# 第1章 背景知识

## 1.1 Win32的软硬件平台

### 1.1.1 80x86系列处理器简史

### 1.1.2 Windows的历史

### 1.1.3 Win32平台的背后——Wintel联盟

## 1.2 Windows的特色

## 1.3 必须了解的东西

### 1.3.1 80x86处理器的工作模式

### 1.3.2 Windows的内存管理

### 1.3.3 Windows的特权保护

# 第2章 准备编程环境

## 2.1 Win32可执行文件的开发过程

## 2.2 编译器和链接器

### 2.2.1 MASM系列

### 2.2.2 TASM系列

### 2.2.3 其他编译器

### 2.2.4 MASM、TASM还是NASM

### 2.2.5 我们的选择——MASM

## 2.3 创建资源

### 2.3.1 资源编译器的使用

### 2.3.2 所见即所得资源编辑器

## 2.4 make工具的使用

### 2.4.1 make工具是什么

### 2.4.2 nmake的用法

### 2.4.3 描述文件的语法

## 2.5 获取资料

### 2.5.1 Windows资料的来源

### 2.5.2 Intel处理器资料

## 2.6 构建编程环境

### 2.6.1 IDE还是命令行

### 2.6.2 本书推荐的编程环境

### 2.6.3 尝试编译第一个程序

# 第3章 使用MASM

## 3.1 Win32汇编程序的结构

Win32的Hello, world!程序：

.386

.model flat,stdcall

option casemap:none

include windows.inc

include user32.inc

include kernel32.inc

includelib user32.lib

includelib kernel32.lib

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

;数据段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.data

szCaption db 'A messageBox!',0

szText db 'Hello,world!',0

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

;代码段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.code

start:

invoke MessageBox,NULL,offset szText,offset szCaption,MB\_OK

invoke ExitProcess,0

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

end start

### 3.1.1 模式定义

程序的第一部分是模式和源程序格式的定义语句：

.386

.model flat,stdcall

option casemap:none

这些指令定义了程序使用的指令集、工作模式和格式。

**1.指令集选择**

.386 语句是汇编语言的伪指令，它在低版本的宏汇编中就已经存在，类似的指令还有.8086，.186，.286，.386/.386p，.486/.486p 和 .586/.586p 等，用于告诉编译器在本程序中使用的指令集。在 DOS 的汇编中默认使用的是 8086 指令集，那时候如果在源程序中写入80386 所特有的指令或使用 32 位的寄存器就会报错，为了在 DOS 环境下进行保护模式编程或仅为了使用32位寄存器，常在DOS的汇编中使用 .386来定义。Win32环境工作在80386及以上的处理器中，所以这一句 .386是必不可少的。

后面带p的伪指令则表示程序中可以使用特权指令，如：

mov cr0,eax

这一类指令必须在特权级0上运行，如果只指定 .386，那么使用普通的指令是可以的，编译时到这一句就会报错，如果我们要写的程序是VxD等驱动程序，需要用到特权指令，那么必须使用带’p’的指令集，否则当扫描到特权指令时汇编器会不识别。

80486和Pentium处理器指令是80386处理器指令的超集，同理，如果程序中要用 80486 处理器或 Pentium 处理器的指令，则必须定义 .486 或 .586。另外，Intel公司的80x86系列处理器从Pentium MMX开始增加了MMX指令集，为了使用MMX指令，除了定义 .586之外，还要加上一句 .mmx伪指令：

.586

.mmx

**2. 工作模式选择**

.model语句在低版本的宏汇编中已经存在， 用来定义程序工作的模式， 它的使用方法是：

.model 内存模式[,语言模式][,其他模式]

内存模式的定义影响最后生成的可执行文件，可执行文件的规模从小到大，可以有很多种类型，在DOS的可执行程序中，有只用到64 KB的 .com文件，也有大大小小的 .exe文件。到了Win32环境下，又有了可以用4 GB内存的PE格式可执行文件，编写不同类型的可执行文件要用 .model语句定义不同的参数，具体如表3.1所示。

