	WINUSER.H 识别字	必需?	IBM 相容键盘
8	08	VK_BACK	√	Backspace
9	09	VK_TAB	√	Tab
12	0C	VK_CLEAR		Num Lock 关闭时的数字键盘 5
13	0D	VK_RETURN	√	Enter （或者另一个）
16	10	VK_SHIFT	√	Shift （或者另一个）
17	11	VK_CONTROL	√	Ctrl （或者另一个）
18	12	VK_MENU	√	Alt （或者另一个）
19	13	VK_PAUSE		Pause
20	14	VK_CAPITAL	√	Caps Lock
27	1B	VK_ESCAPE	√	Esc
32	20	VK_SPACE	√	Spacebar

另外，Windows 程式通常不需要监视 Shift、Ctrl 或 Alt 键的状态。

表 6-4 列出的前八个码可能是与 VK\_INSERT 和 VK\_DELETE 一起最常用的虚拟键码：

表 6-4



十进位	十六进位	WINUSER.H 识别字	必需?	IBM 相容键盘
33	21	VK_PRIOR	√	Page Up
34	22	VK_NEXT	√	Page Down
35	23	VK_END	√	End
36	24	VK_HOME	√	Home
37	25	VK_LEFT	√	左箭头
38	26	VK_UP	√	上箭头
39	27	VK_RIGHT	√	右箭头
40	28	VK_DOWN	√	下箭头
41	29	VK_SELECT		
42	2A	VK_PRINT		
43	2B	VK_EXECUTE		
44	2C	VK_SNAPSHOT		Print Screen
45	2D	VK_INSERT	√	Insert
46	2E	VK_DELETE	√	Delete
47	2F	VK_HELP		

注意，许多名称（例如 VK\_PRIOR 和 VK\_NEXT）都与键上的标志不同，而且也与卷动列中的识别字不统一。Print Screen 键在平时都被 Windows 應用程式所忽略。Windows 本身回应此键时会将视讯显示的点阵图影本存放到剪贴板中。假使有键盘提供了 VK\_SELECT、VK\_PRINT、VK\_EXECUTE 和 VK\_HELP，大概也没几个人看过那样的键盘。

Windows 也包括在主键盘上的字母和数位键的虚拟键码（数字键盘将单独处理）。

表 6-5

十进位	十六进位	WINUSER.H 识别字	必需?	IBM 相容键盘
48-57	30-39	无	√	主键盘上的 0 到 9
65-90	41-5A	无	√	A 到 Z

注意，数字和字母的虚拟键码是 ASCII 码。Windows 程式几乎从不使用这些虚拟键码；实际上，程式使用的是 ASCII 码字元的字元讯息。

表 6-6 所示的代码是由 Microsoft Natural Keyboard 及其相容键盘产生的：

表 6-6

十进位	十六进位	WINUSER.H 识别字	必需?	IBM 相容键盘
91	5B	VK_LWIN		左 Windows 键

92	5C	VK_RWIN		右 Windows 键
93	5D	VK_APPS		Applications 键

Windows 用 VK\_LWIN 和 VK\_RWIN 键打开「开始」功能表或者（在以前的版本中）启动「工作管理员程式」。这两个都可以用於登录或登出 Windows（只在 Microsoft Windows NT 中有效），或者登录或登出网路（在 Windows for Applications 中）。应用程式能够通过显示辅助资讯或者当成捷径键看待来处理 application 键。

表 6-7 所示的代码用於数字键盘上的键（如果有的话）：

表 6-7

十进位	十六进位	WINUSER.H 识别字	必需?	IBM 相容键盘
96-105	60-69	VK_NUMPAD0 到 VK_NUMPAD9		NumLock 打开时 数字键盘上的 0 到 9
106	6A	VK_MULTIPLY		数字键盘上的*
107	6B	VK_ADD		数字键盘上的+
108	6C	VK_SEPARATOR		
109	6D	VK_SUBTRACT		数字键盘上的-
110	6E	VK_DECIMAL		数字键盘上的.
111	6F	VK_DIVIDE		数字键盘上的/

最後，虽然多数的键盘都有 12 个功能键，但 Windows 只需要 10 个，而位元旗标却有 24 个。另外，程式通常用功能键作为键盘加速键，这样，它们通常不处理表 6-8 所示的按键：

表 6-8

十进位	十六进位	WINUSER.H 识别字	必需?	IBM 相容键盘
112-121	70-79	VK_F1 到 VK_F10	√	功能键 F1 到 F10
122-135	7A-87	VK_F11 到 VK_F24		功能键 F11 到 F24
144	90	VK_NUMLOCK		Num Lock
145	91	VK_SCROLL		Scroll Lock

另外，还定义了一些其他虚拟键码，但它们只用於非标准键盘上的键，或者通常在大型主机终端机上使用的键。查看/ Platform SDK / User Interface Services / User Input / Virtual-Key Codes，可得到完整的列表。



## lParam 资讯

在四个按键讯息 (WM\_KEYDOWN、WM\_KEYUP、WM\_SYSKEYDOWN 和 WM\_SYSKEYUP) 中, wParam 讯息参数含有上面所讨论的虚拟键码, 而 lParam 讯息参数则含有对了解按键非常有用的其他资讯。lParam 的 32 位分为 6 个栏位, 如图 6-1 所示。

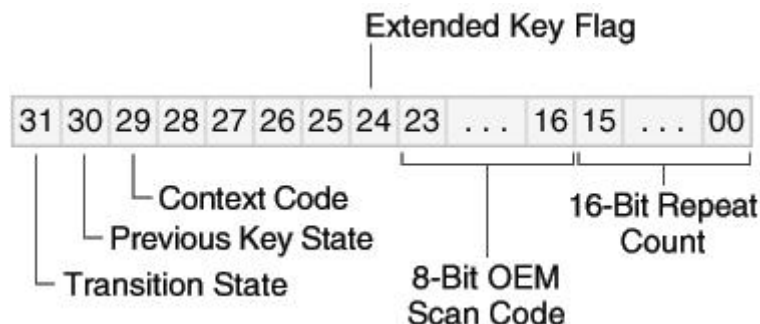


图 6-1 lParam 变数的 6 个按键讯息栏位

## 重复计数 (Repeat Count)

重复计数是该讯息所表示的按键次数, 大多数情况下, 重复计数设定为 1。不过, 如果按下一个键之後, 您的视窗讯息处理程式不够快, 以致不能处理自动重复速率 (您可以在「控制台」的「键盘」中进行设定) 下的按键讯息, Windows 就把几个 WM\_KEYDOWN 或者 WM\_SYSKEYDOWN 讯息组合到单个讯息中, 并相应地增加重复计数。WM\_KEYUP 或 WM\_SYSKEYUP 讯息的重复计数总是为 1。

因为重复计数大於 1 指示按键速率大於您程式的处理能力, 所以您也可能想在处理键盘讯息时忽略重复计数。几乎每个人都有文书处理或执行試算表时画面卷过头的经验, 因为多余的按键堆满了键盘缓冲区, 所以当程式用一些时间来处理每一次按键时, 如果忽略您程式中的重复计数, 就能够解决此问题。不过, 有时可能也会用到重复计数, 您应该尝试使用两种方法执行程式, 并从中找出一种较好的方法。

## OEM 扫描码 (OEM Scan Code)

OEM 扫描码是由硬体 (键盘) 产生的代码。这对中古时代的组合语言程式写作者来说应该很熟悉, 它是从 PC 相容机种的 ROM BIOS 服务中所获得的值 (OEM 指的是 PC 的原始设备制造商 (Original Equipment Manufacturer) 及其与「IBM 标准」同步的内容)。在此我们不需要更多的资讯。除非需要依赖实际键盘布局的样貌, 不然 Windows 程式可以忽略掉几乎所有的 OEM 扫描码资讯, 参见第二十二章的程式 KBMIDI。

## 扩充键旗标 (Extended Key Flag)

如果按键结果来自 IBM 增强键盘的附加键之一，那么扩充键旗标为 1（IBM 增强型键盘有 101 或 102 个键。功能键在键盘顶端，游标移动键从数字键盘中分离出来，但在数字键盘上还保留有游标移动键的功能）。对键盘右端的 Alt 和 Ctrl 键，以及不是数字键盘那部分的游标移动键（包括 Insert 和 Delete 键）、数字键盘上的斜线（/）和 Enter 键以及 Num Lock 键等，此旗标均被设定为 1。Windows 程式通常忽略扩充键旗标。

## 内容代码 (Context Code)

右按键时，假如同时压下 ALT 键，那么内容代码为 1。对 WM\_SYSKEYUP 与 WM\_SYSKEYDOWN 而言，此位元总视为 1；而对 WM\_KEYUP 与 WM\_KEYDOWN 讯息而言，此位元为 0。除了两个之外：

如果活动视窗最小化了，则它没有输入焦点。这时候所有的按键都会产生 WM\_SYSKEYUP 和 WM\_SYSKEYDOWN 讯息。如果 Alt 键未被按下，则内容代码栏位被设定为 0。Windows 使用 WM\_SYSKEYUP 和 WM\_SYSKEYDOWN 讯息，从而使最小化了的  
活动视窗不处理这些按键。

对于一些外国语文（非英文）键盘，有些字元是通过 Shift、Ctrl 或者 Alt 键与其他键相组合而产生的。这时内容代码为 1，但是此讯息并非系统按键讯息。

## 键的先前状态 (Previous Key State)

如果在此之前键是释放的，则键的先前状态为 0，否则为 1。对 WM\_KEYUP 或者 WM\_SYSKEYUP 讯息，它总是设定为 1；但是对 WM\_KEYDOWN 或者 WM\_SYSKEYDOWN 讯息，此位元可以为 0，也可以为 1。如果为 1，则表示该键是自动重复功能所产生的第二个或者后续讯息。

## 转换状态 (Transition State)

如果键正被按下，则转换状态为 0；如果键正被释放，则转换状态为 1。对 WM\_KEYDOWN 或者 WM\_SYSKEYDOWN 讯息，此栏位为 0；对 WM\_KEYUP 或者 WM\_SYSKEYUP 讯息，此栏位为 1。

## 位移状态

在处理按键讯息时，您可能需要知道是否按下了位移键（Shift、Ctrl 和 Alt）或开关键（Caps Lock、Num Lock 和 Scroll Lock）。通过呼叫 GetKeyState

函式，您就能获得此资讯。例如：

```
iState = GetKeyState (VK_SHIFT) ;
```

如果按下了 Shift，则 iState 值为负（即设定了最高位置位元）。如果 Caps Lock 键打开，则从

```
iState = GetKeyState (VK_CAPITAL) ;
```

传回的值低位元被设为 1。此位元与键盘上的小灯保持一致。

通常，您在使用 GetKeyState 时，会带有虚拟键码 VK\_SHIFT、VK\_CONTROL 和 VK\_MENU（在说明 Alt 键时呼叫）。使用 GetKeyState 时，您也可以使用下面的识别字来确定按下的 Shift、Ctrl 或 Alt 键是左边的还是右边的：VK\_LSHIFT、VK\_RSHIFT、VK\_LCONTROL、VK\_RCONTROL、VK\_LMENU、VK\_RMENU。这些识别字只用于 GetKeyState 和 GetAsyncKeyState（下面将详细说明）。

使用虚拟键码 VK\_LBUTTON、VK\_RBUTTON 和 VK\_MBUTTON，您也可以获得鼠标键的状态。不过，大多数需要监视鼠标键与按键相组合的 Windows 程式都使用其他方法来做到这一点——即在接收到鼠标讯息时检查按键。实际上，位移状态资讯包含在鼠标资讯中，正如您在下一章中将看到的一样。

请注意 GetKeyState 的使用，它并非即时检查键盘状态，而只是检查直到目前为止正在处理的讯息的键盘状态。多数情况下，这正符合您的要求。如果您需要确定使用者是否按下了 Shift-Tab，请在处理 Tab 键的 WM\_KEYDOWN 讯息时呼叫 GetKeyState，带有参数 VK\_SHIFT。如果 GetKeyState 传回的值负，那么您就知道在按下 Tab 键之前按下了 Shift 键。并且，如果在您开始处理 Tab 键之前，已经释放了 Shift 键也没有关系。您知道，在按下 Tab 键的时候 Shift 键是按下的。

GetKeyState 不会让您获得独立于普通键盘讯息的键盘资讯。例如，您或许想暂停视窗讯息处理程式的处理，直到您按下 F1 功能键为止：

```
while (GetKeyState (VK_F1) >= 0) ; // WRONG !!!
```

不要这么做！这将让程式当死（除非在执行此叙述之前早就从讯息伫列中接收到了 F1 的 WM\_KEYDOWN）。如果您确实需要知道目前某键的状态，那么您可以使用 GetAsyncKeyState。

## 使用按键讯息

如果程式能够获得每个按键的资讯，这当然很理想，但是大多数 Windows 程式忽略了几乎所有的按键，而只处理部分的按键讯息。WM\_SYSKEYDOWN 和 WM\_SYSKEYUP 讯息是由 Windows 系统函式使用的，您不必为此费心，就算您要处理 WM\_KEYDOWN 讯息，通常也可以忽略 WM\_KEYUP 讯息。

Windows 程式通常为不产生字元的按键使用 WM\_KEYDOWN 讯息。虽然您可能

认为借助按键讯息和位移键状态资讯能将按键讯息转换为字元讯息，但是不要这么做，因为您将遇到国际键盘间的差异所带来的问题。例如，如果您得到 wParam 等於 0x33 的 WM\_KEYDOWN 讯息，您就可以知道使用者按下了键 3，到此为止一切正常。这时，如果用 GetKeyState 发现 Shift 键被按下，您就可能会认为使用者输入了#号，这可不一定。比如英国使用者就是在输入 £。

对于光标移动键、功能键、Insert 和 Delete 键，WM\_KEYDOWN 讯息是最有用的。不过，Insert、Delete 和功能键经常作为功能表加速键。因为 Windows 能把功能表加速键翻译为功能表命令讯息，所以您就不必自己来处理按键。

