附录 A

控制台程序

在 Windows 操作系统中运行一个应用程序后，我们经常会看到两种界面，一种是标准的窗口界面，窗口界面的程序架构在第 4 章中已经有了详细的介绍；另一种是类似于 MS-DOS 程序的文本界面，如常用的 Ping、Xcopy 等命令使用的都是这种界面，这种界面就叫做控制台（Console），由于控制台在 Windows 系统中还是以一个文本窗口的方式出现的，所以一般将这个窗口称为控制台窗口。

从表面看，32 位的控制台程序和 16 位的 MS-DOS 应用程序在外观和表现上都是很相似的，比如它们都是在一个黑洞洞的文本窗口中显示文本，都支持命令行下的重定向操作，读取键盘的方式也是一样的。但是，在这个表象下面，两者却是完全不同的，DOS 应用程序是 16 位的实模式程序，而 Windows 下的控制台程序却是不折不扣的 32 位保护模式程序，它可以使用 Win32 API 函数，文件头中同样有导入表和导出表，可以在程序中建立多个线程执行。总之，控制台程序是长着“ DOS 程序面孔 ”的 Win32 程序，可以使用 Win32 编程中的所有特征。

进一步来说，如果一定要让控制台程序有一个窗口的话，也可以在其中使用 CreateWindow 函数来创建一个窗口，这样控制台程序可以在使用终端界面输入输出的同时使用窗口上的菜单来操作（但估计没有人会做这样的事情）。

控制台程序最主要的用途是用于网络的远程维护。进行远程维护时一般使用 Telnet 等工具登录到远程主机并在上面执行命令，如果执行的是图形界面的程序，这个界面是无法远程操作的，所以我们可以发现 Windows 中用于网络的命令大多数是控制台界面的，如 Ping，Netstat，Tracert，Arp，Route，Ipconfig 和 Finger 等，与此相比，很难想像类似于 Office 这样的软件会用在远程操作中。

作为对第 4 章中窗口模式的补充，本节中将简单介绍控制台程序和窗口程序的区别，以及控制台程序的写法。

A.1 控制台程序和窗口程序的区别

除了和界面相关的代码有所不同外，控制台程序和窗口程序的区别还在于链接的时候指定参数的不同，读者一定还记得 LINK 程序有个 subsystem 参数，当这个参数指定为 Windows 的时候，链接器生成的是窗口程序，本书中绝大部分以窗口为界面的例子程序中，LINK 语句是这样写的：

```
Link /subsystem:windows Test.obj Test.res
```

将 subsystem 参数改为 console 的时候，LINK 程序产生的就是控制台文件：

```
Link /subsystem:console Test.obj Test.res
```

两种参数生成的可执行文件的不同表现在文件头中，可执行文件（PE 文件）的文件头中有一个 IMAGE_OPTIONAL_HEADER32 结构，结构中的 Subsystem 字段就记录了文件类型的不同，读者可以在第 17 章的 17.1.3 节中看到对文件头的详细分析。

运行文件时，操作系统会检查文件头中的 Subsystem 参数，如果发现参数的类型是窗口文件，那么将文件以正常的方式运行；如果发现参数的类型是控制台文件，那么操作系统将为程序创建一个控制台窗口（即类似于 DOS 窗口的这个文本窗口），然后运行文件。

另外，当一个控制台程序是被另一个控制台程序作为子进程运行的时候，系统不会为它创建新的控制台窗口，而是将父进程的窗口指定给它，所以在“我的电脑”中双击运行一个控制台程序的时候，会出现一个新的控制台窗口，而在“命令提示符”窗口中用命令行参数运行一个控制台程序的时候，程序会直接使用“命令提示符”的窗口。

我们可以用几个简单的实验来验证这一点。首先打开“命令提示符”，进入第 4 章例子程序的目录 Chapter04\FirstWindow（这是一个普通的窗口程序而不是控制台程序），在命令行下输入 FirstWindow 来运行程序，程序运行后窗口出现了，但是不必等到窗口关闭，“命令提示符”就会直接回到等待输入命令的状态，也就是说，普通的窗口程序并不会占用父进程的控制台窗口。

现在修改 Chapter04\FirstWindow 目录中的 Makefile 文件，将 LINK 命令的参数改成 /subsystem:console，然后用 nmake /a 重新编译，这样程序的代码没有任何变化，仅仅是它的文件类型变成了控制台程序而已。

重复上面的步骤，在命令行下运行 FirstWindow 程序，可以看到，程序运行后窗口出现了，但是“命令提示符”处于等待状态，只有关闭窗口 FirstWindow 程序，“命令提示符”中才会回到等待输入的状态，这说明控制台程序的父进程如果也是控制台程序的时候，程序将继承父进程的控制台窗口。

现在在“程序管理器”中通过双击 FirstWindow.exe 文件来运行，一个正常窗口出现的同时也出现了一个新的文本窗口，但是源代码中并没有创建过这个窗口呀？原来这个窗口就是操作系统自动“搭配”给程序的控制台窗口，关闭窗口退出程序后，控制台窗口也同时消失。这说明了当父进程不是控制台程序的时候，操作系统会自动为控制台程序创建一个控制台窗口。

所以，除了操作系统会在上述方面对控制台窗口的创建或继承进行一些准备工作外，控制台程序和窗口程序在其他方面并没有什么不同，控制台程序中仍然可以有消息循环，可以创建窗口，也可以做窗口程序能做的任何事情。

A.2 书写控制台程序

现在用一个例子来说明如何在控制台程序中进行输入及输出，例子程序的源代码位于 Appendix A\EchoLine 目录中，程序运行后，会等待用户输入一些字符，当用户按下回车键后，

程序读入输入的行并以蓝色将输入的内容重新显示出来，然后等待用户输入新的一行并循环往复。如果用户按下 Ctrl＋C 组合键，程序将退出执行。

程序的界面如图 A.1 所示，读者可以注意到，例子程序的控制台窗口的标题是“ EchoLine 例子 ”。如果在 “ 命令行提示符 ” 中运行这个程序，窗口的标题也会从 “ 命令提示符 ” 变成 “ EchoLine 例子 ”，等程序退出以后，标题会恢复到 “ 命令提示符 ”。