表3-1 内存模式

|  |  |
| --- | --- |
| 内存模式 | 使用环境和条件 |
| Tiny | 用来建立.com文件，所有的代码、数据和堆栈都在同一个64KB段内 |
| Small | 建立代码和数据分别用一个64KB段的.exe文件 |
| Medium | 代码段可以有多个64KB段，数据段只有一个64KB段 |
| Compact | 代码段只有一个64KB段，数据段可以有多个64KB段 |
| Large | 代码段和数据段都可以有多个64KB段 |
| Huge | 同Large，并且数据段里面的一个数组也可以超过64KB |
| Flat | Win32程序使用的模式，代码和数据段可以使用同一个4GB段 |

在前面章节中已经提到过：Windows程序运行在保护模式下，系统把每一个Win32应用程序都放到分开的虚拟地址空间中去运行，也就是说，每一个应用程序都拥有其相互独立的4 GB 地址空间，对Win32程序来说，只有一种内存模式，即flat（平坦）模式，意思是内存是很“平坦”地从0延伸到 4 GB，再没有64 KB段大小限制。对比一下DOS版本的Hello World和Win32版本的Hello World开始部分的不同，DOS程序中有这样两句：

mov ax,data

mov ds,ax

意思是把数据段寄存器DS指向data数据段，data数据段在前面已经用 data segment 语句定义，只要DS不重新设置，那么从此以后指令中涉及的数据默认将从data数据段中取得，所以下面的语句是从data数据段取出szHello字符串的地址后再显示：

mov ah,9

mov dx,offset szHello

int 21h

纵观Win32汇编的源程序，没有一处可以找到ds或es等段寄存器的使用，因为所有的4 GB空间用32位的寄存器全部都能访问到了，不必在头脑中随时记着当前使用的是哪个数据段，这就是“平坦”内存模式带来的好处。

如果定义了 .model flat，MASM自动为各种段寄存器做了如下定义：

ASSUME cs:FLAT, ds:FLAT, ss:FLAT, es:FLAT, fs:ERROR, gs:ERROR

也就是说，CS，DS，ES 和 SS 段全部使用平坦模式，FS 和 GS 寄存器默认不使用，这时若在源程序中使用FS或GS，在编译时会报错。如果有必要使用它们，只需在使用前用下面的语句声明一下就可以了（比如在第14章的异常处理中要用到FS寄存器） ：

assume fs:nothing, gs:nothing 或者 assume fs:flat, gs:flat

在Win32汇编中，.model语句中还应该指定语言模式，即子程序的调用方式，例子中用的是stdcall，它指出了调用子程序或Win32 API时参数传递的次序和堆栈平衡的方法，相对于 stdcall，不同的语言类型还有 C，SysCall，BASIC，FORTRAN 和 PASCAL，虽然各种高级语言在调用子程序时都是使用堆栈来传递参数，但它们的处理方法各有不同。要和其他语言配合，就必须指定相应的语言种类。Windows 的 API 调用使用的是 stdcall 格式，所以在Win32 汇编中没有选择，必须在 .model 中加上 stdcall 参数。关于参数传递的细节，在 3.4.2节中有详细的描述。

**3. option语句**

用option语句定义的选项有很多，如option language定义和option segment定义等，在Win32 汇编程序中，需要的只是定义 option casemap:none，这个语句定义了程序中的变量和子程序名是否对大小写敏感，由于Win32 API中的API名称是区分大小写的，所以必须指定这个选项，否则在调用API的时候会有问题。

### 3.1.2 段的定义

**1. 段的概念**

把上面的Win32的Hello World源程序中的语句归纳精简一下，再列在下面：

.386

.model flat,stdcall

option casemap:none

<一些include语句>

.data

<一些字符串、变量定义>

.code

<代码>

<开始标号>

<其他语句>

end 开始标号

下面是包含全部段的源程序结构：

.386

.model flat,stdcall

option casemap:none

<一些include语句>

.stack [堆栈段的大小]

.data

<一些初始化过的变量定义>

.data?