在 Windows 之前的 MS-DOS 應用程式中大量使用功能键与 Shift、Ctrl 和 Alt 键的组合，同样地，您也可以在 Windows 程式中使用（实际上，Microsoft Word 将大量的功能键用作命令快捷方式），但并不推荐这样做。如果您确实希望使用功能键，那么这些键应该是重复功能表命令。Windows 的目标之一就是提供不需要记忆或者使用复杂命令流程的使用者介面。

因此，可以归纳如下：多数情况下，您将只为光标移动键（有时也为 Insert 和 Delete 键）处理 WM\_KEYDOWN 讯息。在使用这些键的时候，您可以通过 GetKeyState 来检查 Shift 键和 Ctrl 键的状态。例如，Windows 程式经常使用 Shift 与光标键的组合键来扩大文书处理里选中的范围。Ctrl 键常用于修改光标键的意义。例如，Ctrl 与右箭头键相组合可以表示光标右移一个字。

决定您的程式中使用键盘方式的最好方法之一是了解现有的 Windows 程式使用键盘的方式。如果您不喜欢那些定义，当然可以对其加以修改，但是这样做不利于其他人很快地学会使用您的程式。

## 为 SYSMETS 加上键盘处理功能

在编写第四章中三个版本的 SYSMETS 程式时，我们还不了解键盘，只能使用卷动列和滑鼠来卷动文字。现在我们知道了处理键盘讯息的方法，那么不妨在程式中加入键盘介面。显然，这是处理光标移动键的工作。我们将大多数光标键（Home、End、Page Up、Page Down、Up Arrow 和 Down Arrow）用于垂直卷动，左箭头键和右箭头键用于不太重要的水平卷动。

建立键盘介面的一种简单方法是在视窗讯息处理程式中加入与 WM\_VSCROLL 和 WM\_HSCROLL 处理方式相仿，而且本质上相同的 WM\_KEYDOWN 处理方法。不过这样子做是不聪明的，因为如果要修改卷动列的做法，就必须相对地修改 WM\_KEYDOWN。

为什么不简单地将每一种 WM\_KEYDOWN 讯息都翻译成同等效用的 WM\_VSCROLL 或者 WM\_HSCROLL 讯息呢？通过向视窗讯息处理程式发送假冒讯息，我们可能会

让 WndProc 认为它获得了滚动资讯。

在 Windows 中，这种方法是可行的。发送讯息的函式叫做 SendMessage，它所用的参数与传递到视窗讯息处理程式的参数是相同的：

```
SendMessage (hwnd, message, wParam, lParam) ;
```

在呼叫 SendMessage 时，Windows 呼叫视窗代号为 hwnd 的视窗讯息处理程式，并把这四个参数传给它。当视窗讯息处理程式完成讯息处理之後，Windows 把控制传回到 SendMessage 呼叫之後的下一道叙述。您发送讯息过去的视窗讯息处理程式，可以是同一个视窗讯息处理程式、同一程式中的其他视窗讯息处理程式或者其他应用程式，中的视窗讯息处理程式。

下面说明在 SYSMETS 程式中使用 SendMessage 处理 WM\_KEYDOWN 代码的方法：

```
case WM_KEYDOWN:
    switch (wParam)
    {
        case VK_HOME:
            SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
            break ;

        case VK_END:
            SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;
            break ;

        case VK_PRIOR:
            SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0) ;
            break ;
```

至此，您已经有了大概观念了吧。我们的目标是为滚动列添加键盘介面，并且也正在这么做。通过把滚动讯息发送到视窗讯息处理程式，我们实作了用游标移动键进行滚动列的功能。现在您知道在 SYSMETS3 中为 WM\_VSCROLL 讯息加上 SB\_TOP 和 SB\_BOTTOM 处理码的原因了吧。在那里并没有用到它，但是现在处理 Home 和 End 键时就有用了。如程式 6-1 所示的 SYSENTS4 就加上了这些变化。编译这个程式时还需要用到第四章的 SYSMETS.H 档案。

#### 程式 6-1 SYSMETS4

```
SYSMETS4.C
/*-----
   SYSMETS4.C -- System Metrics Display Program No. 4
               (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 ("SysMets4") ;
    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. 4"),
        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 might 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 might 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_KEYDOWN:
    switch (wParam)
    {
case VK_HOME:
        SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
        break ;

case VK_END:
        SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;
        break ;

case VK_PRIOR:
        SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0) ;
        break ;

case VK_NEXT:
        SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0) ;
        break ;

case VK_UP:
        SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;
        break ;

case VK_DOWN:
        SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0) ;
        break ;

case VK_LEFT:
        SendMessage (hwnd, WM_HSCROLL, SB_PAGEUP, 0) ;
        break ;

case VK_RIGHT:
        SendMessage (hwnd, WM_HSCROLL, SB_PAGEDOWN, 0) ;
        break ;
    }
}
```

```

        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,
            strlen (sysmetrics[i].szLabel)) ;

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

        SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

        TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y, szBuffer,
            sprintf (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) ;  
}
```

## 字元讯息

前面讨论了利用位移状态资讯把按键讯息翻译为字元讯息的方法，并且提到，仅利用转换状态资讯还不够，因为还需要知道与国家/地区有关的键盘配置。由於这个原因，您不应该试图把按键讯息翻译为字元代码。Windows 会为您完成这一工作，在前面我们曾看到过以下的程式码：

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

这是 WinMain 中典型的讯息回圈。GetMessage 函式用伫列中的下一个讯息填入 msg 结构的栏位。DispatchMessage 以此讯息为参数呼叫适当的视窗讯息处理程式。

在这两个函式之间是 TranslateMessage 函式，它将按键讯息转换为字元讯息。如果讯息为 WM\_KEYDOWN 或者 WM\_SYSKEYDOWN，并且按键与位移状态相组合产生一个字元，则 TranslateMessage 把字元讯息放入讯息伫列中。此字元讯息将是 GetMessage 从讯息伫列中得到的按键讯息之後的下一个讯息。

## 四类字元讯息

字元讯息可以分为四类，如表 6-9 所示。

表 6-9

	字元	死字元
非系统字元	WM_CHAR	WM_DEADCHAR
系统字元	WM_SYSCHAR	WM_SYSDEADCHAR

WM\_CHAR 和 WM\_DEADCHAR 讯息是从 WM\_KEYDOWN 得到的；而 WM\_SYSCHAR 和 WM\_SYSDEADCHAR 讯息是从 WM\_SYSKEYDOWN 讯息得到的（我将简要地讨论一下什么是死字元）。

有一个好消息：在大多数情况下，Windows 程式会忽略除 WM\_CHAR 之外的任何讯息。伴随四个字元讯息的 lParam 参数与产生字元代码讯息的按键讯息之 lParam 参数相同。不过，参数 wParam 不是虚拟键码。实际上，它是 ANSI 或 Unicode 字元代码。

这些字元讯息是我们将文字传递给视窗讯息处理程式时遇到的第一个讯

息。它们不是唯一的讯息，其他讯息伴随以 0 结尾的整个字串。视窗讯息处理程式是如何知道该字元是 8 位元的 ANSI 字元还是 16 位元的 Unicode 宽字元呢？很简单：任何与您用 RegisterClassA (RegisterClass 的 ANSI 版) 注册的视窗类别相联系的视窗讯息处理程式，都会获得含有 ANSI 字元代码的讯息。如果视窗讯息处理程式用 RegisterClassW (RegisterClass 的宽字元版) 注册，那么传递给视窗讯息处理程式的讯息就带有 Unicode 字元代码。如果程式用 RegisterClass 注册视窗类别，那么在 UNICODE 识别字被定义时就呼叫 RegisterClassW，否则呼叫 RegisterClassA。

除非在程式写作的时候混合了 ANSI 和 Unicode 的函式与视窗讯息处理程式，用 WM\_CHAR 讯息（及其他三种字元讯息）说明的字元代码将是：

```
(TCHAR) wParam
```

同一个视窗讯息处理程式可能会用到两个视窗类别，一个用 RegisterClassA 注册，而另一个用 RegisterClassW 注册。也就是说，视窗讯息处理程式可能会获得一些 ANSI 字元代码讯息和一些 Unicode 字元代码讯息。如果您的视窗讯息处理程式需要晓得目前视窗是否处理 Unicode 讯息，则它可以呼叫：

```
fUnicode = IsWindowUnicode (hwnd) ;
```

如果 hwnd 的视窗讯息处理程式获得 Unicode 讯息，那么变数 fUnicode 将为 TRUE，这表示视窗是用 RegisterClassW 注册的视窗类别。

### 讯息顺序

因为 TranslateMessage 函式从 WM\_KEYDOWN 和 WM\_SYSKEYDOWN 讯息产生了字元讯息，所以字元讯息是夹在按键讯息之间传递给视窗讯息处理程式的。例如，如果 Caps Lock 未打开，而使用者按下再释放 A 键，则视窗讯息处理程式将接收到如表 6-10 所示的三个讯息：

表 6-10

讯息	按键或者代码
WM_KEYDOWN	「A」的虚拟键码 (0x41)
WM_CHAR	「a」的字元代码 (0x61)
WM_KEYUP	「A」的虚拟键码 (0x41)

如果您按下 Shift 键，再按下 A 键，然後释放 A 键，再释放 Shift 键，就会输入大写的 A，而视窗讯息处理程式会接收到五个讯息，如表 6-11 所示：

表 6-11

讯息	按键或者代码
----	--------

WM_KEYDOWN	虚拟键码 VK_SHIFT (0x10)
WM_KEYDOWN	「A」的虚拟键码 (0x41)
WM_CHAR	「A」的字元代码 (0x41)
WM_KEYUP	「A」的虚拟键码 (0x41)
WM_KEYUP	虚拟键码 VK_SHIFT (0x10)

Shift 键本身不产生字元讯息。

如果使用者按住 A 键，以使自动重复产生一系列的按键，那么对每条 WM\_KEYDOWN 讯息，都会得到一条字元讯息，如表 6-12 所示：

表 6-12

讯息	按键或者代码
WM_KEYDOWN	「A」的虚拟键码 (0x41)
WM_CHAR	「a」的字元代码 (0x61)
WM_KEYDOWN	「A」的虚拟键码 (0x41)
WM_CHAR	「a」的字元代码 (0x61)
WM_KEYDOWN	「A」的虚拟键码 (0x41)
WM_CHAR	「a」的字元代码 (0x61)
WM_KEYDOWN	「A」的虚拟键码 (0x41)
WM_CHAR	「a」的字元代码 (0x61)
WM_KEYUP	「A」的虚拟键码 (0x41)

如果某些 WM\_KEYDOWN 讯息的重复计数大於 1，那么相应的 WM\_CHAR 讯息将具有同样的重复计数。

组合使用 Ctrl 键与字母键会产生从 0x01 (Ctrl-A) 到 0x1A (Ctrl-Z) 的 ASCII 控制代码，其中的某些控制代码也可以由表 6-13 列出的键产生：

表 6-13

按键	字元代码	产生方法	ANSI C 控制字元
Backspace	0x08	Ctrl-H	\b
Tab	0x09	Ctrl-I	\t
Ctrl-Enter	0x0A	Ctrl-J	\n
Enter	0x0D	Ctrl-M	\r
Esc	0x1B	Ctrl-[	

最右列给出了在 ANSI C 中定义的控制字元，它们用於描述这些键的字元代码。

有时 Windows 程式将 Ctrl 与字母键的组合用作功能表加速键（我将在第十章讨论），此时，不会将字母键转换成字元讯息。



## 处理控制字元

处理按键和字元讯息的基本规则是：如果需要读取输入到视窗的键盘字元，那么您可以处理 WM\_CHAR 讯息。如果需要读取游标键、功能键、Delete、Insert、Shift、Ctrl 以及 Alt 键，那么您可以处理 WM\_KEYDOWN 讯息。

但是 Tab 键怎么办？Enter、Backspace 和 Escape 键又怎么办？传统上，这些键都产生表 6-13 列出的 ASCII 控制字元。但是在 Windows 中，它们也产生虚拟键码。这些键应该在处理 WM\_CHAR 或者在处理 WM\_KEYDOWN 期间处理吗？

经过 10 年的考虑（回顾这些年来我写过的 Windows 程式码），我更喜欢将 Tab、Enter、Backspace 和 Escape 键处理成控制字元，而不是虚拟键。我通常这样处理 WM\_CHAR：

```
case WM_CHAR:
    //其他行程式
    switch (wParam)
    {
    case '\b':          // backspace
        //其他行程式
        break ;

    case '\t':          // tab
        //其他行程式
        break ;

    case '\n':          // linefeed
        //其他行程式
        break ;

    case '\r':          // carriage return
        //其他行程式
        break ;

    default:            // character codes
        //其他行程式
        break ;
    }
    return 0 ;
```

## 死字元讯息

Windows 程式经常忽略 WM\_DEADCHAR 和 WM\_SYSDEADCHAR 讯息，但您应该明确地知道死字元是什么，以及它们工作的方式。

在某些非 U.S. 英语键盘上，有些键用於给字母加上音调。因为它们本身不产生字元，所以称之为「死键」。例如，使用德语键盘时，对于 U.S. 键盘上的

+/=键，德语键盘的对应位置就是一个死键，未按下 Shift 键时它用於标识锐音，按下 Shift 键时则用於标识抑音。

当使用者按下这个死键时，视窗讯息处理程式接收到一个 wParam 等於音调本身的 ASCII 或者 Unicode 代码的 WM\_DEADCHAR 讯息。当使用者再按下可以带有此音调的字母键（例如 A 键）时，视窗讯息处理程式会接收到 WM\_CHAR 讯息，其中 wParam 等於带有音调的字母「a」的 ANSI 代码。