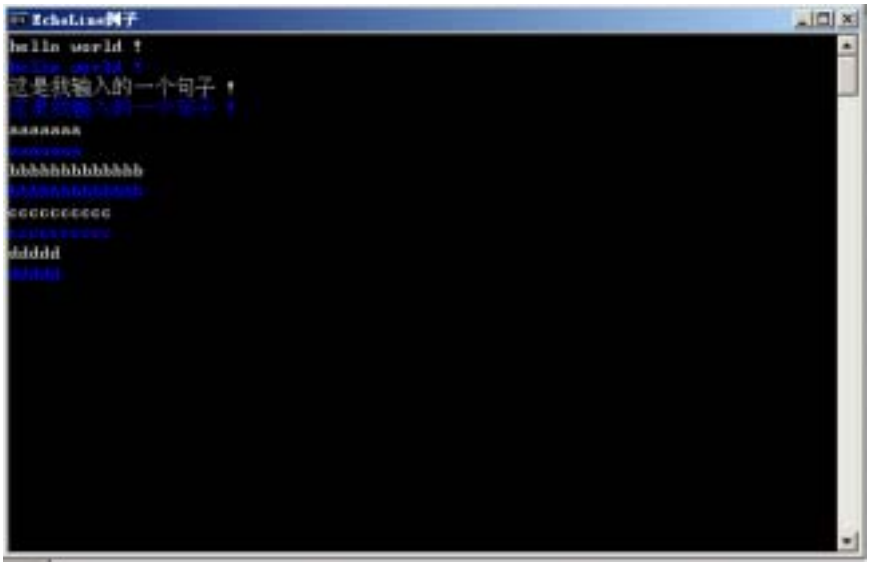

图 A.1 EchoLine 例子程序的界面

例子程序的源代码如下：

```
                .386
                .model flat, stdcall
                option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include         windows.inc
include         kernel32.inc
includelib      kernel32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                .data?
hStdIn          dd      ?               ;控制台输入句柄
hStdOut         dd      ?               ;控制台输出句柄
szBuffer        db      1024 dup (?)
dwBytesRead     dd      ?
dwBytesWrite    dd      ?
                .const
szTitle         db      'EchoLine 例子',0
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 代码段
```

```
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                .code
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 控制台 Ctrl+C 捕获例程
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_CtrlHandler    proc    _dwCtrlType

                pushad
                mov     eax,_dwCtrlType
                .if     eax ==  CTRL_C_EVENT || eax == CTRL_BREAK_EVENT
                        invoke  CloseHandle,hStdIn
                .endif
                popad
                mov     eax,TRUE
                ret

_CtrlHandler    endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
start:
;*********************************************************************
; 获取控制台句柄、设置句柄属性
;*********************************************************************
                invoke  GetStdHandle,STD_INPUT_HANDLE
                mov     hStdIn,eax
                invoke  GetStdHandle,STD_OUTPUT_HANDLE
                mov     hStdOut,eax
                invoke  SetConsoleMode,hStdIn,ENABLE_LINE_INPUT or \
                        ENABLE_ECHO_INPUT or ENABLE_PROCESSED_INPUT
                invoke  SetConsoleCtrlHandler,addr _CtrlHandler,TRUE

                invoke  SetConsoleTitle,addr szTitle
;*********************************************************************
; 循环读取控制台输入并显示
;*********************************************************************
                .while  TRUE
                        invoke SetConsoleTextAttribute,hStdOut,FOREGROUND_RED\
                                or FOREGROUND_GREEN or FOREGROUND_BLUE
                        invoke  ReadConsole,hStdIn,addr szBuffer,\
                                sizeof szBuffer,addr dwBytesRead,NULL
                        .break  .if ! eax

                        invoke  SetConsoleTextAttribute,hStdOut,\
                                FOREGROUND_BLUE or FOREGROUND_INTENSITY
                        invoke  WriteConsole,hStdOut,addr szBuffer,\
                                dwBytesRead,addr dwBytesWrite,NULL
                .endw
                invoke  ExitProcess,NULL
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                end     start
```

控制台程序输入输出的表现和 DOS 程序类似，它可以从标准输入设备进行输入，也可

以输出到标准输出设备或者标准错误输出设备，所以，控制台程序中首先要做的事情是获取这些设备的句柄。

1. 控制台句柄的获取和设置

控制台的各种句柄可以用 GetStdHandle 函数来获取，函数的用法如下：

```
invoke  GetStdHandle,nStdHandle
.if     eax != INVALID_HANDLE_VALUE
        mov     hStd,eax
.endif
```

将 nStdHandle 参数指定为以下不同的取值即可获取不同类型的句柄。

- STD_INPUT_HANDLE——标准输入句柄。
- STD_OUTPUT_HANDLE——标准输出句柄。
- STD_ERROR_HANDLE——标准出错信息句柄。

函数执行成功将返回对应的句柄，否则将返回 INVALID_HANDLE_VALUE 值。在例子程序中，程序两次调用 GetStdHandle 函数来分别获取输入和输出句柄，并保存到 hStdIn 和 hStdOut 变量中以便在以后用来读取键盘输入，以及进行屏幕输出，对应的代码如下：

```
    invoke  GetStdHandle,STD_INPUT_HANDLE
    mov     hStdIn,eax
    invoke  GetStdHandle,STD_OUTPUT_HANDLE
    mov     hStdOut,eax
    ...
```

获取句柄后就可以用 ReadConsole 函数来读取键盘输入了，但在此之前，可以用 SetConsoleMode 函数对输入句柄的工作模式进行设置，这样能让 ReadConsole 函数以我们所期望的方式工作，如一次读取一行还是读取一个字符，是否对 Ctrl＋C 组合键进行拦截等。

SetConsoleMode 函数的用法如下：

```
invoke  SetConsoleMode,hConsoleHandle,dwMode
```

hConsoleHandle 参数是要设置的控制台句柄，dwMode 是工作模式，当控制台句柄是输入句柄时，dwMode 可以指定为以下标志位的组合。

- ENABLE_LINE_INPUT——行模式标志，指定该标志位后 ReadConsole 函数将在用户输入回车后才返回，否则用户输入任何字符后即返回。
- ENABLE_ECHO_INPUT——指定该标志后，用户输入的时候字符将在屏幕上回显。
- ENABLE_PROCESSED_INPUT——指定该标志后，系统将拦截 Ctrl＋C 组合键，如果不指定该标志的话，系统将 Ctrl＋C 组合键的键值（03h）作为字符返回给 ReadConsole 函数。
- ENABLE_WINDOW_INPUT 和 ENABLE_MOUSE_INPUT——当用户对控制台窗口的大小进行改变，或者按动鼠标时，系统将记录这些消息并允许程序用 ReadConsoleInput 函数读取（ReadConsole 函数会忽略这种类型的消息）。

例子程序中对输入句柄的模式进行设置时指定了三个标志位：ENABLE_LINE_INPUT，ENABLE_ECHO_INPUT 和 ENABLE_PROCESSED_INPUT，表示程序在后面将以行模式读

取键盘输入，用户输入的同时需要将字符回显并且程序将过滤 Ctrl＋C 组合键，对应的代码如下：

```
        invoke  SetConsoleMode,hStdIn,ENABLE_LINE_INPUT or \
                ENABLE_ECHO_INPUT or ENABLE_PROCESSED_INPUT
```

例子程序中还用到 SetConsoleTitle 函数将控制台窗口的标题设置为“ EchoLine 例子 ”，SetConsoleTitle 函数只有一个参数——指向标题字符串的指针：

```
        invoke  SetConsoleTitle,addr szTitle
```

2. 截获 Ctrl＋Break

在控制台程序中往往需要截获 Ctrl＋C（或 Ctrl＋Break）的组合键来判断是否要中途退出，这在 DOS 时代的程序中靠截获 Int 23h 中断来实现，但在 Win32 中不能再使用这种方法。