<一些没有初始化过的变量定义>

.const

<一些常量定义>

.code

<代码>

<开始标号>

<其他语句>

end 开始标号

.stack，.data，.data?，.const 和 .code是分段伪指令，Win32中实际上只有代码和数据之分，.data，.data?和 .const都是数据段，.code是代码段，与DOS汇编不同，由于Win32汇编不必考虑堆栈，系统会为程序分配一个向下扩展的、足够大的段作为堆栈段，所以 .stack段定义常常被忽略。

|  |
| --- |
| 灯泡.BMP前面不是说过 Win32 环境下不用“段”了吗？是的，这些“段”实际上并不是 DOS 汇编中那种意义的段，而是内存的“分段”。上一个段的结束就是下一个段的开始，所有的“分段”合起来，包括系统使用的地址空间，就组成了整个可以寻址的4 GB空间。由于Win32环境的内存管理使用了80386处理器的分页机制，每个页（4 KB大小）可以自由指定属性，所以上一个4 KB可能是代码，属性是可执行但不可写，下一个4 KB就有可能是既可读也可写但不可执行的数据，再下面呢？有可能是可读不可写也不可执行的数据。Win32汇编源程序中“分段”的概念实际上是把不同类型的数据或代码归类，再放到不同属性的内存页（也就是不同的“分段”）中，这中间不涉及使用不同的段选择器。虽然使用和DOS汇编同样的 .code和 .data语句来定义，意思可是完全不同了！为了简单起见，在本书中还是简称“段”，读者应该注意到其中不同的含义。 |

**2. 数据段**

.data，.data?和 .const 定义的是数据段，分别对应不同方式的数据定义，在最后生成的可执行文件中也分别放在不同的节区（Section）中。程序中的数据定义一般可以归纳为 3 类。

.data区间存放的是已经初始化过的变量，这些初始化信息被保存到可执行文件中，因此这个区间中定义的变量越大，那么该程序最终生成的文件也就越大。.data段一般存放在可执行文件的\_DATA节区内。

.data?区间存放的是没有初始化的变量。在.DATA区间里面的变量的空间申请是在汇编的时候实现的，而在该区间里面的变量的空间是在程序被载入内存时申请的（像C语言里的malloc）。在汇编时，汇编器只记录了该区间中变量所占用的空间的大小，比如把.DATA?区间里面的一个变量从1B大小改为1000B大小，最终生成的可执行文件大小却没有增加。.data?段在可执行文件中一般存放在\_BSS节区中。

.const区间存放的是程序中的常量，该区间是具有只读属性的，即当程序运行时这个区间的内容是不能改变的。

如果不怕程序可读性不佳的话，把 .const段中定义的东西混到 .code段中去也可以正常使用，因为 .code段也是可以读的。

**3. 代码段**

.code段是代码段，所有的指令都必须写在代码段中，在可执行文件中，代码段一般是放在\_TEXT节区中的。Win32环境中的数据段是不可执行的，只有代码段有可执行的属性。对于工作在特权级3 的应用程序来说，.code段是不可写的，在编 DOS 汇编程序的时候，好事的程序员往往有个习惯，就是靠改动代码段中的代码来做一些反跟踪的事情，如果企图在Win32汇编下做同样的事情，结果就是和上面同样的“非法操作”。

当然事物总有两面性，在 Windows 95 下，在特权级 0 下运行的程序对所有的段都有读写的权利，包括代码段。另外，在优先级3下运行的程序也不是一定不能写代码段，代码段的属性是由可执行文件PE头部中的属性位决定的，通过编辑磁盘上的 .exe文件，把代码段属性位改成可写，那么在程序中就允许修改自己的代码段。一个典型的应用就是一些针对可执行文件的压缩软件和加壳软件，如Upx和PeCompact等，这些软件靠把代码段进行变换来达到解压缩或解密的目的，被处理过的可执行文件在执行时需要由解压代码来将代码段解压缩，这就需要写代码段，所以这些软件对可执行文件代码段的属性预先做了修改。