因此，使用者程式不需要处理 WM\_DEADCHAR 讯息，原因是 WM\_CHAR 讯息已含有程式所需要的所有资讯。Windows 的做法甚至还设计了内部错误处理。如果在死键之後跟有不能带此音调符号的字母（例如「s」），那么视窗讯息处理程式将在一行接收到两条 WM\_CHAR 讯息——前一个讯息的 wParam 等於音调符号本身的 ASCII 代码（与传递到 WM\_DEADCHAR 讯息的 wParam 值相同），第二个讯息的 wParam 等於字母 s 的 ASCII 代码。

当然，要感受这种做法的运作方式，最好的方法就是实际操作。您必须载入使用死键的外语键盘，例如前面讲过的德语键盘。您可以这样设定：在「控制台」中选择「键盘」，然後选择「语系」页面标签。然後您需要一个應用程式，该程式可以显示它接收的每一个键盘讯息的详细资讯。下面的 KEYVIEW1 就是这样的程式。

## 键盘讯息和字元集

本章剩下的范例程式有缺陷。它们不能在所有版本的 Windows 下都正常执行。这些缺陷不是特意引过程式码中的；事实上，您也许永远不会遇到这些缺陷。只有在不同的键盘语言和键盘布局间切换，以及在中位元组字元集的远东版 Windows 下执行程式时，这些问题才会出现——所以我不愿将它们称为「错误」。

不过，如果程式使用 Unicode 编译并在 Windows NT 下执行，那么程式会执行得更好。我在第二章提到过这个问题，并且展示了 Unicode 对简化棘手的国际化问题的重要性。

## KEYVIEW1 程式

了解键盘国际化问题的第一步，就是检查 Windows 传递给视窗讯息处理程式的键盘内容和字元讯息。程式 6-2 所示的 KEYVIEW1 会对此有所帮助。该程式在显示区域显示 Windows 向视窗讯息处理程式发送的 8 种不同键盘讯息的全部资讯。

### 程式 6-2 KEYVIEW1

```

KEYVIEW1.C
/*-----
    KEYVIEW1.C --      Displays Keyboard and Character Messages
                      (c) Charles Petzold, 1998
-----*/

#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("KeyView1") ;
    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 ("Keyboard Message Viewer #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    cxClientMax, cyClientMax, cxClient, cyClient, cxChar, cyChar ;
    static int    cLinesMax, cLines ;
    static PMSG   pmsg ;
    static RECT   rectScroll ;
    static TCHAR  szTop[] = TEXT ("Message Key   Char ")
                                TEXT ("Repeat Scan Ext ALT Prev
Tran") ;
    static TCHAR  szUnd[] = TEXT ("_____")
                                TEXT ("_____") ;

    static TCHAR * szFormat[2] = {
        TEXT ("% -13s %3d %-15s%c%6u %4d %3s %3s %4s %4s"),
        TEXT ("% -13s 0x%04X%1s%c %6u %4d %3s %3s %4s %4s") } ;

    static TCHAR * szYes  = TEXT ("Yes") ;
    static TCHAR * szNo   = TEXT ("No") ;
    static TCHAR * szDown = TEXT ("Down") ;
    static TCHAR * szUp   = TEXT ("Up") ;

    static TCHAR * szMessage [] = {
        TEXT ("WM_KEYDOWN"), TEXT ("WM_KEYUP"),
        TEXT ("WM_CHAR"), TEXT ("WM_DEADCHAR"),
        TEXT ("WM_SYSKEYDOWN"), TEXT ("WM_SYSKEYUP"),
        TEXT ("WM_SYSCHAR"), TEXT ("WM_SYSDEADCHAR") } ;

    HDC          hdc ;
    int           i, iType ;
    PAINTSTRUCT   ps ;
    TCHAR         szBuffer[128], szKeyName [32] ;
    TEXTMETRIC    tm ;

    switch (message)
    {
    case WM_CREATE:
    case WM_DISPLAYCHANGE:
        // Get maximum size of client area
        cxClientMax = GetSystemMetrics (SM_CXMAXIMIZED) ;
        cyClientMax = GetSystemMetrics (SM_CYMAXIMIZED) ;

        // Get character size for fixed-pitch font
        hdc = GetDC (hwnd) ;
        SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
        GetTextMetrics (hdc, &tm) ;
        cxChar = tm.tmAveCharWidth ;

```

```

    cyChar = tm.tmHeight ;

    ReleaseDC (hwnd, hdc) ;
        // Allocate memory for display lines
    if (pmsg)
        free (pmsg) ;
        cLinesMax = cyClientMax / cyChar ;
        pmsg = malloc (cLinesMax * sizeof (MSG)) ;
        cLines = 0 ;
        // fall through
case WM_SIZE:
    if (message == WM_SIZE)
    {
        cxClient = LOWORD (lParam) ;
        cyClient = HIWORD (lParam) ;
    }
        // Calculate scrolling rectangle
    rectScroll.left      = 0 ;
    rectScroll.right = cxClient ;
    rectScroll.top       = cyChar ;
    rectScroll.bottom = cyChar * (cyClient / cyChar) ;

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

case WM_KEYDOWN:
case WM_KEYUP:
case WM_CHAR:
case WM_DEADCHAR:
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
case WM_SYSDEADCHAR:
        // Rearrange storage array
    for (i = cLinesMax - 1 ; i > 0 ; i--)
    {
        pmsg[i] = pmsg[i - 1] ;
    }
        // Store new message
    pmsg[0].hwnd = hwnd ;
    pmsg[0].message = message ;
    pmsg[0].wParam = wParam ;
    pmsg[0].lParam = lParam ;

    cLines = min (cLines + 1, cLinesMax) ;
        // Scroll up the display
    ScrollWindow (hwnd, 0, -cyChar, &rectScroll, &rectScroll) ;
    break ;        // i.e., call DefWindowProc so Sys messages work

```

```

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;
    SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
    SetBkMode (hdc, TRANSPARENT) ;
    TextOut (hdc, 0, 0, szTop, lstrlen (szTop)) ;
    TextOut (hdc, 0, 0, szUnd, lstrlen (szUnd)) ;

    for (i = 0 ; i < min (cLines, cyClient / cyChar - 1) ; i++)
    {
        iType =      pmsg[i].message == WM_CHAR ||
                    pmsg[i].message == WM_SYSCHAR ||
                    pmsg[i].message == WM_DEADCHAR ||
                    pmsg[i].message == WM_SYSDEADCHAR ;

        GetKeyNameText (pmsg[i].lParam, szKeyName,
                        sizeof (szKeyName) / sizeof (TCHAR)) ;

        TextOut (hdc, 0, (cyClient / cyChar - 1 - i) * cyChar, szBuffer,
        wsprintf (szBuffer, szFormat [iType],
        szMessage [pmsg[i].message - WM_KEYFIRST],
        pmsg[i].wParam,
        (PTSTR) (iType ? TEXT (" ") : szKeyName),
        (TCHAR) (iType ? pmsg[i].wParam : ' '),
        LOWORD (pmsg[i].lParam),
        HIWORD (pmsg[i].lParam) & 0xFF,
                0x01000000 & pmsg[i].lParam ? szYes : szNo,
                0x20000000 & pmsg[i].lParam ? szYes : szNo,
                0x40000000 & pmsg[i].lParam ? szDown : szUp,
                0x80000000 & pmsg[i].lParam ? szUp : szDown)) ;
    }
    EndPaint (hwnd, &ps) ;
    return 0 ;

case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}

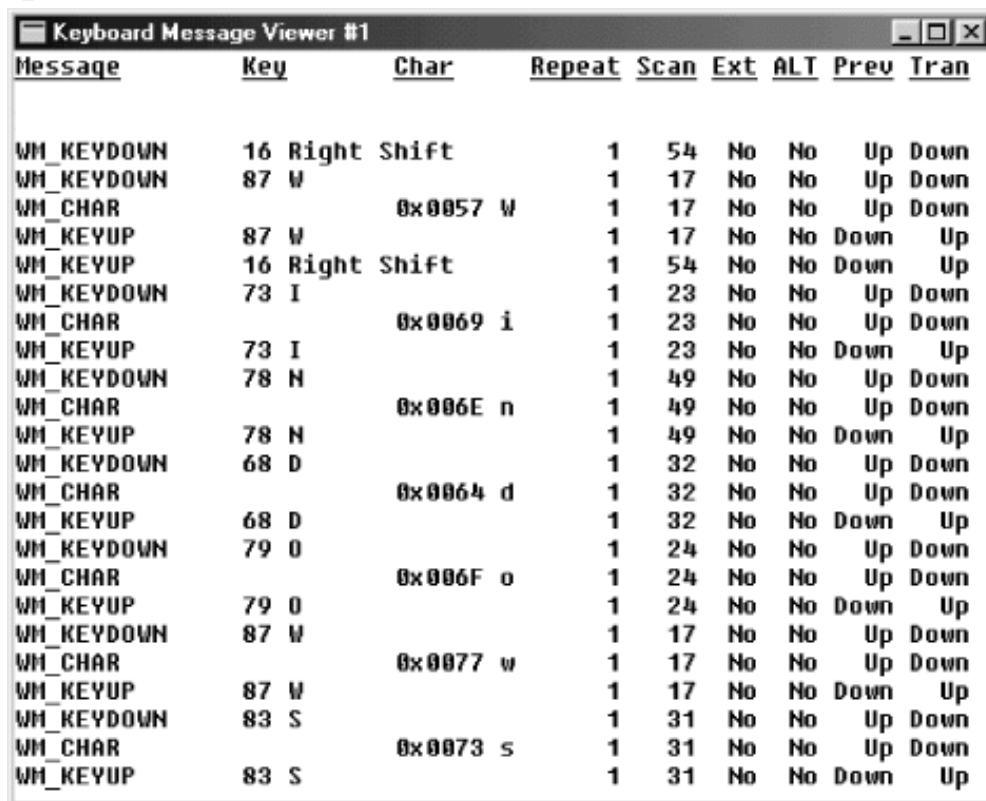
return DefWindowProc (hwnd, message, wParam, lParam) ;
}

```

KEYVIEW1 显示视窗讯息处理程式接收到的每次按键和字元讯息的内容，并将这些讯息储存在一个 MSG 结构的阵列中。该阵列的大小依据最大化视窗的大小和等宽的系统字体。如果使用者在程式执行时调整了视讯显示的大小（在这种情况下 KEYVIEW1 接收 WM\_DISPLAYCHANGE 讯息），将重新分配此阵列。KEYVIEW1 使用标准 C 的 malloc 函式为阵列配置记忆体。

图 6-2 给出了在键入「Windows」之後 KEYVIEW1 的萤幕显示。第一列显示

了键盘讯息；第二列在键名称的前面显示了按键讯息的虚拟键代码，此代码是经由 `GetKeyNameText` 函式取得的；第三列（标注为「Char」）在字元本身的後面显示字元讯息的十六进位字元代码。其余六列显示了 `lParam` 讯息参数中六个栏位的状态。



Message	Key	Char	Repeat	Scan	Ext	ALT	Prev	Tran
WM_KEYDOWN	16 Right Shift		1	54	No	No	Up	Down
WM_KEYDOWN	87 W		1	17	No	No	Up	Down
WM_CHAR		0x0057 W	1	17	No	No	Up	Down
WM_KEYUP	87 W		1	17	No	No	Down	Up
WM_KEYUP	16 Right Shift		1	54	No	No	Down	Up
WM_KEYDOWN	73 I		1	23	No	No	Up	Down
WM_CHAR		0x0069 i	1	23	No	No	Up	Down
WM_KEYUP	73 I		1	23	No	No	Down	Up
WM_KEYDOWN	78 N		1	49	No	No	Up	Down
WM_CHAR		0x006E n	1	49	No	No	Up	Down
WM_KEYUP	78 N		1	49	No	No	Down	Up
WM_KEYDOWN	68 D		1	32	No	No	Up	Down
WM_CHAR		0x0064 d	1	32	No	No	Up	Down
WM_KEYUP	68 D		1	32	No	No	Down	Up
WM_KEYDOWN	79 O		1	24	No	No	Up	Down
WM_CHAR		0x006F o	1	24	No	No	Up	Down
WM_KEYUP	79 O		1	24	No	No	Down	Up
WM_KEYDOWN	87 W		1	17	No	No	Up	Down
WM_CHAR		0x0077 w	1	17	No	No	Up	Down
WM_KEYUP	87 W		1	17	No	No	Down	Up
WM_KEYDOWN	83 S		1	31	No	No	Up	Down
WM_CHAR		0x0073 s	1	31	No	No	Up	Down
WM_KEYUP	83 S		1	31	No	No	Down	Up

图 6-2 KEYVIEW1 的萤幕显示

为便於以分行的方式显示此资讯，KEYVIEW1 使用了等宽字体。与前一章所讨论的一样，这需要呼叫 `GetStockObject` 和 `SelectObject`：

```
SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
```

KEYVIEW1 在显示区域上部画了一个标题以确定分成九行。此列文字带有底线。虽然可以建立一种带底线的字体，但这里使用了另一种方法。我定义了两个字串变数 `szTop`（有文字）和 `szUnd`（有底线），并在 `WM_PAINT` 讯息处理期间将它们同时显示在视窗顶部的同一位置。通常，Windows 以一种「不透明」的方式显示文字，也就是说显示字元时 Windows 将擦除字元背景区。这将导致第二个字串（`szUnd`）擦除掉前一个（`szTop`）。要防止这一现象的发生，可将装置内容切换到「透明」模式：

```
SetBkMode (hdc, TRANSPARENT) ;
```

这种加底线的方法只有在使用等宽字体时才可行。否则，底线字元将无法与显现在底线上面的字元等宽。



## 外语键盘问题

如果您执行美国英语版本的 Windows, 那么您可安装不同的键盘布局, 并输入外语。可以在 **控制台** 的 **键盘** 中安装外语键盘布局。选择 **语系** 页面标签, 按下 **新增** 键。要查看死键的工作方式, 您可能想安装「德语」键盘。此外, 我还要讨论「俄语」和「希腊语」的键盘布局, 因此您也可安装这些键盘布局。如果在「键盘」显示的列表中找到「俄语」和「希腊语」的键盘布局, 则需要安装多语系支援: 从「控制台」中选择 **新增/删除** 程式, 然後选择 **Windows 安装程式** 页面标签, 确认选中 **多语系支援** 核取方块。在任何情况下, 这些变更都需要原始的 Windows 光碟。

安装完其他键盘布局後, 您将在工作列右侧的通知区看到一个带有两个字母代码的蓝色框。如果内定的是英语, 那么这两个字母是「EN」。单击此图示, 将得到所有已安装键盘布局的列表。从中单击需要的键盘布局即可更改目前活动程式的键盘。此改变只影响目前活动的程式。

现在开始进行实验。不使用 UNICODE 识别字定义来编译 KEYVIEW1 程式 (在本书附带的光碟中, 非 Unicode 版本的 KEYVIEW1 程式位於 RELEASE 子目录)。在美国英语版本的 Windows 下执行该程式, 并输入字元『abcde』。WM\_CHAR 讯息与您所期望的一样: ASCII 字元代码 0x61、0x62、0x63、0x64 和 0x65 以及字母 a、b、c、d 和 e。

现在, KEYVIEW1 还在执行, 选择德语键盘布局。按下=键然後输入一个母音 (a、e、i、o 或者 u)。=键将产生一个 WM\_DEADCHAR 讯息, 母音产生一个 WM\_CHAR 讯息和 (单独的) 字元代码 0xE1、0xE9、0xED、0xF3、0xFA 和字元 á、é、í、ó 或 ú。这就是死键的工作方式。

现在选择希腊键盘布局。输入『abcde』, 您会得到什么? 您将得到 WM\_CHAR 讯息和字元代码 0xE1、0xE2、0xF8、0xE4、0xE5 和字元 á、â、ç、ä 和 å。在这里有些字元不能正确显示。难道您不应该得到希腊字母表中的字母吗?