Win32 控制台程序使用 SetConsoleCtrlHandler 函数来将 Ctrl＋C 的处理代码重新定义到自己指定的子程序中，这样当输入句柄具有 ENABLE_PROCESSED_INPUT 属性时，用户按下 Ctrl＋C 组合键后系统即调用指定的子程序：

```
    invoke  SetConsoleCtrlHandler,HandlerRoutine,Add
```

当 HandlerRoutine 参数指定为处理 Ctrl+C 按键的子程序地址，Add 参数指定为 TRUE 的时候，系统将设置这个子程序为 Ctrl＋C 处理程序，如果 Add 参数指定为 FALSE，系统将取消这个设置。

HandlerRoutine 参数也可以为 NULL，这时当 Add 参数设置为 TRUE 时，系统将忽略对 Ctrl＋C 的处理，设置为 FALSE 的话系统将恢复原来的处理方式。

例子程序中使用 SetConsoleCtrlHandler 函数将处理子程序设置到_CtrlHandler 中。

Ctrl＋C 的处理子程序必须按照规定的格式定义：

```
    HandlerRoutine  proc    dwCtrlType
```

该子程序有一个输入参数 dwCtrlType，系统调用该子程序的时候将使用这个参数指明发生事件的类型，事件类型可能是以下几种。

- CTRL_C_EVENT——收到 Ctrl＋C 字符。
- CTRL_BREAK_EVENT——收到 Ctrl＋Break 字符。
- CTRL_CLOSE_EVENT——用户关闭了控制台窗口（比如按下了控制台窗口上面的关闭按钮或在控制台窗口的菜单上选择了“ 关闭 ”等）。
- CTRL_LOGOFF_EVENT——当前用户注销。
- CTRL_SHUTDOWN_EVENT——系统准备关闭。

在例子代码中，_CtrlHandler 子程序仅处理 CTRL_BREAK_EVENT 事件以及 CTRL_C_EVENT 事件，并在检测到这两个事件的时候用 CloseHandle 函数将输入句柄关闭，这样在后面的循环中 ReadConsole 函数就会出错返回，程序即可退出循环。

3. 控制台窗口的输入和输出

从控制台窗口接收键盘输入可以用 ReadConsole 或 ReadFile 两种函数来完成（在第 10

章中有对 ReadFile 函数的详细介绍），ReadConsole 函数的使用方法如下：

```
invoke  ReadConsole,hConsoleInput,lpBuffer,\
        nNumberOfCharsToRead,lpNumberOfCharsRead,lpReserved
```

hConsoleInput 参数为控制台的标准输入句柄，lpBuffer 指向用来接收输入数据的缓冲区，nNumberOfCharsToRead 参数指定要读取的数据长度，lpNumberOfCharsRead 指向一个双字，函数在这里返回实际读取的字节数。如果函数读取输入成功，则返回非零值，如果读取失败，则返回零。

ReadConsole 和 ReadFile 函数的区别在于能否支持重定向，ReadFile 函数允许使用“＜”符号将一个文本文件的内容作为输入重定向到输入句柄中，而 ReadConsole 函数不支持重定向。

两个函数读取输入字符的方式取决于 SetConsoleMode 函数对输入句柄工作模式的设置，当工作模式指定为行读取模式的时候，只有用户输入的字符数等于 nNumberOfCharsToRead 参数指定的数量时，或者用户输入回车后函数才返回；当工作模式设置为非行读取模式时，即使 nNumberOfCharsToRead 参数指定的值很大，只要有任何字符输入，函数即返回，函数实际读取的字符数量放在 lpNumberOfCharsRead 参数指向的双字中。

如果需要向控制台窗口输出文本，同样可以使用 WriteConsole 函数或者 WriteFile 函数（在第 10 章中有 WriteFile 函数的详细介绍），WriteConsole 函数的用法如下：

```
invoke  WriteConsole,hConsoleOutput,lpBuffer,\
        nNumberOfCharsToWrite,lpNumberOfCharsWritten,lpReserved
```

参数 hConsoleOutput 指定为前面获取的标准输出句柄，lpBuffer 参数指向输出的内容，nNumberOfCharsToWrite 参数为要输出的数据长度，lpNumberOfCharsWritten 指向一个双字变量，用来返回实际输出的数据长度。

同样，两种方法的区别在于 WriteConsole 函数不支持输出内容的重定向，而 WriteFile 函数支持重定向，这样使用命令行方式的管道操作符可以将内容重定向到一个文件中，如下面的命令将 Ping 的结果存放到 result.txt 文件中：

```
ping www.yahoo.com > result.txt
```

如果程序希望某些内容可以重定向，而某些内容不允许重定向，那么可以混合使用这两个函数来输出文本。

例子程序中循环使用 ReadConsole 函数来读入用户的输入，并用 WriteConsole 函数来将输入的字符串回显在控制台窗口中。当用户输入 Ctrl＋C 组合键的时候，Ctrl＋C 的处理子程序中将输入句柄关闭，这样 ReadConsole 会返回失败，循环退出。

4. 设置控制台窗口文本的颜色

例子程序在用户输入的时候显示的字符颜色是白色的，但是在将同样的内容输出时，字符颜色却是加亮的蓝色，这是因为在调用 ReadConsole 和 WriteConsole 函数前分别用 SetConsoleTextAttribute 函数设置了不同的文本颜色。

SetConsoleTextAttribute 函数用于设置控制台窗口中将要显示的文本颜色，一旦设置完毕后，以后显示的字符将全部使用新的颜色，但是最初显示的字符颜色不受影响，函数的用法

如下：

```
invoke  SetConsoleTextAttribute,hConsoleHandle,dwColor
```

hConsoleHandle 参数是需要设置的输出句柄，dwColor 参数指定颜色值，颜色值可以是下列取值的组合。

- FOREGROUND_BLUE，FOREGROUND_GREEN，FOREGROUND_RED——分别表示字符颜色为蓝色、绿色和红色。
- FOREGROUND_INTENSITY——字符颜色加亮。
- BACKGROUND_BLUE，BACKGROUND_GREEN，BACKGROUND_RED——分别表示字符的背景色为蓝色、绿色和红色。
- BACKGROUND_INTENSITY——字符背景颜色加亮。

如果需要其他颜色，可以将这些颜色值按照三原色相加的方式组合起来，比如设置字符颜色为加亮的白色时，因为白色是由红绿蓝三色组合而成的，所以 dwColor 参数可以指定为 FOREGROUND_INTENSITY or FOREGROUND_RED or FOREGROUND_GREEN or FOREGROUND_BLUE；需要红色加亮的字符，但是背景是黄色的时候，可以指定为 FOREGROUND_RED or FOREGROUND_INTENSITY or BACKGROUND_GREEN or BACKGROUND_RED，这是因为背景的黄色是由红色和绿色组合而成的。