**4. 堆栈段**

在程序中不必定义堆栈段，系统会自动分配堆栈空间。惟一值得一提的是，堆栈段的内存属性是可读写并且是可执行的，这样靠动态修改代码的反跟踪模块可以拷贝到堆栈中去边修改边执行。一些病毒或者黑客工具用到的缓冲区溢出技术也用到了这个特征，有兴趣了解的读者可以查阅相关的资料。

### 3.1.3 程序结束和程序入口

在 C 语言源程序中，程序不必显式地指定程序由哪里开始执行，编译器已经约定好从main()函数开始执行了。而在汇编源程序中，并没有一个main函数，程序员可以指定从代码段的任何一个地方开始执行，这个地方由程序最后一句的end语句来指定：

end label

上面这条语句告诉汇编器在遇到这条指令的时候结束对源文件的汇编工作，同时也告诉汇编器程序的入口点是哪个标号。

但是，一个源程序不必非要指定入口标号，这时候可以把开始地址忽略不写，此时这个程序是一个模块。这种情况发生在编写多模块程序的单个模块的时候。当分开写多个程序模块时，每个模块的源程序中也可以包括 .data，.data?，.const和 .code段，只是end后没有入口地址。当最后把多个模块链接在一起的时候，只能有一个主模块指定入口地址， 在多个模块中指定入口地址或者没有一个模块指定了入口地址，链接器都会报错。

### 3.1.4 注释和换行

在MASM下Win32的注释是以分号’;’开始的，类似于C++的单行注释’//’。

当源程序一行过长时，可以用续航符’\’来分行书写，即在一行的最后用’\’来进行续行，如：

invoke MessageBox,NULL,offset szText,offset szCaption,MB\_OK

可以写为：

invoke MessageBox,\

Null,\ ;父窗口句柄

offset szText,\ ;消息框中的文字

offset szCaption,\ ;标题文字

MB\_OK

“一行的最后”是指最后一个对程序有影响的字符。而在续行符后加些空格和注释，都不会影响续行符的使用。

## 3.2 调用API

### 3.2.1 API是什么

在DOS下，操作系统的功能是通过各种软中断来实现的，如大家都知道int 21h是DOS中断，int 13h和int 10h是BIOS中的磁盘中断和视频中断。

DOS下中断方式的不便之处，首先，所有的功能号定义是冷冰冰的数字，其次，80x86 系列处理器能处理的中断最多只能有 256 个，不同的系统服务程序使用了不同的中断号，这可怜的中断数量显得太少了，结果到最后是中断挂中断，大家抢来抢去的，把好好的一个系统搞得像接力赛跑一样。

与DOS的结构相比，Win32的系统功能模块放在 Windows 的动态链接库（DLL）中，DLL 是一种 Windows 的可执行文件，采用的是和 .exe文件同样的PE格式， 在PE格式文件头的导出表中， 以字符串形式指出了这个DLL能提供的函数列表。应用程序使用字符串类型的函数名指定要调用的函数。

应用程序在使用的时候由Windows自动装入DLL程序并调用相应的函数。

实际上，Win32的基础就是由DLL组成的。Win32 API的核心由3个DLL提供，它们是：

● KERNEL32.DLL——系统服务功能。包括内存管理、任务管理和动态链接等。

● GDI32.DLL——图形设备接口。 利用VGA与DRV之类的显示设备驱动程序完成显示

文本和矩形等功能。

● USER32.DLL——用户接口服务。建立窗口和传送消息等。

当然，Win32 API还包括其他很多函数，这些也是由DLL提供的，不同的DLL提供了不同的系统功能。所有的这些DLL提供的函数组成了现在所用的Win32编程环境。