现在切换到俄语键盘并重新输入『abcde』。现在您得到 WM\_CHAR 讯息和字元代码 0xF4、0xE8、0xF1、0xE2 和 0xF3, 以及字元 ô、è、ñ、â 和 ó。而且, 还是有些字母不能正常显示。您应从斯拉夫字母表中得到这些字母。

问题在於: 您已经切换键盘以产生不同的字元代码, 但您还没有将此切换通知 GDI, 好让 GDI 能选择适当的符号来显示解释这些字元代码。

如果您非常勇敢, 还有可用的备用 PC, 并且是专业或全球版 Microsoft Developer Network (MSDN) 的订阅户, 那么您也许想安装 (例如) 希腊版的 Windows, 您还可以把那四种键盘布局 (英语、希腊语、德语和俄语) 安装上去。现在执行 KEYLOOK1, 切换到英语键盘布局, 然後输入『abcde』。您应得到 ASCII

字元代码 0x61、0x62、0x63、0x64 和 0x65 以及字元 a、b、c、d 和 e（并且您可以放心：即使在希腊版，ASCII 还是正常通行的）。

在希腊版的 Windows 中，切换到希腊键盘布局并输入『abcde』。您将得到 WM\_CHAR 讯息和字元代码 0xE1、0xE2、0xF8、0xE4 和 0xE5。这与您在安装希腊键盘布局的英语版 Windows 中得到的字元代码相同。但现在显示的字元是  $\tau$ 、 $\beta$ 、 $\psi$ 、 $\delta$  和  $\epsilon$ 。这些确实是小写的希腊字母 alpha、beta、psi、delta 和 epsilon（gamma 怎么了？是这样，如果使用希腊版的 Windows，那么您将使用键帽上带有希腊字母的键盘。与英语 c 相对应的键正好是 psi。gamma 由与英语 g 相对应的键产生。您可在 Nadine Kano 编写的《Developing International Software for Windows 95 and Windows NT》的第 587 页看到完整的希腊字母表）。

继续在希腊版的 Windows 下运行 KEYVIEW1，切换到德语键盘布局。输入『=』键，然後依次输入 a、e、i、o 和 u。您将得到 WM\_CHAR 讯息和字元代码 0xE1、0xE9、0xED、0xF3 和 0xFA。这些字元代码与安装德语键盘布局的英语版 Windows 中的一样。不过，显示的字元却是  $\alpha$ 、 $\iota$ 、 $\nu$ 、 $\sigma$  和  $\omicron$ ，而不是正确的  $\acute{a}$ 、 $\acute{e}$ 、 $\acute{i}$ 、 $\acute{o}$  和  $\acute{u}$ 。

现在切换到俄语键盘并输入『abcde』。您会得到字元代码 0xF4、0xE8、0xF1、0xE2 和 0xF3，这与安装俄语键盘的英语版 Windows 中得到的一样。不过，显示的字元是  $\tau$ 、 $\theta$ 、 $\rho$ 、 $\beta$  和  $\sigma$ ，而不是斯拉夫字母表中的字母。

您还可安装俄语版的 Windows。现在您可以猜到，英语和俄语键盘都可以工作，而德语和希腊语则不行。

现在，如果您真的很勇敢，您还可安装日语版的 Windows 并执行 KEYVIEW1。如果再依美国键盘输入，那么您将输入英语文字，一切似乎都正常。不过，如果切换到德语、希腊语或者俄语键盘布局，并且试著作上述介绍的任何练习，您将看到以点显示的字元。如果输入大写的字母——无论是带重音符号的德语字母、希腊语字母还是俄语字母——您将看到这些字母显示为日语中用於拼写外来语的片假名。您也许对输入片假名感兴趣，但那不是德语、希腊语或者俄语。

远东版本的 Windows 包括一个称作「输入法编辑器」（IME）的实用程式，该程式显示为浮动的工具列，它允许您用标准键盘输入象形文字，即汉语、日语和朝鲜语中使用的复杂字元。一般来说，输入一组字母後，组成的字元将显示在另一个浮动视窗内。然後按 **Enter** 键，合成的字元代码就发送到了活动视窗（即 KEYVIEW1）。KEYVIEW1 几乎没什么回应——WM\_CHAR 讯息带来的字元代码大於 128，但这些代码没有意义（Nadine Kano 的书中有许多关於使用 IME 的内容）。

这时，我们已经看到了许多 KEYLOOK1 显示错误字元的例子——当执行安装了俄语或希腊语键盘布局的英语版 Windows 时，当执行安装了俄语或德语键盘布局的希腊版 Windows 时，以及执行安装了德语、俄语或者希腊语键盘布局的俄语版 Windows 时，都是这样。我们也看到了从日语版 Windows 的输入法编辑器输入字元时的错误显示。

## 字元集和字体

KEYLOOK1 的问题是字体问题。用於在萤幕上显示字元的字体和键盘接收的字元代码不一致。因此，让我们看一下字体。

我将在第十七章进行详细讨论，Windows 支援三类字体——点阵字体、向量字体和（从 Windows 3.1 开始的）TrueType 字体。

事实上向量字体已经过时了。这些字体中的字元由简单的线段组成，但这些线段没有定义填入区域。向量字体可以较好地缩放到任意大小，但字元通常看上去有些单薄。

TrueType 字体是定义了填入区域的文字轮廓字体。TrueType 字体可缩放；而且该字元的定义包括「提示」，以消除可能带来的文字不可见或者不可读的圆整问题。使用 TrueType 字体，Windows 就真正实现了 WYSIWYG（「所见即所得」），即文字在视讯显示器显示与印表机输出完全一致。

在点阵字体中，每个字元都定义为与视讯显示器上的图素对应的位元点阵。点阵字体可拉伸到较大的尺寸，但看上去带有锯齿。点阵字体通常被设计成方便在视讯显示器上阅读的字體。因此，Windows 中的标题列、功能表、按钮和对话方块的显示文字都使用点阵字体。

在内定的装置内容下获得的点阵字体称为系统字体。您可通过呼叫带有 SYSTEM\_FONT 识别字的 GetStockObject 函式来获得字体代号。KEYVIEW1 程式选择使用 SYSTEM\_FIXED\_FONT 表示的等宽系统字体。GetStockObject 函式的另一个选项是 OEM\_FIXED\_FONT。

这三种字体有（各自的）字体名称——System、FixedSys 和 Terminal。程式可以在 CreateFont 或者 CreateFontIndirect 函式呼叫中使用字体名称来指定字体。这三种字体储存在两组放在 Windows 目录内的 FONTS 子目录下的三个档案中。Windows 使用哪一组档案取决於「控制台」里的「显示器」是选择显示「小字体」还是「大字体」（亦即，您希望 Windows 假定视讯显示器是 96 dpi 的解析度还是 120 dpi 的解析度）。表 6-14 总结了所有的情况：

表 6-14

GetStockObject 识别字	字体名称	小字体档案	大字体档案
SYSTEM_FONT	System	VGASYS.FON	8514SYS.FON
SYSTEM_FIXED_FONT	FixedSys	VGAFIX.FON	8514FIX.FON
OEM_FIXED_FONT	Terminal	VGAOEM.FON	8514OEM.FON

在档案名称中,「VGA」指的是视频图形阵列 (Video Graphics Array), IBM 在 1987 年推出的显示卡。这是 IBM 第一块可显示 640 480 图素大小的 PC 显示卡。如果在「控制台」的「显示器」中选择了「小字体」(表示您希望 Windows 假定视讯显示的解析度为 96 dpi), 则 Windows 使用的这三种字体档案名将以「VGA」开头。如果选择了「大字体」(表示您希望解析度为 120 dpi), Windows 使用的档案名将以「8514」开头。8514 是 IBM 在 1987 年推出的另一种显示卡, 它的最大显示尺寸为 1024 768。

Windows 不希望您看到这些档案。这些档案的属性设定为系统和隐藏, 如果用 Windows Explorer 来查看 FONTS 子目录的内容, 您是不会看到它们的, 即使选择了查看系统和隐藏档案也不行。从开始功能表选择「寻找」选项来寻找档名满足 \*.FON 限定条件的档案。这时, 您可以双击档案名来查看字体字元是些什么。

对于许多标准控制项和使用者介面元件, Windows 不使用系统字体。相反地, 使用名称为 MS Sans Serif 的字体 (「MS」代表 Microsoft)。这也是一种点阵字体。档案 (名为 SSERIFE.FON) 包含依据 96 dpi 视讯显示器的字体, 点值为 8、10、12、14、18 和 24。您可在 GetStockObject 函式中使用 DEFAULT\_GUI\_FONT 识别字来得到该字体。Windows 使用的点值取决于「控制台」的「显示」中选择的显示解析度。

到目前为止, 我已提到四种识别字, 利用这四种识别字, 您可以用 GetStockObject 来获得用于装置内容的字体。还有三种其他字体识别字: ANSI\_FIXED\_FONT、ANSI\_VAR\_FONT 和 DEVICE\_DEFAULT\_FONT。为了开始处理键盘和字元显示问题, 让我们先看一下 Windows 中的所有备用字体。显示这些字体的程式是 STOKFONT, 如程式 6-3 所示。

### 程式 6-3 STOKFONT

```
STOKFONT.C
/*-----
    STOKFONT.C --      Stock Font Objects
                      (c) Charles Petzold, 1998
    -----*/

#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
```

```

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("StokFont") ;
    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 ("Stock Fonts"),
                          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 struct
    {

```

```

        int      idStockFont ;
        TCHAR *  szStockFont ;
    }
    stockfont [] = { OEM_FIXED_FONT,          "OEM_FIXED_FONT",
                    ANSI_FIXED_FONT, "ANSI_FIXED_FONT",
                    ANSI_VAR_FONT,          "ANSI_VAR_FONT",
                    SYSTEM_FONT,            "SYSTEM_FONT",
                    DEVICE_DEFAULT_FONT, "DEVICE_DEFAULT_FONT",
                    SYSTEM_FIXED_FONT,
                    "SYSTEM_FIXED_FONT",
                    DEFAULT_GUI_FONT,
                    "DEFAULT_GUI_FONT" } ;

    static int  iFont, cFonts = sizeof stockfont / sizeof stockfont[0] ;
    HDC          hdc ;
    int          i, x, y, cxGrid, cyGrid ;
    PAINTSTRUCT ps ;
    TCHAR          szFaceName [LF_FACESIZE], szBuffer [LF_FACESIZE +
64] ;
    TEXTMETRIC  tm ;
    switch (message)
    {
    case WM_CREATE:
        SetScrollRange (hwnd, SB_VERT, 0, cFonts - 1, TRUE) ;
        return 0 ;

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

    case WM_VSCROLL:
        switch (LOWORD (wParam))
        {
            case SB_TOP:          iFont = 0 ;          break ;
        case SB_BOTTOM:          iFont = cFonts - 1 ;    break ;
            case SB_LINEUP:
        case SB_PAGEUP:          iFont -= 1 ;          break ;
        case SB_LINEDOWN:
        case SB_PAGEDOWN:        iFont += 1 ;          break ;
        case SB_THUMBPOSITION: iFont = HIWORD (wParam) ; break ;
        }
        iFont = max (0, min (cFonts - 1, iFont)) ;
        SetScrollPos (hwnd, SB_VERT, iFont, TRUE) ;
        InvalidateRect (hwnd, NULL, TRUE) ;
        return 0 ;

    case WM_KEYDOWN:
        switch (wParam)