将控制台程序和普通的窗口程序对比就可以发现，由于不必处理复杂的消息机制，控制台程序的流程可以使用和 DOS 程序类似的架构，也就是说程序可以按顺序化的方式来写，而不是以消息驱动的方式来写。这种架构可以让程序的编写更加简单。

附录 B

窗口消息实验

在本节中，将通过不同的实验来了解常见的窗口消息，并进一步理解窗口的工作机制。我们将构造一个程序，在程序中将收到的窗口消息查表翻译成文本以“ WM_XXX ”格式显示出来，同时将和窗口相关的 API 函数的调用也显示出来，这样可以分析窗口的各种行为和消息之间的关系。

实验用到的源代码请参考附书光盘的 Appendix B 目录。

B.1 MsgWindow 程序

为了把窗口消息翻译成文本信息显示出来，我们可以选择在窗口的客户区中显示文本，但这样会引入新的消息，干扰实验，所以，这里选择了一种新的方法，就是先打开 Windows 附件中自带的 Notepad 记事本程序，然后在程序中将要显示的内容通过 SendMessage 发送到记事本中，这样可以通过查看记事本中的内容来了解 MsgWindow 的运行情况。

程序以 Chapter04\FirstWindow 例子为模板，在此基础上增加了一些功能。增加的内容共有 3 个部分。

第一部分是将消息查表转换为字符串，首先在 .const 段中增加两个表：十六进制的消息编号列表 dwMsgTable 和字符串列表 szStringTable，两表中的项目一一对应，代码如下：

```
                .const
dwMsgTable      dd      WM_NULL
                dd      WM_CREATE
                dd      WM_DESTROY
                dd      WM_MOVE
                …
                dd      WM_EXITSIZEMOVE
MSG_TABLE_LEN   equ     ($ - dwMsgTable)/sizeof dword

MSG_STRING_LEN  equ     sizeof szStringTable
szStringTable   db      'WM_NULL               ',0
                        db'WM_CREATE           ',0
                db      'WM_DESTROY            ',0
                db      'WM_MOVE               ',0
                …
                db      'WM_EXITSIZEMOVE       ',0
szFormat        db      'WndProc: [%04x]%s %08x %08x',0dh,0
```

MSG_TABLE_LEN 定义了表的项数，MSG_STRING_LEN 定义了字符串表中每一项的长度。sizeof 操作符取的是 szStringTable 这一行中的数据长度，而非包括下面全部的字符串行。为了简化处理，全部字符串的长度保持相等。由于篇幅所限，这里没有列出全部的消息列表，完整的源代码可以在本书附带光盘的 Appendix B\MsgWindow01 目录中找到。

程序在窗口过程的入口处调用_ShowMessage 子程序来翻译消息并传给记事本：

```
_ProcWinMain        proc    uses ebx edi esi,hWnd,uMsg,wParam,lParam

                    invoke  _ShowMessage,uMsg,wParam,lParam
                    mov     eax,uMsg
                    .if     eax ==  WM_XXX
                    …
```

_ShowMessage 子程序用来将消息查表翻译成字符串，源程序如下：

```
_ShowMessage        proc    _uMsg,_wParam,_lParam
                    local   @szBuffer[128]:byte

                    pushad
;*********************************************************************
; 查找消息的说明字符串
;*********************************************************************
                    mov     eax,_uMsg
                    mov     edi,offset dwMsgTable
                    mov     ecx,MSG_TABLE_LEN
                    cld
                    repnz   scasd
                    .if     ZERO?
                            sub     edi,offset dwMsgTable + sizeof dword
                            shr     edi,2
                            mov     eax,edi
                            mov     ecx,MSG_STRING_LEN
                            mul     ecx
                            add     eax,offset szStringTable
;*********************************************************************
; 翻译格式并发送到 Notepad 窗口
;*********************************************************************
                            invoke  wsprintf,addr @szBuffer,addr szFormat,\
                                    _uMsg,eax,_wParam,_lParam
                            invoke _SendtoNotepad,addr @szBuffer
                    .endif
                    popad
                    ret


_ShowMessage        endp
```

在这里要用到 repnz scasd 指令，scasd 指令是把 eax 中的值从[edi]开始的内存中按双字比较，同时将 edi 加 4，如果相等，则 ZR 标志置位，否则为 NZ，repnz 表示如果标志为 NZ，则以 ecx 为重复次数重复搜索，直到相等或 ecx 为零为止。

将 ecx 赋值为消息表的项数 MSG_TABLE_LEN，将 edi 赋值为消息表的开始地址 offset

dwMsgTable，然后开始查找，停止后可以查看标志 Zero 位，如果是非 ZERO，表示查完全部都没有找到，如果是 ZERO，则表示找到表项。

当标志为 ZERO 时，edi 指向找到项目的后一项，将 edi 减去一项的长度（sizeof dword）以及表的基址，再除以表项的长度（sizeof dword 等于 4，除以 4 等于右移两位，所以程序中用 shr edi,2），就是消息在表中的索引了，接下来算出消息字符串的位置，位置等于：索引×字符串长＋字符串表基址，代码如下：

```
    mov     ecx,MSG_STRING_LEN
    mul     ecx
    add     eax,offset szStringTable
```

这样，eax 中就是字符串的地址了。最后将消息编号、名称和参数用 wsprintf 函数格式化成可以发送的字符串存放到@szBuffer 中，并用_SendtoNotepad 子程序将@szBuffer 中的内容发送到记事本中。

程序增加的第二部分就是下面这个_SendtoNotepad 子程序：

```
    szDestClass     db        'Notepad',0
    _SendtoNotepad  proc      _lpsz
                    local     @hWinNotepad

                    pushad
                    invoke    FindWindow,addr szDestClass,NULL
                    .if       eax
                              mov      ecx,eax
                              invoke   ChildWindowFromPoint,ecx,20,20
                    .endif
                    .if       eax
                              mov      @hWinNotepad,eax
                              mov      esi,_lpsz
                              @@:
                              lodsb
                              or       al,al
                              jz        @F
                              movzx    eax,al
                              invoke   PostMessage,@hWinNotepad,WM_CHAR,eax,1
                              jmp      @B
                              @@:
                    .endif
                    popad
                    ret

    _SendtoNotepad  endp
```

该子程序中首先用 FindWindow 函数来查找记事本程序是否已经运行，记事本程序的窗口类名称为“ Notepad ”，FindWindow 可以用窗口类当做第一个参数来查找，如果找到，返回的是记事本程序的主窗口句柄，否则返回 0。

要发送的是模拟键盘按键的消息 WM_CHAR，这样就好像在记事本中人工键入字符，但直接向记事本主窗口发送 WM_CHAR 消息是不行的，要向记事本窗口客户区中的编辑子窗

口发送消息才行，所以程序中又用从位置获取子窗口句柄的函数 ChildWindowFromPoint 来获得编辑子窗口的句柄。

锁定了最后的目标即记事本中的编辑子窗口后，程序用 PostMessage 向它发送消息，根据字符串的长度，用一个循环每次发送一个 WM_CHAR 消息，WM_CHAR 消息的 wParam 和 lParam 的含义如下：

```
wParam = chCharCode    // wParam 是键值
lParam = lKeyData      // lParam 是键数据（重复次数）
```