### 3.2.2 调用API

与在DOS中用中断方式调用系统功能一样，用API方式调用存放在DLL中的函数必须同样约定一个规范，用来定义函数的调用方法、参数的传递方法和参数的定义，洋洋洒洒几百 MB 的 Windows 系统比起才几百 KB 规模的 DOS，其系统函数的规模和复杂程度都上了一个数量级，所以在使用一个 API 时，带的参数数量多达十几个是常有的事，在 DOS 下用寄存器来传递参数的方法显然已经不能胜任了。

Win32 API是用堆栈来传递参数的，调用者把参数一个个压入堆栈，DLL中的函数程序再从堆栈中取出参数处理，并在返回之前将堆栈中已经无用的参数丢弃。在Microsoft发布的《Microsoft Win32 Programmer's Reference》中定义了常用API的参数和函数声明，先来看消息框函数的声明：

int MessageBox(

HWND hWnd, // handle to owner window

LPCTSTR lpText, // text in message box

LPCTSTR lpCaption, // message box title

UINT uType // message box style

);

最后还有一句说明：

Library: Use User32.lib.

上述函数声明说明了 MessageBox 有 4 个参数，它们分别是 HWND 类型的窗口句柄（hWnd） ，LPCTSTR类型的要显示的字符串地址（lpText）和标题字符串地址（lpCaption） ，还有UINT类型的消息框类型（uType） 。这些数据类型看起来很复杂，但有一点是很重要的，对于汇编语言来说，Win32环境中的参数实际上只有一种类型，那就是一个32位的整数，所有这些HWND，LPCTSTR和UINT实际上就是汇编中的dword（double word） ，之所以定义为不同的模样，是为了说明其用途。可能是因为Windows是用C写成的吧，或者是因为世界上的程序员用C语言的最多，Windows所有编程资料发布的格式都是用C格式的。

上面的声明用汇编的格式来表达就是：

MessageBox Proto hWnd:dword,lpText:dword,lpCaption:dword,uType:dword

上面最后一句Library: Use User32.lib则说明了这个函数包括在User32.dll中。

有了函数原型的定义以后，就是调用的问题了，Win32 API调用中要把参数放入堆栈，顺序是最后一个参数最先进栈，在汇编中调用MessageBox函数的方法是：

push uType

push lpCaption

push lpText

push hWnd

call MessageBox

在源程序编译链接成可执行文件后，call MessageBox语句中的MessageBox会被换成一个地址，指向可执行文件中的导入表，导入表中指向 MessageBox 函数的实际地址会在程序装入内存的时候，根据User32.dll在内存中的位置由Windows系统动态填入。

**1. 使用invoke语句**

API 是可以调用了，另一个烦人的问题又出现了，Win32的 API 动辄就是十几个参数，整个源程序一眼看上去基本上都是把参数压入堆栈的push指令， 参数的个数和顺序很容易搞错，由此引起的莫名其妙的错误源源不断，源程序的可读性看上去也很差。如果写的时候少写了一句push指令，程序在编译和链接的时候都不会报错，但在执行的时候必定会崩溃，原因是堆栈对不齐了。

有没有解决的办法呢？最好是像C语言一样，能在同一句中打入所有的参数，并在参数使用错误的时候能够提示。

Microsoft 终于做了一件好事，在 MASM 中提供了一个伪指令实现了这个功能，那就是invoke伪指令，它的格式是：

invoke 函数名[,参数1][,参数2]……

对MessageBox的调用在MASM中可以写成：

invoke MessageBox,NULL,offset szText,offset szCaption,MB\_OK

注意，invoke 并不是 80386 处理器的指令，而是一个 MASM 编译器的伪指令，在编译的时候由编译器把上面的指令展开成我们需要的4个push指令和1个call指令，同时，进行参数数量的检查工作，如果带的参数数量和声明时的数量不符，编译器会报错：

error A2137: too few arguments to INVOKE

编译时看到这样的错误报告，首先要检查的是有没有少写了一个参数。由于对于不带参数的API调用，invoke伪指令的参数检查功能可有可无，所以既可以用call API\_Name这样的语法也可以用invoke API\_Name这样的语法。