```

```

    {
    case VK_HOME: SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
break ;

    case VK_END:  SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;   break ;
    case VK_PRIOR:
    case VK_LEFT:
    case VK_UP:   SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;   break ;
    case VK_NEXT:
    case VK_RIGHT:
    case VK_DOWN: SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0) ; break ;
    }
    return 0 ;

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

    SelectObject (hdc, GetStockObject (stockfont[iFont].idStockFont)) ;
    GetTextFace (hdc, LF_FACESIZE, szFaceName) ;
    GetTextMetrics (hdc, &tm) ;
    cxGrid = max (3 * tm.tmAveCharWidth, 2 * tm.tmMaxCharWidth) ;
    cyGrid = tm.tmHeight + 3 ;

    TextOut (hdc, 0, 0, szBuffer,
    wsprintf ( szBuffer, TEXT (" %s: Face Name = %s, CharSet = %i"),
                stockfont[iFont].szStockFont,
                szFaceName, tm.tmCharSet)) ;

    SetTextAlign (hdc, TA_TOP | TA_CENTER) ;
    // vertical and horizontal lines
    for (i = 0 ; i < 17 ; i++)
    {
        MoveToEx (hdc, (i + 2) * cxGrid, 2 * cyGrid, NULL) ;
        LineTo   (hdc, (i + 2) * cxGrid, 19 * cyGrid) ;

        MoveToEx (hdc,      cxGrid, (i + 3) * cyGrid, NULL) ;
        LineTo   (hdc, 18 * cxGrid, (i + 3) * cyGrid) ;
    }

    // vertical and horizontal headings

    for (i = 0 ; i < 16 ; i++)
    {
        TextOut (hdc, (2 * i + 5) * cxGrid / 2, 2 * cyGrid + 2, szBuffer,
        wsprintf (szBuffer, TEXT ("%X-"), i)) ;

        TextOut (hdc, 3 * cxGrid / 2, (i + 3) * cyGrid + 2, szBuffer,
        wsprintf (szBuffer, TEXT ("-%X"), i)) ;
    }

    // characters

```



```

for (y = 0 ; y < 16 ; y++)
for (x = 0 ; x < 16 ; x++)
{
    TextOut (hdc, (2 * x + 5) * cxGrid / 2,
              (y + 3) * cyGrid + 2, szBuffer,
              wsprintf (szBuffer, TEXT ("%c"), 16 * x + y)) ;
}

EndPaint (hwnd, &ps) ;
return 0 ;

case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}

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

```

这个程式相当简单。它使用卷动列和游标移动键让您选择显示七种备用字体之一。该程式在一个网格中显示一种字体的 256 个字元。顶部的标题和网格的左侧显示字元代码的十六进位值。

在显示区域的顶部, STOKFONT 用 GetStockObject 函式显示用於选择字体的识别字。它还显示由 GetTextFace 函式得到的字体样式名称和 TEXTMETRIC 结构的 tmCharSet 栏位。这个「字元集识别字」对理解 Windows 如何处理外语版本的 Windows 是非常重要的。

如果在美国英语版本的 Windows 中执行 STOKFONT, 那么您看到的第一个画面将显示使用 OEM\_FIXED\_FONT 识别字呼叫 GetStockObject 函式得到的字体。如图 6-3 所示。

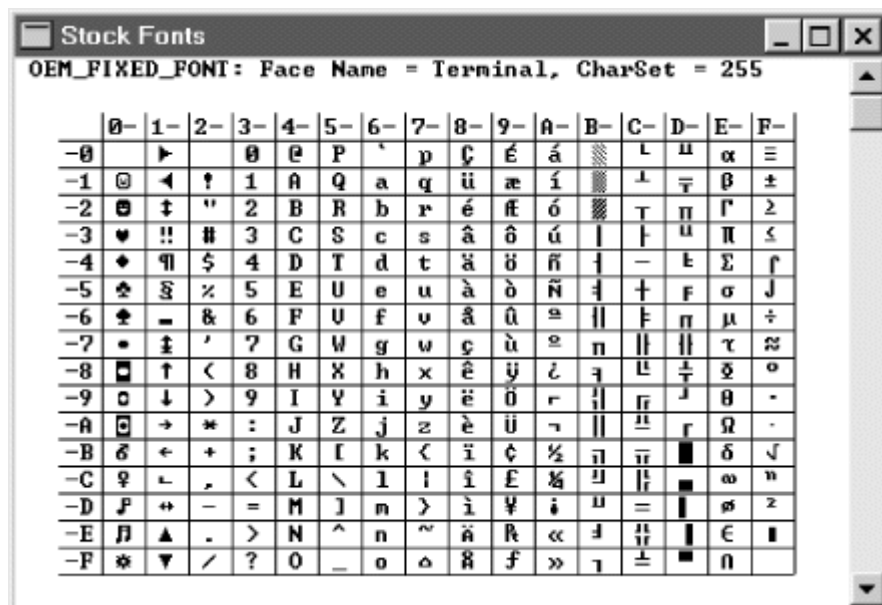


图 6-3 美国版 Windows 中的 OEM\_FIXED\_FONT

在本字元集中（与本章其他部分一样），您将看到一些 ASCII。但请记住 ASCII 是 7 位元代码，它定义了从代码 0x20 到 0x7E 的可显示字元。到 IBM 开发出 IBM PC 原型机时，8 位元位元组代码已被稳固地建立起来，因此可使用全 8 位元代码作为字元代码。IBM 决定使用一系列由线和方块组成的字元、带重音字母、希腊字母、数学符号和一些其他字元来扩展 ASCII 字元集。许多文字模式的 MS-DOS 程式在其萤幕显示中都使用绘图字元，并且许多 MS-DOS 程式都在档案中使用了一些扩展字元。

这个特殊的字元集给 Windows 最初的开发者带来了一个问题。一方面，因为 Windows 有完整的图形程式设计语言，所以线和方块字元在 Windows 中不需要。因此，这些字元使用的 48 个代码最好用於许多西欧语言所需要的附带重音字母。另一方面，IBM 字元集定义了一个无法完全忽略的标准。

因此，Windows 最初的开发者决定支援 IBM 字元集，但将其重要性降低到第二位——它们大多用於在视窗中执行的旧 MS-DOS 应用程式，和需要使用由 MS-DOS 应用程式建立档案的 Windows 程式。Windows 应用程式不使用 IBM 字元集，并且随著时间的推移，其重要性日渐衰退。然而，如果需要，您还是可以使用。在此环境下，「OEM」指的就是「IBM」。

（您应知道外语版本的 Windows 不必支援与美国英语版相同的 OEM 字元集。其他国家有其自己的 MS-DOS 字元集。这是个独立的问题，就不在本书中讨论了。）

因为 IBM 字元集被认为不适合 Windows，於是选择了另一种扩展字元集。此字元集称作「ANSI 字元集」，由美国国家标准协会(American National Standards Institute)制定，但它实际上是 ISO(International Standards Organization, 国际标准化组织)标准，也就是 ISO 标准 8859。它还称为 Latin 1、Western European、或者内码表 1252。图 6-4 显示了 ANSI 字元集的一个版本——美国英语版 Windows 的系统字体。

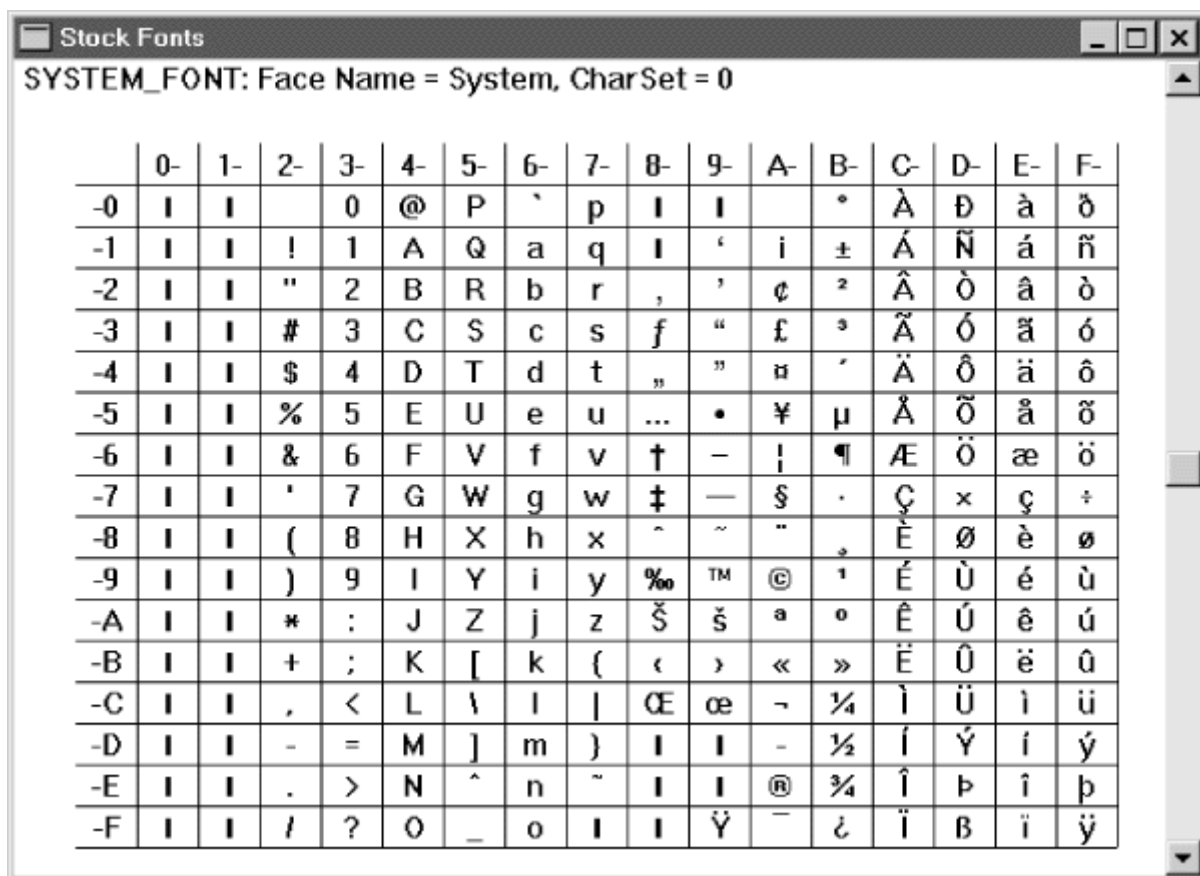


图 6-4 美国版 Windows 中的 SYSTEM\_FONT

粗的垂直条表示这些字元代码没有定义。注意，代码 0x20 到 0x7E 还是 ASCII。此外，ASCII 控制字元（0x00 到 0x1F 以及 0x7F）并不是可显示字元。它们本应如此。

代码 0xC0 到 0xFF 使得 ANSI 字元集对外语版 Windows 来说非常重要。这些代码提供 64 个在西欧语言中普遍使用的字元。字元 0xA0，看起来像空格，但实际上定义为非断开空格，例如「WW II」中的空格。

之所以说这是 ANSI 字元集的「一个版本」，是因为存在代码 0x80 到 0x9F 的字元。等宽的系统字体只包括其中的两个字元，如图 6-5 所示。

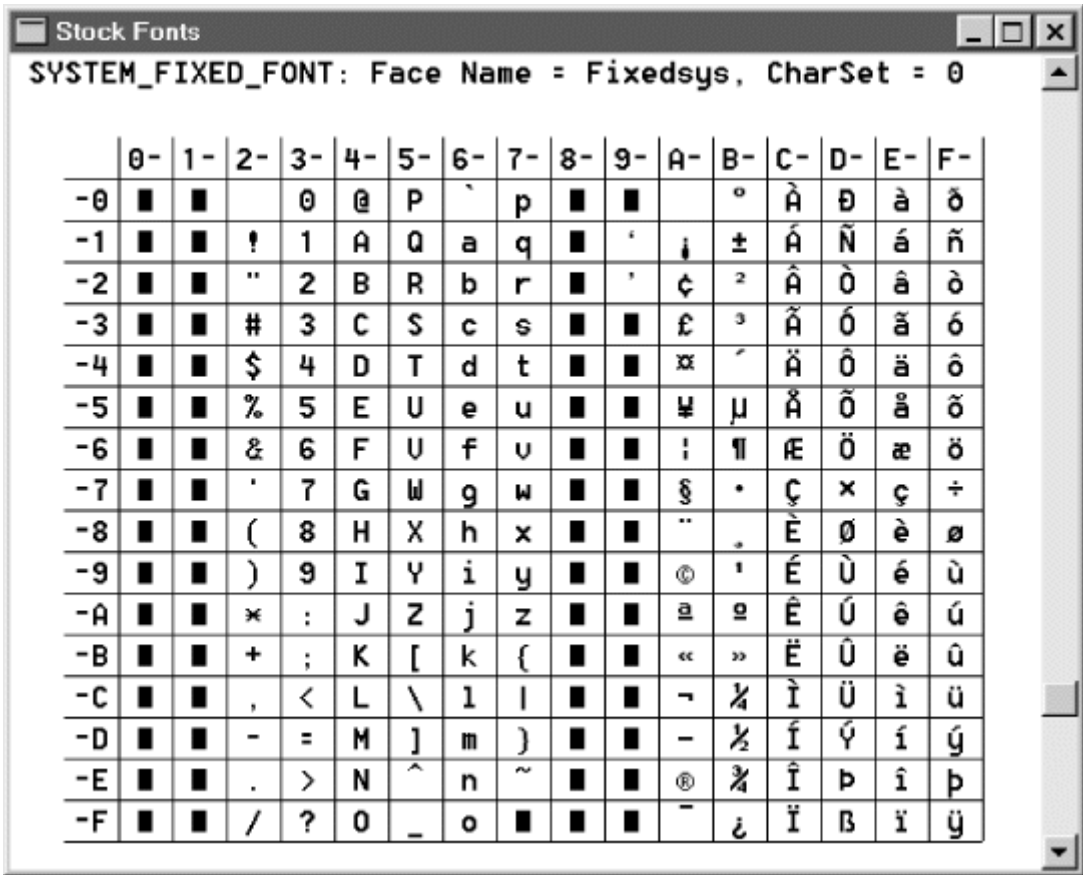


图 6-5 美国版 Windows 中的 SYSTEM\_FIXED\_FONT

在 Unicode 中,代码 0x0000 到 0x007F 与 ASCII 相同,代码 0x0080 到 0x009F 复制了 0x0000 到 0x001F 的控制字元,代码 0x00A0 到 0x00FF 与 Windows 中使用的 ANSI 字元集相同。