程序中用 mov eax,al 将键值扩展到参数所需的 32 位，当做 wParam 参数发送，lParam 为 1，表示键的重复次数为 1 次，这样一来，记事本中就源源不断地显示出 MsgWindow 程序的运行轨迹了。

MsgWindow 程序增加的第三部分是在每个函数的前后增加了显示状态的语句，它们只是简单地把一个字符串发送到记事本中。

```
;定义一些字符串
szCreateWindow1    db       'Creating Window...',0dh,0
szCreateWindow2    db       'CreateWindow end',0dh,0
szShowWindow1      db       'Showing Window...',0dh,0
szShowWindow2      db       'ShowWindow end',0dh,0
szUpdateWindow1    db       'Updating Window...',0dh,0
szUpdateWindow2    db       'UpdateWindow end',0dh,0
szGetMsg1          db       'Getting Message...',0dh,0
szGetMsg2          db       '[%04x]Message gotten',0dh,0
szDispatchMsg1     db       'Dispatching Message...',0dh,0
szDispatchMsg2     db       'DispatchMessage end',0dh,0
                            …
                   invoke   _SendtoNotepad,addr szCreateWindow1
                   invoke   CreateWindowEx,…
                   invoke   _SendtoNotepad,addr szCreateWindow2
                   invoke   _SendtoNotepad,addr szShowWindow1
                   invoke   ShowWindow,hWinMain,SW_SHOWNORMAL
                   invoke   _SendtoNotepad,addr szShowWindow2
                   invoke   _SendtoNotepad,addr szUpdateWindow1
                   invoke   UpdateWindow,hWinMain
                   invoke   _SendtoNotepad,addr szUpdateWindow2
                   …
```

上面代码中的粗体部分就是相对于 FirstWindow 程序增加的内容，好了，现在 DOS 控制台上键入 nmake 将 MsgWindow 程序编译出来，然后打开记事本，再运行 MsgWindow.exe，如果记事本上出现一大堆的东西，就说明实验可以开始了！

B.2　开始实验

实验 1. 验证收到消息的顺序

打开记事本，然后运行 MsgWindow 程序，记事本上出现的内容如下：

```
Creating Window...
WndProc: [0024]WM_GETMINMAXINFO       00000000 0012fda4
WndProc: [0081]WM_NCCREATE            00000000 0012fd8c
WndProc: [0083]WM_NCCALCSIZE          00000000 0012fdc4
WndProc: [0001]WM_CREATE              00000000 0012fd68
CreateWindow end
Showing Window...
WndProc: [0018]WM_SHOWWINDOW          00000001 00000000
WndProc: [0046]WM_WINDOWPOSCHANGING   00000000 0012fec0
WndProc: [0046]WM_WINDOWPOSCHANGING   00000000 0012fec0
WndProc: [001c]WM_ACTIVATEAPP         00000001 00000450
WndProc: [0086]WM_NCACTIVATE          00000001 00000000
WndProc: [000d]WM_GETTEXT             000001fe 0012f52c
WndProc: [0006]WM_ACTIVATE            00000001 00000000
WndProc: [0007]WM_SETFOCUS            00000000 00000000
WndProc: [0085]WM_NCPAINT             00000001 00000000
WndProc: [000d]WM_GETTEXT             000001fe 0012f52c
WndProc: [0014]WM_ERASEBKGND          e3010449 00000000
WndProc: [0047]WM_WINDOWPOSCHANGED    00000000 0012fec0
WndProc: [0005]WM_SIZE                00000000 00450064
WndProc: [0003]WM_MOVE                00000000 004b0038
ShowWindow end
Updating Window...
WndProc: [000f]WM_PAINT               00000000 00000000
UpdateWindow end
Getting Message...
```

以 WndProc 带头的是在窗口过程中收到的消息，显然，和 4.2.4 节中讲述的是一致的，在调用 CreateWindowEx 时，窗口过程就开始接收消息，里面有重要的 WM_CREATE，然后在 ShowWindow 的时候，Windows 向窗口过程发送了很多的消息，而 UpdateWindow 只给窗口过程发送了一条 WM_PAINT 消息，接下来就进入了消息循环。

可以看到，GetMessage 函数是程序主动上交空闲时间的办法之一，因为显示出 Getting Message.…以后，程序就等着那里了，这表示程序的空闲时间并不浪费在消息循环中，而是在 GetMessage 函数内部由 Windows 自己分配了。

接下来把鼠标移过 MsgWindow 窗口，在记事本上看到了什么？用户一个小小的动作就够窗口过程忙的——我们看到了多次重复的下列内容：

```
WndProc: [0084]WM_NCHITTEST           00000000 00830096
WndProc: [0020]WM_SETCURSOR           001b0304 02000001
[0200]Message gotten
Dispatching Message...
WndProc: [0200]WM_MOUSEMOVE           00000000 0038005e
```

```
    DispatchMessage end
    Getting Message...
```

首先，Windows 在 GetMessage 没有返回的时候就调用了两次窗口过程，分别是处理 WM_NCHITTEST 和 WM_SETCURSOR，它们并不经过消息循环；然后，GetMessage 取到[0200]消息并返回，0200 是 WM_MOUSEMOVE 消息的编号；接下来，DispatchMessage 函数开始工作，在这个函数的内部，消息被 Windows 发送给窗口过程处理，最后 DispatchMessage 返回，然后开始新的 GetMessage。

最后在 MsgWindow 上单击“关闭”按钮，看发生了什么：

```
    [00a1]Message gotten
    Dispatching Message...
    WndProc: [00a1]WM_NCLBUTTONDOWN          00000014 003d0097
    WndProc: [0215]WM_CAPTURECHANGED         00000000 00000000
    WndProc: [0112]WM_SYSCOMMAND             0000f060 003d0097
    WndProc: [0010]WM_CLOSE                  00000000 00000000
    WndProc: [0046]WM_WINDOWPOSCHANGING      00000000 0012fad8
    WndProc: [0047]WM_WINDOWPOSCHANGED       00000000 0012fad8
    WndProc: [0086]WM_NCACTIVATE             00000000 00000000
    WndProc: [0006]WM_ACTIVATE               00000000 00000000
    WndProc: [001c]WM_ACTIVATEAPP            00000000 00000450
    WndProc: [0008]WM_KILLFOCUS              00000000 00000000
    WndProc: [0002]WM_DESTROY                00000000 00000000
    WndProc: [0082]WM_NCDESTROY              00000000 00000000
    DispatchMessage end
    Getting Message...
    [0012]Message gotten
```

GetMessage 收到的是按下鼠标的 WM_NCLBUTTONDOWN 的消息，由 DispatchMessage 转给窗口过程处理后，窗口过程将它转手给了 DefWindowProc，DefWindowProc 根据鼠标的位置得出结论：用户按的是“关闭”按钮，放开鼠标后，它就给窗口过程发送 WM_CLOSE 消息，当窗口过程调用 DestroyWindow 后，窗口被摧毁，窗口过程最后收到的是 WM_DESTROY 消息和 WM_NCDESTROY 消息，而消息循环中 GetMessage 最后收到的是 0012 号 WM_QUIT 消息，消息循环结束。