如果执行德语版的 Windows , 那么当您用 SYSTEM\_FONT 或者 SYSTEM\_FIXED\_FONT 识别字来呼叫 GetStockObject 函数时会得到同样的 ANSI 字元集。其他西欧版 Windows 也是如此。ANSI 字元集中含有这些语言所需要的所有字元。

不过, 当您执行希腊版的 Windows 时, 内定的字元集就改变了。相反地, SYSTEM\_FONT 如图 6-6 所示。

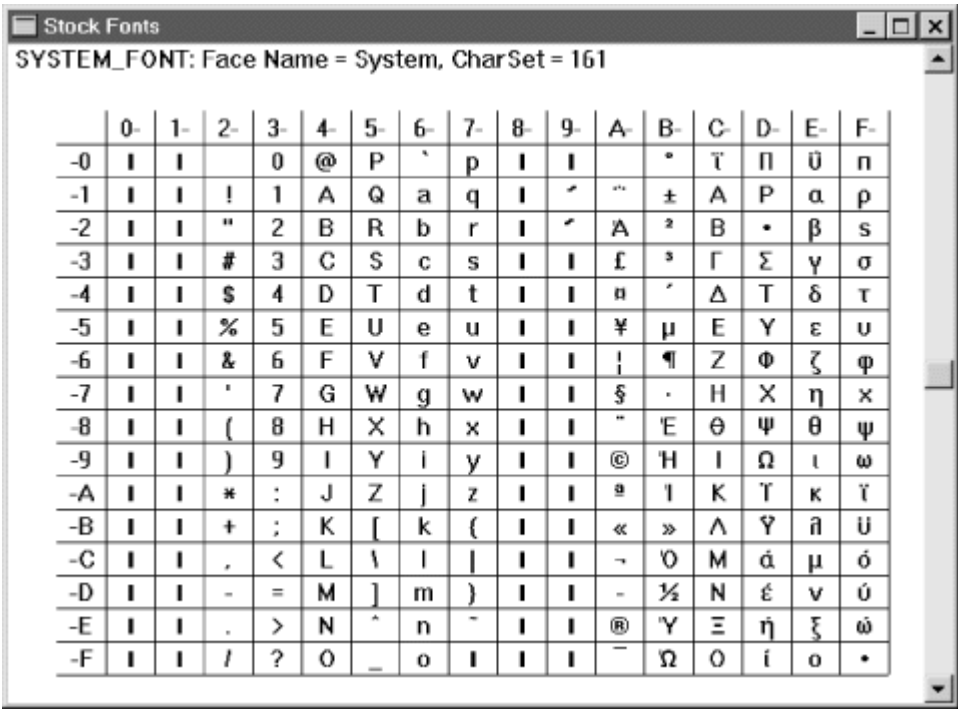


图 6-6 希腊版 Windows 中的 SYSTEM\_FONT

SYSTEM\_FIXED\_FONT 有同样的字元。注意从 0xC0 到 0xFF 的代码。这些代码包含希腊字母表中的大写字母和小写字母。当您执行俄语版 Windows 时，内定的字元集如图 6-7 所示。

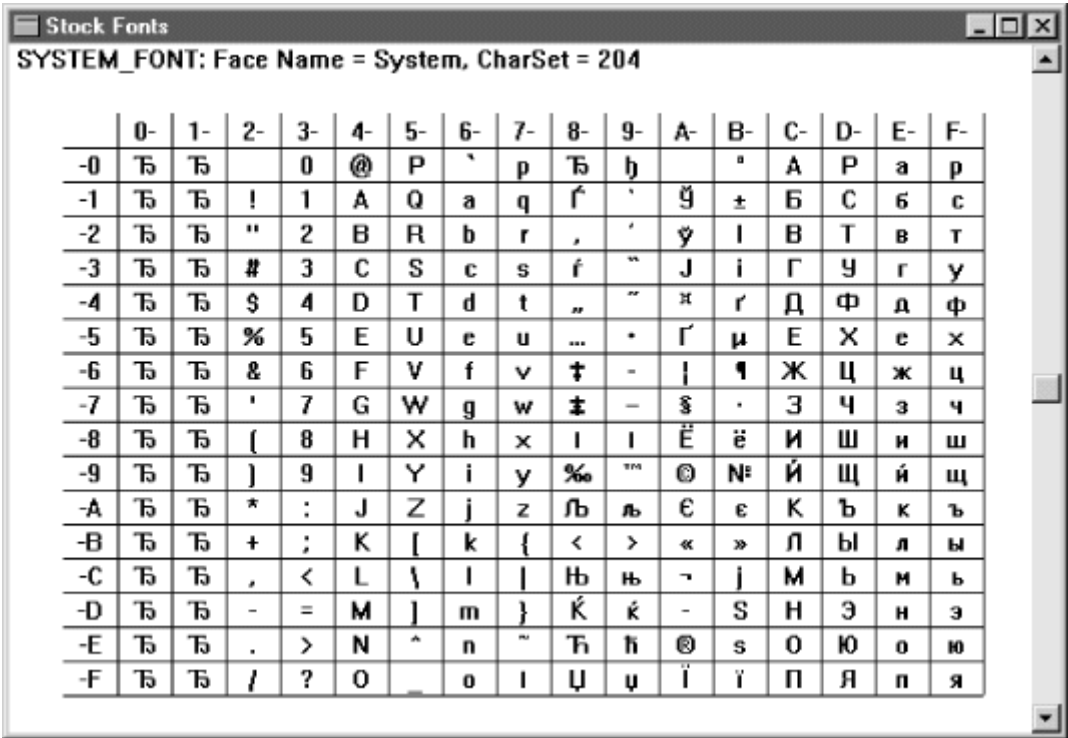


图 6-7 俄语版 Windows 中的 SYSTEM\_FONT

此外， 注意斯拉夫字母表中的大写和小写字母占用了代码 0xC0 和 0xFF。

图 6-8 显示了日语版 Windows 的 SYSTEM\_FONT。从 0xA5 到 0xDF 的字元都是片假名字母表的一部分。

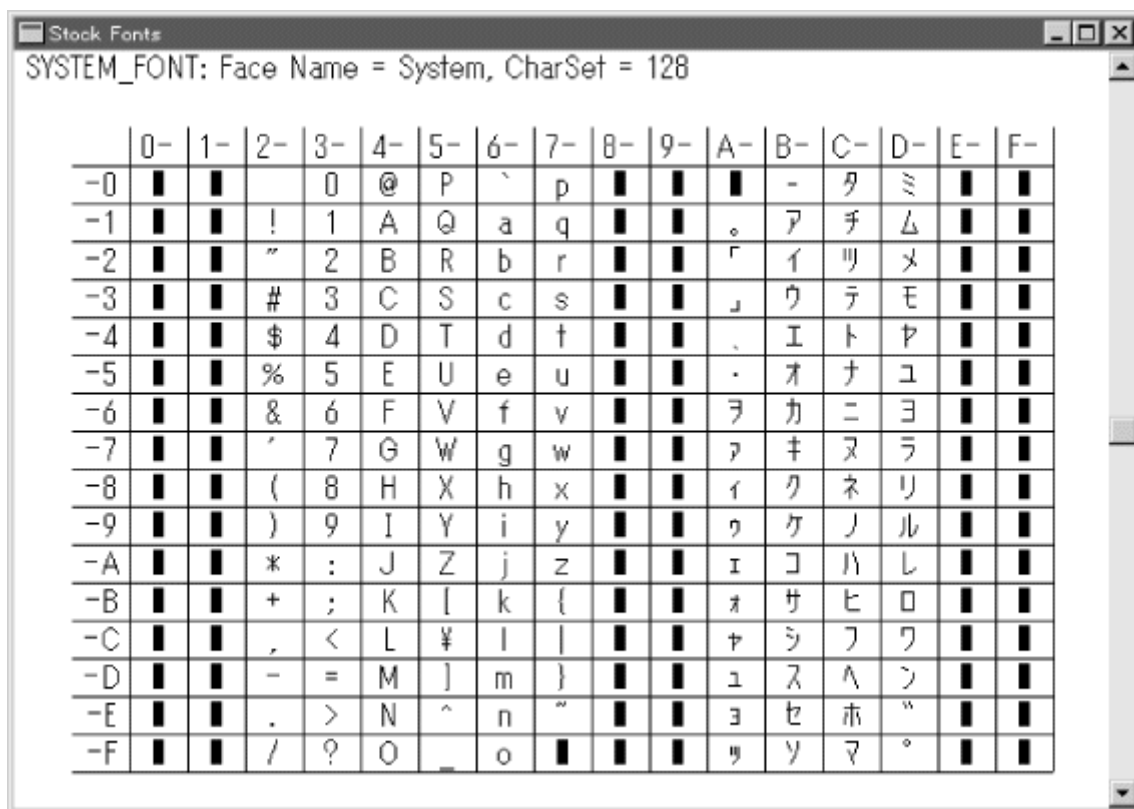


图 6-8 日语版 Windows 中的 SYSTEM\_FONT

图 6-8 所示的日文系统字体不同於前面显示的那些，因为它实际上是双位元组字元集 (DBCS)，称为「Shift-JIS」（「JIS」代表日本工业标准，Japanese Industrial Standard）。从 0x81 到 0x9F 以及从 0xE0 到 0xFF 的大多数字元代码实际上只是双位元组代码的第一个位元组，其第二个位元组通常在 0x40 到 0xFC 的范围内（关于这些代码的完整表格，请参见 Nadine Kano 书中的附录 G）。

现在，我们就可以看看 KEYVIEW1 中的问题在哪里：如果您安装了希腊键盘布局并键入『abcde』，不考虑执行的 Windows 版本，Windows 将产生 WM\_CHAR 讯息和字元代码 0xE1、0xE2、0xF8、0xE4 和 0xE5。但只有执行带有希腊系统字体的希腊版 Windows 时，这些字元代码才能与  $\tau$ 、 $\beta$ 、 $\psi$ 、 $\delta$  和  $\varepsilon$  相对应。

如果您安装了俄语键盘布局并敲入『abcde』，不考虑所使用的 Windows 版本，Windows 将产生 WM\_CHAR 讯息和字元代码 0xF4、0xE8、0xF1、0xE2 和 0xF3。但只有在使用俄语版 Windows 或者使用斯拉夫字母表的其他语言版，并且使用斯拉夫系统字体时，这些字元代码才会与字元  $\phi$ 、и、с、в 和 у 相对应。

如果您安装了德语键盘布局并按下=键（或者位於同一位置的键），然後按下 a、e、i、o 或者 u 键，不考虑使用的 Windows 版本，Windows 将产生 WM\_CHAR 讯息和字元代码 0xE1、0xE9、0xED、0xF3 和 0xFA。只有执行西欧版或者美国版的 Windows 时，也就是说有西欧系统字体，这些字元代码才会和字元  $\&\nbsp\&$ 、á、é、í、ó 和 ú 相对应。

如果安装了美国英语键盘布局，则您可在键盘上键入任何字元，Windows 将

产生 WM\_CHAR 讯息以及与字元正确匹配的字元代码。

## Unicode 怎么样？

我在第二章谈到过 Windows NT 支援的 Unicode 有助於为国际市场程式写作。让我们编译一下定义了 UNICODE 识别字的 KEYVIEW1, 并在不同版本的 Windows NT 下执行 (在本书附带的光碟中, Unicode 版的 KEYVIEW1 位於 DEBUG 目录中)。

如果程式编译时定义了 UNICODE 识别字, 则「KeyView1」视窗类别就用 RegisterClassW 函式注册, 而不是 RegisterClassA 函式。这意味著任何带有字元或文字资料的讯息传递给 WndProc 时都将使用 16 位元字元而不是 8 位元字元。特别是 WM\_CHAR 讯息, 将传递 16 位元字元代码而不是 8 位元字元代码。

请在美国英语版的 Windows NT 下执行 Unicode 版的 KEYVIEW1。这里假定您已经安装了至少三种我们试验过的键盘布局——即德语、希腊语和俄语。

使用美国英语版的 Windows NT, 并安装了英语或者德语的键盘布局, Unicode 版的 KEYVIEW1 在工作时将与非 Unicode 版相同。它将接收相同的字元代码 (所有 0xFF 或者更低的值), 并显示同样正确的字元。这是因为最初的 256 个 Unicode 字元与 Windows 中使用的 ANSI 字元集相同。

现在切换到希腊键盘布局, 并键入『abcde』。WM\_CHAR 讯息将含有 Unicode 字元代码 0x03B1、0x03B2、0x03C8、0x03B4 和 0x03B5。注意, 我们先看到的字元代码值比 0xFF 高。这些 Unicode 字元代码与希腊字母  $\alpha$ 、 $\beta$ 、 $\psi$ 、 $\delta$  和  $\epsilon$  相对应。不过, 所有这五个字元都显示为方块! 这是因为 SYSTEM\_FIXED\_FONT 只含有 256 个字元。

现在切换到俄语键盘布局, 并键入『abcde』。KEYVIEW1 显示 WM\_CHAR 讯息和 Unicode 字元代码 0x0444、0x0438、0x0441、0x0432 和 0x0443, 这些字元对应於斯拉夫字母  $\phi$ 、 $\mu$ 、 $\sigma$ 、 $\nu$  和  $\gamma$ 。不过, 所有这五个字母也显示为实心方块。

简言之, 非 Unicode 版的 KEYVIEW1 显示错误字元的地方, Unicode 版的 KEYVIEW1 就显示实心方块, 以表示目前的字体没有那种特殊字元。虽然我不愿说 Unicode 版的 KEYVIEW1 是非 Unicode 版的改进, 但事实确实如此。非 Unicode 版显示错误字元, 而 Unicode 版不会这样。

Unicode 和非 Unicode 版 KEYVIEW1 的不同之处主要在两个方面。

首先, WM\_CHAR 讯息伴随一个 16 位元字元代码, 而不是 8 位元字元代码。在非 Unicode 版本的 KEYVIEW1 中, 8 位元字元代码的含义取决於目前活动的键盘布局。如果来自德语键盘, 则 0xE1 代码表示  $\acute{a}$ , 如果来自希腊语键盘则代表  $\alpha$ , 如果来自俄语键盘则代表  $\sigma$ 。在 Unicode 版本程式中, 16 位元字元代码的含义

很明确：**a** 字元是 **0x00E1**，**α** 字元是 **0x03B1**，而 **σ** 字元是 **0x0431**。

第二，Unicode 的 TextOutW 函式显示的字元依据 16 位元字元代码，而不是非 Unicode 的 TextOutA 函式的 8 位元字元代码。因为这些 16 位元字元代码含义明确，GDI 可以确定目前在装置内容中选择的字体是否可显示每个字元。

在美国英语版 Windows NT 下执行 Unicode 版的 KEYVIEW1 多少让人感到有些迷惑，因为它所显示的就好像 GDI 只显示了 0x0000 到 0x00FF 之间的字元代码，而没有显示高於 0x00FF 的代码。也就是说，只是在字元代码和系统字体中 256 个字元之间简单的一对一映射。

然而，如果安装了希腊或者俄语版的 Windows NT，您将发现情况就大不一样了。例如，如果安装了希腊版的 Windows NT，则美国英语、德语、希腊语和俄语键盘将会产生与美国英语版 Windows NT 同样的 Unicode 字元代码。不过，希腊版的 Windows NT 将不显示德语重音字元或者俄语字元，因为这些字元并不在希腊系统字体中。同样，俄语版的 Windows NT 也不显示德语重音字元或者希腊字元，因为这些字元也不在俄语系统字体中。

其中，Unicode 版的 KEYVIEW1 的区别在日语版 Windows NT 下更具戏剧性。您从 IME 输入日文字元，这些字元可以正确显示。唯一的问题是格式：因为日文字元通常看起来非常复杂，它们的显示宽度是其他字元的两倍。

## TrueType 和大字体

我们使用的点阵字体（在日文版 Windows 中带有附加字体）最多包括 256 个字元。这是我们所希望的，因为当假定字元代码是 8 位元时，点阵字体档案的格式就跟早期 Windows 时代的样子一样了。这就是为什么当我们使用 SYSTEM\_FONT 或者 SYSTEM\_FIXED\_FONT 时，某些语言中一些字元总不能正确显示（日本系统字体有点不同，因为它是双位元组字元集；大多数字元实际上保存在 TrueType 集合档案中，档案副档名是 .TTC）。

TrueType 字体包含的字元可以多於 256 个。并不是所有 TrueType 字体中的字元都多於 256 个，但 Windows 98 和 Windows NT 中的字体包含多於 256 个字元。或者，安装了多语系支援後，TrueType 字体中也包含多於 256 个字元。在「[控制台](#)」的「[新增/删除程式](#)」中，单击「[Windows 安装程式](#)」页面标签，并确保选中了「[多语系支援](#)」。这个多语系支援包括五个字元集：波罗的海语系、中欧语系、斯拉夫语系、希腊语系和土耳其语系。波罗的海语系字元集用於爱沙尼亚语、拉脱维亚语和立陶宛语。中欧字元集用於阿尔巴尼亚语、捷克语、克罗地亚语、匈牙利语、波兰语、罗马尼亚语、斯洛伐克语和斯洛文尼亚语。斯拉夫字元集用於保加利亚语、白俄罗斯语、俄语、塞尔维亚语和乌



克兰语。

Windows 98 中的 TrueType 字体支援这五种字元集，再加上西欧 (ANSI) 字元集，西欧字元集实际上用於其他所有语言，但远东语言 (汉语、日语和朝鲜语) 除外。支援多种字元集的 TrueType 字体有时也称为「大字体」。在这种情况下下的「大」并不是指字元的大小，而是指数量。

即使在非 Unicode 程式中也可利用大字体，这意味著可以用大字体显示几种不同字母表中的字元。然而，为了要将得到的字体选进装置内容，还需要 GetStockObject 以外的函式。

函式 CreateFont 和 CreateFontIndirect 建立了一种逻辑字体，这与 CreatePen 建立逻辑画笔以及 CreateBrush 建立逻辑画刷的方式类似。CreateFont 用 14 个参数描述要建立的字体。CreateFontIndirect 只有一个参数，但该参数是指向 LOGFONT 结构的指标。LOGFONT 结构有 14 个栏位，分别对应於 CreateFont 函式的参数。我将在第十七章详细讨论这些函式。现在，让我们看一下 CreateFont 函式，但我们只注意其中两个参数，其他参数都设定为 0。

如果需要等宽字体 (就像 KEYVIEW1 程式中使用的)，将 CreateFont 的第 13 个参数设定为 FIXED\_PITCH。如果需要非内定字元集的字体 (这也是我们所需要的)，将 CreateFont 的第 9 个参数设定为某个「字元集 ID」。此字元集 ID 将是 WINGDI.H 中定义的下列值之一。我已给出注释，指出和这些字元集相关的内码表：

#define ANSI_CHARSET	0	// 1252 Latin 1 (ANSI)
#define DEFAULT_CHARSET	1	
#define SYMBOL_CHARSET	2	
#define MAC_CHARSET	77	
#define SHIFTJIS_CHARSET	128	// 932 (DBCS, 日本)
#define HANGEUL_CHARSET	129	// 949 (DBCS, 韩文)
#define HANGUL_CHARSET	129	// " "
#define JOHAB_CHARSET	130	// 1361 (DBCS, 韩文)
#define GB2312_CHARSET	134	// 936 (DBCS, 简体中文)
#define CHINESEBIG5_CHARSET	136	// 950 (DBCS, 繁体中文)
#define GREEK_CHARSET	161	// 1253 希腊文
#define TURKISH_CHARSET	162	// 1254 Latin 5 (土耳其文)
#define VIETNAMESE_CHARSET	163	// 1258 越南文
#define HEBREW_CHARSET	177	// 1255 希伯来文
#define ARABIC_CHARSET	178	// 1256 阿拉伯文
#define BALTIC_CHARSET	186	// 1257 波罗的海字集

#define RUSSIAN_CHARSET	204	// 1251 俄文 (斯拉夫语系)
#define THAI_CHARSET	222	// 874 泰文
#define EASTEUROPE_CHARSET	238	// 1250 Latin 2 (中欧语系)
#define OEM_CHARSET	255	// 地区自订

为什么 Windows 对同一个字元集有两个不同的 ID: 字元集 ID 和内码表 ID? 这只是 Windows 中的一种怪癖。注意, 字元集 ID 只需要 1 位元组的储存空间, 这是 LOGFONT 结构中字元集栏位的大小 (试回忆 Windows 1.0 时期, 记忆体和储存空间有限, 每个位元组都必须斤斤计较)。注意, 有许多不同的 MS-DOS 内码表用於其他国家, 但只有一种字元集 ID——OEM\_CHARSET——用於 MS-DOS 字元集。

您还会注意到, 这些字元集的值与 STOKFONT 程式最上头的「CharSet」值一致。在美国英语版 Windows 中, 我们看到常备字体的字元集 ID 是 0 (ANSI\_CHARSET) 和 255 (OEM\_CHARSET)。希腊版 Windows 中的是 161 (GREEK\_CHARSET), 在俄语版中的是 204 (RUSSIAN\_CHARSET), 在日语版中是 128 (SHIFTJIS\_CHARSET)。

在上面的代码中, DBCS 代表双位元组字元集, 用於远东版的 Windows。其他版的 Windows 不支援 DBCS 字体, 因此不能使用那些字元集 ID。

CreateFont 传回 HFONT 值——逻辑字体的代号。您可以使用 SelectObject 将此字体选进装置内容。实际上, 您必须呼叫 DeleteObject 来删除您建立的所有逻辑字体。

大字体解决方案的其他部分是 WM\_INPUTLANGCHANGE 讯息。一旦您使用桌面下端的突现式功能表来改变键盘布局, Windows 都会向您的视窗讯息处理程式发送 WM\_INPUTLANGCHANGE 讯息。wParam 讯息参数是新键盘布局的字元集 ID。

程式 6-4 所示的 KEYVIEW2 程式实作了键盘布局改变时改变字体的逻辑。

#### 程式 6-4 KEYVIEW2

```
KEYVIEW2.C
/*-----
--
KEYVIEW2.C -- Displays Keyboard and Character Messages
(c) Charles Petzold, 1998
-----
*/

#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
```

```

static TCHAR szAppName[] = TEXT ("KeyView2") ;
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 ("Keyboard Message Viewer #2"),
                    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 DWORD dwCharSet = DEFAULT_CHARSET ;
    static int   cxClientMax, cyClientMax, cxClient, cyClient, cxChar, cyChar ;
    static int   cLinesMax, cLines ;
    static PMSG  pmsg ;
    static RECT  rectScroll ;

```

```

static TCHAR szTop[] = TEXT ("Message Key Char ")
                        TEXT ("Repeat Scan Ext ALT Prev
Tran");
static TCHAR szUnd[] = TEXT ("_____")
                        TEXT ("_____
_____");

static TCHAR * szFormat[2] = {
    TEXT ("%13s %3d %-15s%c%6u %4d %3s %3s %4s %4s"),
    TEXT ("%13s 0x%04X%1s%c %6u %4d %3s %3s %4s %4s") };

static TCHAR * szYes = TEXT ("Yes");
static TCHAR * szNo = TEXT ("No");
static TCHAR * szDown = TEXT ("Down");
static TCHAR * szUp = TEXT ("Up");

static TCHAR * szMessage [] = {
    TEXT ("WM_KEYDOWN"), TEXT ("WM_KEYUP"),
    TEXT ("WM_CHAR"), TEXT ("WM_DEADCHAR"),
    TEXT ("WM_SYSKEYDOWN"), TEXT ("WM_SYSKEYUP"),
    TEXT ("WM_SYSCHAR"), TEXT ("WM_SYSDEADCHAR") };

HDC hdc;
int i, iType;
PAINTSTRUCT ps;
TCHAR szBuffer[128], szKeyName [32];
TEXTMETRIC tm;

switch (message)
{
case WM_INPUTLANGCHANGE:
    dwCharSet = wParam;
    // fall through
case WM_CREATE:
case WM_DISPLAYCHANGE:
    // Get maximum size of client area
    cxClientMax = GetSystemMetrics (SM_CXMAXIMIZED);
    cyClientMax = GetSystemMetrics (SM_CYMAXIMIZED);

    // Get character size for fixed-pitch font
    hdc = GetDC (hwnd);
    SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                dwCharSet, 0, 0, 0, FIXED_PITCH,
NULL));
    GetTextMetrics (hdc, &tm);
    cxChar = tm.tmAveCharWidth;
    cyChar = tm.tmHeight;

    DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT)));

```

```

ReleaseDC (hwnd, hdc) ;

        // Allocate memory for display lines
if (pmsg)
        free (pmsg) ;
cLinesMax = cyClientMax / cyChar ;
pmsg = malloc (cLinesMax * sizeof (MSG)) ;
cLines = 0 ;
        // fall through
case WM_SIZE:
if (message == WM_SIZE)
{
        cxClient          = LOWORD (lParam) ;
        cyClient          = HIWORD (lParam) ;
}

        // Calculate scrolling rectangle

rectScroll.left          = 0 ;
rectScroll.right = cxClient ;
rectScroll.top           = cyChar ;
rectScroll.bottom= cyChar * (cyClient / cyChar) ;

InvalidateRect (hwnd, NULL, TRUE) ;

if (message == WM_INPUTLANGCHANGE)
        return TRUE ;
return 0 ;

case WM_KEYDOWN:
case WM_KEYUP:
case WM_CHAR:
case WM_DEADCHAR:
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
case WM_SYSDEADCHAR:
        // Rearrange storage array
for (i = cLinesMax - 1 ; i > 0 ; i--)
{
        pmsg[i] = pmsg[i - 1] ;
}

        // Store new message
pmsg[0].hwnd = hwnd ;
pmsg[0].message = message ;
pmsg[0].wParam = wParam ;
pmsg[0].lParam = lParam ;

cLines = min (cLines + 1, cLinesMax) ;

```

```

        // Scroll up the display
        ScrollWindow (hwnd, 0, -cyChar, &rectScroll, &rectScroll) ;
        break ;      // ie, call DefWindowProc so Sys messages work

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

    SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0, 0,
        dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;
    SetBkMode (hdc, TRANSPARENT) ;
    TextOut (hdc, 0, 0, szTop, strlen (szTop)) ;
    TextOut (hdc, 0, 0, szUnd, strlen (szUnd)) ;

    for (i = 0 ; i < min (cLines, cyClient / cyChar - 1) ; i++)
    {
        iType =      pmsg[i].message == WM_CHAR ||
                    pmsg[i].message == WM_SYSCHAR ||
                    pmsg[i].message == WM_DEADCHAR ||
                    pmsg[i].message == WM_SYSDEADCHAR ;

        GetKeyNameText (pmsg[i].lParam, szKeyName,
            sizeof (szKeyName) / sizeof (TCHAR)) ;

        TextOut (hdc, 0, (cyClient / cyChar - 1 - i) * cyChar, szBuffer,
            wsprintf ( szBuffer, szFormat [iType],
                szMessage [pmsg[i].message - WM_KEYFIRST],
                pmsg[i].wParam,
                (PTSTR) (iType ? TEXT (" ") : szKeyName),
                (TCHAR) (iType ? pmsg[i].wParam : ' '),
                LOWORD (pmsg[i].lParam),
                HIWORD (pmsg[i].lParam) & 0xFF,
                0x01000000 & pmsg[i].lParam ? szYes : szNo,
                0x20000000 & pmsg[i].lParam ? szYes : szNo,
                0x40000000 & pmsg[i].lParam ? szDown : szUp,
                0x80000000 & pmsg[i].lParam ? szUp : szDown));
    }
    DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
    EndPaint (hwnd, &ps) ;
    return 0 ;

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