实验 2. 全部消息都经过消息循环吗

在做这个实验之前，读者已经知道并不是所有的消息都是经过消息循环的，它们中有些是 Windows 直接发送到窗口过程的，上一个实验中就已经可以看到 GetMessage 返回的次数明显比调用窗口过程的次数少，这意味着窗口过程有很多次是由 Windows 直接调用的。

这次，我们用极端的方式来验证，先把消息循环中的 DispatchMessage 去掉，这样 GetMessage 得到的消息将不会再被送到窗口过程了，窗口过程收到的就是由 Windows 直接调用的。改变后的源代码见所附光盘中的 Appendix B\MsgWindow02 目录。

编译后，同样先打开记事本，再执行 MsgWindow，然后将鼠标移过 MsgWindow 窗口，并尝试着单击“关闭”按钮和双击等各种动作，结果是窗口过程还是在被调用，如下所示：

```
WndProc: [0084]WM_NCHITTEST          00000000 007e0088
WndProc: [0020]WM_SETCURSOR          0026030c 02000001
WndProc: [0084]WM_NCHITTEST          00000000 006c0070
WndProc: [0020]WM_SETCURSOR          0026030c 02000001
…
```

由于没有了 DispatchMessage，大部分消息被忽略了，窗口就停在了屏幕上，不能进行移动、缩放或关闭等操作，但还是有一部分消息直接由 Windows 发送给窗口过程，它们是鼠标位置测试的 WM_NCHITTEST 消息和要求设置光标的 WM_SETCURSOR 消息，所以在鼠标移动到边框的时候，鼠标光标还是会变成双箭头的样子。

另外，尝试着单击其他窗口来切换焦点，然后再单击标题栏来重新激活窗口，可以发现 WM_MOUSEACTIVATE，WM_ACTIVATE 和 WM_KILLFOCUS 等消息也是不经过消息循环的。接下来，把一个窗口移动到 MsgWindow 窗口前覆盖它的位置，再移开，可以发现 WM_SYNCPAINT 和 WM_ERASEBKGND 等消息也是由 Windows 直接发给窗口过程的。

最后，关闭窗口，当然这个窗口只能用 Ctrl+Alt+Del 组合键在任务管理器中关闭了！

实验 3. TranslateMessage 有什么用

首先执行实验 1 的 MsgWindow，在窗口上敲几个键，每次敲一个键，得到的消息是：WM_KEYDOWN，WM_CHAR 和 WM_KEYUP。如果按下键盘不放，则首先得到一个 WM_KEYDOWN，接下来就是重复的 WM_CHAR 和 WM_KEYUP 消息，直到放开键盘为止，最后才会看到一个 WM_KEYUP。显示如下：

```
WndProc: [0100]WM_KEYDOWN            00000041 001e0001
WndProc: [0102]WM_CHAR               00000061 001e0001
WndProc: [0101]WM_KEYUP              00000041 c01e0001
```

在 WM_KEYDOWN 和 WM_KEYUP 消息中，wParam 中是按键的扫描码，上面的数据是按下了“ A ”键得到的，00000041h 是“ A ”的扫描码，到了 WM_CHAR 消息中，wParam 中就是已经转换过的 ASCII 码 61 了，代表输入的是小写的字母“ a ”。

好！现在从程序中去掉 TranslateMessage 语句（修改以后的源代码放在 Appendix B\MsgWindow03 目录中），然后看这个程序的运行结果，同样，按几次键以及按下键盘不放，我们发现：这中间的区别就是少了 WM_CHAR，所以只有在处理键盘输入要用到转换后的 ASCII 码的时候，TranslateMessage 函数才是有用的，在别的时候完全可以省略这个语句。这个函数的功能就是看到 WM_KEYDOWN 的时候把消息检查一下，然后根据键值将一条新的 WM_CHAR 或 WM_SYSCHAR 消息放入消息循环中。

实验 4. DefWindowProc 做了什么工作

现在把 DefWindowProc 语句去掉（源代码详见 Appendix B\MsgWindow04 目录），然后再以同样的方法运行，窗口根本就没有出现！看记事本中出现了什么：

```
Creating Window...
WndProc: [0024]WM_GETMINMAXINFO      00000000 0012fda4
WndProc: [0081]WM_NCCREATE           00000000 0012fd8c
```

```
WndProc: [0082]WM_NCDESTROY           00000000 00000000
CreateWindow end
Showing Window...
ShowWindow end
Updating Window...
UpdateWindow end
Getting Message...
```

原来在建立窗口的时候执行到 WM_NCCREATE 消息后窗口就摧毁掉了，看 WM_NCCREATE 的说明：The DefWindowProc function returns TRUE，原来需要返回 1 来表示执行成功，所以需要处理 WM_NCCREATE 并返回 1，现在在窗口过程中加上下列分支：

```
.elseif eax ==  WM_NCCREATE
        mov     eax,1
        ret
```

接着编译后执行，怎么编译不成功了？不能写 exe 文件?原来上次的程序还停留在消息循环中没有退出来，让我们在任务管理器中将它终止再编译，成功了！

好！现在继续执行，窗口成功建立了，但似乎陷入了死循环，因为记事本上不停地有消息冒出来，而且只是冒出 WM_PAINT 消息来，为什么呢？原来 WM_PAINT 消息是不能不处理的，也不能丢弃，只要 Windows 认为窗口的客户区需要绘画（或者说是无效的），它就会不停地向窗口发送 WM_PAINT 消息，一般 WM_PAINT 消息的处理中用 BeginPaint 和 EndPaint 会隐含地让客户区有效，如果不用 BeginPaint/EndPaint，程序必须显式地把客户区设置为有效，Windows 才不会再发送 WM_PAINT 消息。这个函数是 ValidateRect，现在在分支中再加上处理 WM_PAINT 的代码：

```
.elseif  eax ==  WM_PAINT
         invoke  ValidateRect,hWnd,NULL
```

再编译执行，现在程序可以正常执行下去了，记事本上出现的信息也显示程序停留在了 GetMessage 处，一切正常。但是，窗口在哪里呢，屏幕上什么都没有，隐身了？把鼠标移到窗口原来应该出现的地方，记事本中熟悉的 WM_NCHITTEST 和 WM_SETCURSOR 消息出现了，原来窗口还在那里，只不过没有了 DefWindowProc 的处理，窗口的绘画等所有工作都没有做，窗口的边框与客户区等所有东西连画都没有画上去，所以窗口是存在的，但我们看不到它！

是不是再加上 WM_NCPAINT 消息自己画边框呢，这就不是这个实验的内容了。我们已经知道，DefWindowProc 做的工作太多了，缺了它我们要补上的代码可不是一两个分支的问题，而是上百个分支了！在这个实验中，我们根本不可能把它补全。