```

注意，键盘输入语言改变後，KEYVIEW2 就清除画面并重新分配储存空间。

这样做有两个原因：第一，因为 KEYVIEW2 并不是某种字体专用的，当输入语言改变时字体文字的大小也会改变。程式需要根据新字元大小重新计算某些变数。第二，在接收每个字元讯息时，KEYVIEW2 并不有效地保留字元集 ID。因此，如果键盘输入语言改变了，而且 KEYVIEW2 需要重画显示区域时，所有的字元将用新字体显示。

第十七章将详细讨论字体和字元集。如果您想深入研究国际化问题，可以在 /Platform SDK/Windows Base Services/International Features 找到需要的文件，还有许多基础资讯则位於 /Platform SDK/Windows Base Services/General Library/String Manipulation。

## 插入符号（不是游标）

当您往程式中输入文字时，通常有一个底线、竖条或者方框来指示输入的下一个字元将出现在萤幕上的位置。这个标志通常称为「游标」，但是在 Windows 下写程式，您必须改变这个习惯。在 Windows 中，它称为「插入符号」。「游标」是指表示滑鼠位置的那个点阵图图像。

## 插入符号函式

主要有五个插入符号函式：

- CreateCaret 建立与视窗有关的插入符号
- SetCaretPos 在视窗中设定插入符号的位置
- ShowCaret 显示插入符号
- HideCaret 隐藏插入符号
- DestroyCaret 撤消插入符号

另外还有取得插入符号目前位置 (GetCaretPos) 和取得以及设定插入符号闪烁时间 (GetCaretBlinkTime 和 SetCaretBlinkTime) 的函式。

在 Windows 中，插入符号定义为水平线、与字元大小相同的方框，或者与字元同高的竖线。如果使用调和字体，例如 Windows 内定的系统字体，则推荐使用竖线插入符号。因为调和字体中的字元没有固定大小，水平线或方框不能设定为字元的大小。

如果程式中需要插入符号，那么您不应该简单地在视窗讯息处理程式的 WM\_CREATE 讯息处理期间建立它，然後在 WM\_DESTROY 讯息处理期间撤消。其原因显而易见：一个讯息伫列只能支援一个插入符号。因此，如果您的程式有多个视窗，那么各个视窗必须有效地共用相同的插入符号。

其实，它并不像听起来那么多限制。您再想想就会发现，只有在视窗有输

入焦点时，视窗内显示插入符号才有意义。事实上，闪烁的插入符号只是一种视觉提示：您可以在程式中输入文字。因为任何时候都只有一个视窗拥有输入焦点，所以多个视窗同时都有闪烁的插入符号是没有意义的。

通过处理 WM\_SETFOCUS 和 WM\_KILLFOCUS 讯息，程式就可以确定它是否有输入焦点。正如名称所暗示的，视窗讯息处理程式在有输入焦点的时候接收到 WM\_SETFOCUS 讯息，失去输入焦点的时候接收到 WM\_KILLFOCUS 讯息。这些讯息成对出现：视窗讯息处理程式在接收到 WM\_KILLFOCUS 讯息之前将一直接收到 WM\_SETFOCUS 讯息，并且在视窗打开期间，此视窗总是接收到相同数量的 WM\_SETFOCUS 和 WM\_KILLFOCUS 讯息。

使用插入符号的主要规则很简单：视窗讯息处理程式在 WM\_SETFOCUS 讯息处理期间呼叫 CreateCaret，在 WM\_KILLFOCUS 讯息处理期间呼叫 DestroyCaret。

这里还有几条其他规则：插入符号刚建立时是隐蔽的。如果想使插入符号可见，那么您在呼叫 CreateCaret 之後，视窗讯息处理程式还必须呼叫 ShowCaret。另外，当视窗讯息处理程式处理一条非 WM\_PAINT 讯息而且希望在视窗内绘制某些东西时，它必须呼叫 HideCaret 隐藏插入符号。在绘制完毕後，再呼叫 ShowCaret 显示插入符号。HideCaret 的影响具有累积效果，如果多次呼叫 HideCaret 而不呼叫 ShowCaret，那么只有呼叫 ShowCaret 相同次数时，才能看到插入符号。

## TYPYR 程式

程式 6-5 所示的 TYPYR 程式使用了本章讨论的所有内容，您可以认为 TYPYR 是一个相当简单的文字编辑器。在视窗中，您可以输入字元，用游标移动键（也可以称为插入符号移动键）来移动游标（I 型标），按下 Escape 键清除视窗的内容等。缩放视窗、改变键盘输入语言时都会清除视窗的内容。本程式没有卷动，没有文字寻找和定位功能，不能储存档案，没有拼写检查，但它确实是写作一个文字编辑器的开始。

### 程式 6-5 TYPYR

```
TYPYR.C
/*-----
   TYPYR.C --      Typing Program
                        (c) Charles Petzold, 1998
   -----*/

#include <windows.h>

#define BUFFER(x,y) *(pBuffer + y * cxBuffer + x)
```



```

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR        szAppName[] = TEXT ("Typer") ;
    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 ("Typing Program"),
                           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 DWORD        dwCharSet = DEFAULT_CHARSET ;
    static int          cxChar, cyChar, cxClient, cyClient, cxBuffer, cyBuffer,

```

```

        xCaret, yCaret ;

static TCHAR *   pBuffer = NULL ;
HDC              hdc ;
int              x, y, i ;
PAINTSTRUCT      ps ;
TEXTMETRIC       tm ;

switch (message)
{
case WM_INPUTLANGCHANGE:
    dwCharSet = wParam ;
    // fall through
case WM_CREATE:
    hdc = GetDC (hwnd) ;
    SelectObject (   hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                     dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

    GetTextMetrics (hdc, &tm) ;
    cxChar = tm.tmAveCharWidth ;
    cyChar = tm.tmHeight ;

    DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
    ReleaseDC (hwnd, hdc) ;
    // fall through
case WM_SIZE:
    // obtain window size in pixels

    if (message == WM_SIZE)
    {
        cxClient = LOWORD (lParam) ;
        cyClient = HIWORD (lParam) ;
    }

    // calculate window size in characters

    cxBuffer = max (1, cxClient / cxChar) ;
    cyBuffer = max (1, cyClient / cyChar) ;

    // allocate memory for buffer and clear it

    if (pBuffer != NULL)
        free (pBuffer) ;

    pBuffer = (TCHAR *) malloc (cxBuffer * cyBuffer * sizeof (TCHAR)) ;

    for (y = 0 ; y < cyBuffer ; y++)
        for (x = 0 ; x < cxBuffer ; x++)
            BUFFER(x,y) = ' ' ;

```

```
        // set caret to upper left corner

xCaret = 0 ;
yCaret = 0 ;

if (hwnd == GetFocus ())
    SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;

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

case WM_SETFOCUS:
    // create and show the caret
    CreateCaret (hwnd, NULL, cxChar, cyChar) ;
    SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
    ShowCaret (hwnd) ;
    return 0 ;

case WM_KILLFOCUS:
    // hide and destroy the caret
    HideCaret (hwnd) ;
    DestroyCaret () ;
    return 0 ;

case WM_KEYDOWN:
    switch (wParam)
    {
    case VK_HOME:
        xCaret = 0 ;
        break ;

    case VK_END:
        xCaret = cxBuffer - 1 ;
        break ;

    case VK_PRIOR:
        yCaret = 0 ;
        break ;

    case VK_NEXT:
        yCaret = cyBuffer - 1 ;
        break ;

    case VK_LEFT:
        xCaret = max (xCaret - 1, 0) ;
        break ;

    case VK_RIGHT:
```

```

        xCaret = min (xCaret + 1, cxBuffer - 1) ;
        break ;

    case VK_UP:
        yCaret = max (yCaret - 1, 0) ;
        break ;

    case VK_DOWN:
        yCaret = min (yCaret + 1, cyBuffer - 1) ;
        break ;

    case VK_DELETE:
        for (x = xCaret ; x < cxBuffer - 1 ; x++)
            BUFFER (x, yCaret) = BUFFER (x + 1, yCaret) ;

        BUFFER (cxBuffer - 1, yCaret) = ' ' ;

        HideCaret (hwnd) ;
        hdc = GetDC (hwnd) ;

        SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                         dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;
        TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
                 & BUFFER (xCaret, yCaret),
                 cxBuffer - xCaret) ;

        DeleteObject (SelectObject (hdc, GetStockObject
(SYSTEM_FONT))) ;

        ReleaseDC (hwnd, hdc) ;
        ShowCaret (hwnd) ;
        break ;
    }
    SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
    return 0 ;

case WM_CHAR:
    for (i = 0 ; i < (int) LOWORD (lParam) ; i++)
    {
        switch (wParam)
        {
            case '\\b': // backspace
                if (xCaret > 0)
                {
                    xCaret-- ;
                    SendMessage (hwnd, WM_KEYDOWN, VK_DELETE,
1) ;
                }
                break ;
        }
    }

```

```

case '\t':                                // tab
    do
    {
        SendMessage (hwnd, WM_CHAR, ' ', 1) ;
    }
    while (xCaret % 8 != 0) ;
    break ;

case '\n':                                // line feed
    if (++yCaret == cyBuffer)
        yCaret = 0 ;
    break ;

case '\r':                                // carriage return
    xCaret = 0 ;

    if (++yCaret == cyBuffer)
        yCaret = 0 ;
    break ;

case '\x1B':                              // escape
    for (y = 0 ; y < cyBuffer ; y++)
        for (x = 0 ; x < cxBuffer ; x++)
            BUFFER (x, y) = ' ' ;

    xCaret = 0 ;
    yCaret = 0 ;

    InvalidateRect (hwnd, NULL, FALSE) ;
    break ;

default:                                  // character codes
    BUFFER (xCaret, yCaret) = (TCHAR) wParam ;

    HideCaret (hwnd) ;
    hdc = GetDC (hwnd) ;

    SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                   dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;
    TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
             & BUFFER (xCaret, yCaret), 1) ;
    DeleteObject (
        SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
    ReleaseDC (hwnd, hdc) ;
    ShowCaret (hwnd) ;

    if (++xCaret == cxBuffer)
    {

```

```

        xCaret = 0 ;
        if (++yCaret == cyBuffer)
            yCaret = 0 ;
    }
    break ;
}

SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
return 0 ;

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

    SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                   dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;
    for (y = 0 ; y < cyBuffer ; y++)
        TextOut (hdc, 0, y * cyChar, & BUFFER(0,y), cxBuffer) ;
    DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
    EndPaint (hwnd, &ps) ;
    return 0 ;

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

```

为了简单起见, TYPED 程式使用一种等宽字体, 因为编写处理调和字体的文字编辑器要困难得多。程式在好几个地方取得装置内容: 在 WM\_CREATE 讯息处理期间, 在 WM\_KEYDOWN 讯息处理期间, 在 WM\_CHAR 讯息处理期间以及在 WM\_PAINT 讯息处理期间, 每次都通过 GetStockObject 和 SelectObject 呼叫来选择等宽字体。

在 WM\_SIZE 讯息处理期间, TYPED 计算视窗的字元宽度和高度并把值保存在 cxBuffer 和 cyBuffer 变数中, 然後使用 malloc 分配缓冲区以保存在视窗内输入的所有字元。注意, 缓冲区的位元组大小取决於 cxBuffer、cyBuffer 和 sizeof (TCHAR), 它可以是 1 或 2, 这依赖於程式是以 8 位元的字元处理还是以 Unicode 方式编译的。

xCaret 和 yCaret 变数保存插入符号位置。在 WM\_SETFOCUS 讯息处理期间, TYPED 呼叫 CreateCaret 来建立与字元有相同宽度和高度的插入符号, 呼叫 SetCaretPos 来设定插入符号的位置, 呼叫 ShowCaret 使插入符号可见。在 WM\_KILLFOCUS 讯息处理期间, TYPED 呼叫 HideCaret 和 DestroyCaret。

对 WM\_KEYDOWN 的处理大多要涉及游标移动键。Home 和 End 把插入符号送至一行的开始和末尾处, Page Up 和 Page Down 把插入符号送至视窗的顶端和底部, 箭头的用法不变。对 Delete 键, TYPED 将缓冲区中从插入符号之後的那个位置开始到行尾的所有内容向前移动, 并在行尾显示空格。

WM\_CHAR 处理 Backspace、Tab、Linefeed (Ctrl-Enter)、Enter、Escape 和字元键。注意, 在处理 WM\_CHAR 讯息时 (假设使用者输入的每个字元都非常重要), 我使用了 lParam 中的重复计数; 而在处理 WM\_KEYDOWN 讯息时却不这么作 (避免有害的重复卷动)。对 Backspace 和 Tab 的处理由於使用了 SendMessage 函式而得到简化, Backspace 与 Delete 做法相仿, 而 Tab 则如同输入了若干个空格。

前面我已经提到过, 在非 WM\_PAINT 讯息处理期间, 如果要在视窗中绘制内容, 则应该隐蔽游标。TYPED 为 Delete 键处理 WM\_KEYDOWN 讯息和为字元键处理 WM\_CHAR 讯息时即是如此。在这两种情况下, TYPED 改变缓冲区中的内容, 然後在视窗中绘制一个或者多个新字元。

虽然 TYPED 使用了与 KEYVIEW2 相同的做法以在字元集之间切换 (就像使用者切换键盘布局一样), 但對於远东版的 Windows, 它还是不能正常工作。TYPED 不允许使用两倍宽度的字元。此问题将在第十七章讨论, 那时我们将详细讨论字体与文字输出。