附录 C

浏览目录对话框

C.1 浏览目录对话框简介

在众多由系统提供的对话框中，除了第 8 章中介绍的众多通用对话框外，还有一个很常用的浏览目录对话框，该对话框如图 C.1 所示，这个对话框虽然也是通用型的，但是它是由 Shell32.dll 提供的，而不是由 Comdlg32.dll 提供的，在实现的方法上也和上面介绍的通用对话框有很大的不同，本节以一个例子来演示它的使用。

图 C.1 浏览目录对话框

例子程序的源代码位于所附光盘的 Appendix C\BrowseFolder 目录中，目录中包含了 BrowseFolder.asm 文件和_BrowseFolder.asm 文件。BrowseFolder.asm 文件的内容很简单，如下所示：

```
                    .386
                    .model flat,stdcall
```

```
                option casemap:none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include         windows.inc
include         user32.inc
includelib      user32.lib
include         kernel32.inc
includelib      kernel32.lib
include         shell32.inc
includelib      shell32.lib
include         ole32.inc
includelib      ole32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                .data?
szPath          db      MAX_PATH dup (?)

                .data
szSelect        db      '您选择的目录',0
szNoSelect      db      '您按下了取消键',0
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 代码段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                .code

include         _BrowseFolder.asm
start:
                invoke  GetCurrentDirectory,sizeof szPath,addr szPath
                invoke  _BrowseFolder,NULL,addr szPath
                .if     eax
                        invoke  MessageBox,NULL,offset szPath,\
                        offset szSelect,MB_OK
                .else
                        invoke MessageBox,NULL,offset szNoSelect,NULL,MB_OK
                .endif
                invoke  ExitProcess,NULL
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                end     start
```

主文件中仅包含了几句调用和显示结果的代码，全部的功能集中在_BrowseFolder.asm 中，该文件用 include 语句包含进主文件中，这样安排代码的原因是目录浏览对话框的实现比较复杂，把功能模块写成一个单独的文件可以便于在其他文件中引用，读者也可以直接把这个源文件不加修改地用在其他地方。_BrowseFolder.asm 文件的内容如下：

```
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; IUnknown 接口定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
externdef               IID_IUnknown:IID
```

```
LPUNKNOWN               typedef DWORD
LPPUNKNOWN              typedef ptr LPUNKNOWN

IUnknown_QueryInterfaceProto   typedef proto :DWORD, :DWORD, :DWORD
IUnknown_AddRefProto           typedef proto :DWORD
IUnknown_ReleaseProto          typedef proto :DWORD
IUnknown_QueryInterface        typedef ptr IUnknown_QueryInterfaceProto
IUnknown_AddRef                typedef ptr IUnknown_AddRefProto
IUnknown_Release               typedef ptr IUnknown_ReleaseProto

IUnknown struct DWORD
        QueryInterface  IUnknown_QueryInterface?
        AddRef          IUnknown_AddRef        ?
        Release         IUnknown_Release       ?
IUnknown ends
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;IMalloc 接口定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
externdef               IID_IMalloc:IID
LPMALLOC                typedef DWORD
LPPMALLOC               typedef ptr LPMALLOC

IMalloc_AllocProto             typedef proto :DWORD, :DWORD
IMalloc_ReallocProto           typedef proto :DWORD, :DWORD,:DWORD
IMalloc_FreeProto              typedef proto :DWORD, :DWORD
IMalloc_GetSizeProto           typedef proto :DWORD, :DWORD
IMalloc_DidAllocProto          typedef proto :DWORD, :DWORD
IMalloc_HeapMinimizeProto      typedef proto :DWORD

IMalloc_Alloc                  typedef ptr IMalloc_AllocProto
IMalloc_Realloc                typedef ptr IMalloc_ReallocProto
IMalloc_Free                   typedef ptr IMalloc_FreeProto
IMalloc_GetSize                typedef ptr IMalloc_GetSizeProto
IMalloc_DidAlloc               typedef ptr IMalloc_DidAllocProto
IMalloc_HeapMinimize           typedef ptr IMalloc_HeapMinimizeProto

IMalloc struct DWORD
        QueryInterface  IUnknown_QueryInterface?
        AddRef          IUnknown_AddRef        ?
        Release         IUnknown_Release       ?
        Alloc           IMalloc_Alloc          ?
        Realloc         IMalloc_Realloc        ?
        Free            IMalloc_Free           ?
        GetSize         IMalloc_GetSize        ?
        DidAlloc        IMalloc_DidAlloc       ?
        HeapMinimize    IMalloc_HeapMinimize   ?
IMalloc ends
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                .data   ?
_BrowseFolderTmp        dd      ?
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
                .const
_szDirInfo      db      '请选择目录：',0

                .code
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 弹出选择目录的对话框
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_BrowseFolderCallBack   proc    hWnd,uMsg,lParam,lpData
                local   @szBuffer[260]:byte

                mov     eax,uMsg
                .if     eax ==  BFFM_INITIALIZED
                        invoke  SendMessage,hWnd,BFFM_SETSELECTION,\
                        TRUE,_BrowseFolderTmp
                .elseif eax ==  BFFM_SELCHANGED
                        invoke  SHGetPathFromIDList,lParam,addr @szBuffer
                        invoke  SendMessage,hWnd,BFFM_SETSTATUSTEXT,\
                                0,addr @szBuffer
                .endif
                xor     eax,eax
                ret

_BrowseFolderCallBack   endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_BrowseFolder   proc    _hWnd,_lpszBuffer
                local   @stBrowseInfo:BROWSEINFO
                local   @stMalloc
                local   @pidlParent,@dwReturn

                pushad
;********************************************************************
; 初始化 COM 库
;********************************************************************
                invoke  CoInitialize,NULL
                invoke  SHGetMalloc,addr @stMalloc
                .if     eax == E_FAIL
                        mov     @dwReturn,FALSE
                        jmp     @F
                .endif
                invoke  RtlZeroMemory,addr @stBrowseInfo,sizeof @stBrowseInfo
;********************************************************************
; SHBrowseForFolder 选择一个目录，把不含路径的目录名放入
; stBrowseInfo.pszDisplayName 中，SHGetPathFromIDList 把
; stBrowseInfo.pszDisplayName 转换成含全部路径的目录名
;********************************************************************
                push    _hWnd
                pop     @stBrowseInfo.hwndOwner
                push    _lpszBuffer
                pop     _BrowseFolderTmp
                mov     @stBrowseInfo.lpfn,offset _BrowseFolderCallBack
                mov     @stBrowseInfo.lpszTitle,offset _szDirInfo
```

```
                mov     @stBrowseInfo.ulFlags,\
                        BIF_RETURNONLYFSDIRS or BIF_STATUSTEXT
                invoke  SHBrowseForFolder,addr @stBrowseInfo
                mov     @pidlParent,eax
                .if     eax !=  NULL
                        invoke  SHGetPathFromIDList,eax,_lpszBuffer
                        mov     eax,TRUE
                .else
                        mov     eax,FALSE
                .endif
                mov     @dwReturn,eax
;*****************************************************************
; 释放 COM 库
;*****************************************************************
                mov     eax,@stMalloc
                mov     eax,[eax]
                invoke  (IMalloc PTR [eax]).Free,@stMalloc,@pidlParent
                mov     eax,@stMalloc
                mov     eax,[eax]
                invoke  (IMalloc PTR [eax]).Release,@stMalloc

                @@:
                invoke  CoUninitialize
                popad
                mov     eax,@dwReturn
                ret

_BrowseFolder   endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

C.2　使用浏览目录对话框

浏览目录对话框的实现分为两个部分：初始化部分和对话框功能部分。

1. 初始化 COM 库

浏览目录对话框要用到 COM 接口，所以必须首先调用 CoInitialize 函数来初始化 COM 库并调用 SHGetMalloc 函数来获取一个 IMalloc 类型的接口，这个接口实际上是一个内存块，中间包含各种 COM 功能模块的入口地址。在对话框返回的时候，再调用接口中的 Free 模块释放接口，并调用 CoUninitialize 函数释放 COM 库。

COM 编程是一个很大的课题，有关 COM 组件中的 IUnknown 接口和 IMalloc 接口等工作机理的内容本身就是一个比较深奥的问题，在本章中读者不必深入研究_BrowseFolder.asm 文件中关于这两种接口的定义，只要将定义部分和代码中初始化和释放 COM 库的代码直接拿过来用就是了。

COM 库的相关函数包含在 ole32.dll 中，所以源程序中必须包含下列语句：

```
include     ole32.inc
includelib  ole32.lib
```

2. 显示对话框

显示对话框的功能函数包含在 Shell32.dll 中，所以在源程序的头部也应该有下列包含语句：

```
include       shell32.inc
includelib    shell32.lib
```

显示对话框的功能由 SHBrowseForFolder 函数实现，函数的用法如下：

```
invoke        SHBrowseForFolder, lpbi
mov           lpItemIDList,eax          ;返回一个ITEMIDLIST结构指针
```

参数 lpbi 指向一个包含对话框初始化数据的 BROWSEINFO 结构，如下所示：

```
BROWSEINFO STRUCT
 HwndOwner             dd ?      ;对话框的父窗口
 PidlRoot              dd ?      ;用来表示起始目录的 ITEMIDLIST 目录
 PszDisplayName        dd ?      ;用来接收用户选择目录的缓冲区
 LpszTitle             dd ?      ;对话框中的用户定义文字
 ulFlags               dd ?      ;标志
 lpfn                  dd ?      ;回调函数地址
 lParam                dd ?      ;传给回调函数的参数
 iImage                dd ?      ;用来接收选中目录的图像
BROWSEINFO ENDS
```

结构的各个重要字段说明如下。

- lpszTitle——对话框中的自定义文字，如例子中显示的是“ 请选择目录 ”。
- lpfn——回调函数的地址。
- ulFlags——用来定义对话框类型的标志，下面是一些重要的标志：
  - BIF_BROWSEFORPRINTER——对话框中只能选择打印机。
  - BIF_BROWSEINCLUDEFILES——同时显示目录中的文件。
  - BIF_RETURNONLYFSDIRS——只返回文件系统中的目录。
  - BIF_STATUSTEXT——对话框中显示一个状态栏。
  - BIF_EDITBOX——显示一个编辑框供用户手工输入目录。
  - BIF_VALIDATE——显示编辑框的时候检测用户输入目录的合法性。

3. 对话框的回调函数

当函数执行后，将显示对话框，当对话框初始化以及每当用户选择不同的目录的时候，函数调用 lpfn 指定的回调函数，回调函数的参数有 4 个，分别是父窗口句柄 hWnd、消息类型 uMsg、消息参数 lParam 和自定义数据 lpData，回调函数的语法如下：

```
_BrowseFolderCallBack       proc     hWnd,uMsg,lParam,lpData

        mov      eax,uMsg
        .if      eax ==  BFFM_INITIALIZED
                 ...
        .elseif  eax ==  BFFM_SELCHANGED
```

```
                ...
        .endif
        xor     eax,eax
        ret

_BrowseFolderCallBack       endp
```

回调函数可能收到的消息有 3 种，如下所示：

- BFFM_INITIALIZED——在对话框初始化的时候收到。
- BFFM_SELCHANGED——在用户选择了一个目录的时候收到，这时 lParam 参数指向一个表示当前被选择目录的ITEMIDLIST结构。
- BFFM_VALIDATEFAILED——用户输入了一个不合法的目录名。

在回调函数中，程序可以根据情况向对话框发送控制消息，当收到 BFFM_SELCHANGED 消息时，可以根据选择情况决定是否允许用户单击对话框中的“ 确定 ”按钮，通过发送 BFFM_ENABLEOK 消息可以控制 “ 确定 ” 按钮的状态：

```
invoke  SendMessage,hWnd, BFFM_ENABLEOK,0,TRUE   ；允许 “ 确定 ” 按钮
invoke  SendMessage,hWnd, BFFM_ENABLEOK,0,FALSE  ；灰化 “ 确定 ” 按钮
```

程序也可以通过发送 BFFM_SETSELECTION 消息来设定目录：

```
invoke  SendMessage,hWnd,BFFM_SETSELECTION,TRUE,lpPath
```

如果消息的 wParam 参数为 TRUE，则目录用 lParam 参数指定的字符串表示 如果 wParam 为 FALSE，则目录用 lParam 指定的 ITEMIDLIST 结构表示。由于用字符串表示比较方便，所以例子程序在收到初始化消息的时候用上面的语句设置初始目录。

调用 SHBrowseForFolder 函数的时候，通过 BROWSEINFO 结构的 pidlRoot 字段也可以设置初始目录，但设置 ITEMIDLIST 结构比较麻烦，所以一般不使用这种方法。

另外，在回调函数中可以通过 BFFM_SETSTATUSTEXT 消息设置状态栏的文字，对话框的状态栏并不是指一般窗口底部的状态栏，而是指自定义文字下面的那一行文字，在图 8.6 中就是显示被选择目录名的地方，使用 BFFM_SETSTATUSTEXT 消息的格式如下：

```
invoke  SendMessage,hWnd,BFFM_SETSTATUSTEXT,0,lpsz
```

例子程序中，回调函数每次在收到用户选择目录的 BFFM_SELCHANGED 消息后，在状态栏中显示目录的名称。

4. 获取返回的目录

当 SHBrowseForFolder 返回的时候，如果用户单击的是 “ 取消 ” 按钮，那么函数的返回值是 0，否则的话，函数返回一个指向ITEMIDLIST结构的指针，对于这个结构可以不必去深究，因为使用 SHGetPathFromIDList 函数可以很方便地将它转换成目录字符串：

```
invoke  SHGetPathFromIDList,lpItemIDList,addr szPath
```

函数执行后，szPath 中就是字符串格式的用户选择的目录名称了。