**2. API函数的返回值**

有的API函数有返回值，如MessageBox定义的返回值是int类型的数，返回值的类型对汇编程序来说也只有 dword 一种类型，它永远放在 eax 中。如果要返回的内容不是一个 eax所能容纳的，Win32 API 采用的方法一般是eax 中返回一个指向返回数据的指针，或者在调用参数中提供一个缓冲区地址，干脆把数据直接返回到缓冲区中去。

**3. 函数的声明**

在调用API函数的时候， 函数原型也必须预先声明， 否则， 编译器会不认这个函数。 invoke伪指令也无法检查参数个数。声明函数的格式是：

函数名 proto [距离] [语言] [参数1]：数据类型,[参数2]：数据类型,……

句中的proto是函数声明的伪指令， 距离可以是NEAR， FAR， NEAR16， NEAR32， FAR16或FAR32，Win32中只有一个平坦的段，无所谓距离，所以在定义时是忽略的；语言类型就是 .model那些类型，如果忽略，则使用 .model定义的默认值。

后面就是参数的列表了，由于Win32 API仅仅使用dword类型的参数，所以绝大多数的数据类型都是dword， 另外对于编译器来说， 它只关心参数的数量， 参数的名称在这里是“无用”的，仅是为了可读性而设置的，可以省略掉，所以下面两句消息框函数的定义实际上是一样的：

MessageBox Proto hWnd:dword,lpText:dword,lpCaption:dword,uType:dword

MessageBox Proto :dword,:dword,:dword,:dword

在 Win32 环境中，和字符串相关的 API 共有两类，分别对应两个字符集：一类是处理 ANSI 字符集的，另一类是处理 Unicode 字符集的。前一类函数名字的尾部带一个“A”字符，处理 Unicode的则带一个“W”字符。

|  |
| --- |
| 灯泡.BMP为了使程序更有移植性，在源程序中一般不直接指明使用 Unicode 还是 ANSI 版本，而是使用宏汇编中的条件汇编功能来统一替换，如在源程序中使用MessageBox，但在头文件中定义：  if UNICODE  MessageBox equ <MessageBoxW>  else  MessageBox equ <MessageBoxA>  endif  所有涉及版本问题的API都可以按此方法定义， 然后在源程序的头部指定UNICODE=1或UNICODE=0，重新编译后就能产生不同的版本。 |

**4．include语句**

对于所有要用到的API函数，在程序的开始部分都必须预先声明，但这个步骤显然是比较麻烦的，为了简化操作，可以采用各种语言通用的解决办法，就是把所有的声明预先放在一个文件中，在用到的时候再用include语句包含进来。现在回到Win32 Hello World程序，这个程序用到了两个 API 函数：MessageBox 和 ExitProcess，它们分别在 User32.dll 和Kernel32.dll中， 在MASM32工具包中已经包括了所有DLL的API函数声明列表， 每个DLL对应<DLL名.inc>文件，在源程序中只要使用include语句包含进来就可以了：

include user32.inc

include kernel32.inc

当用到其他的API函数时，只需相应增加对应的include语句。

include语句还用来在源程序中包含其他文件，当多个源程序用到相同的函数定义、常量定义，甚至源代码时，可以把相同的部分写成一个文件，然后在不同的源程序中用 include语句包含进来。

编译器对include语句的处理仅是简单地把这一行用指定的文件内容替换掉而已。

include语句的语法是：

include 文件名

或 include <文件名>

当遇到要包括的文件名和 MASM 的关键字同名等可能会引起编译器混淆的情况时，可以用“< >”将文件名括起来。

**5. includelib语句**

在使用外部函数的时候，DOS下有函数库的概念，那时的函数库实际上是静态库，静态库是一组已经编写好的代码模块，在程序中可以自由引用，在源程序编译成目标文件，最后要链接成可执行文件的时候，由link程序从库中找出相应的函数代码，一起链接到最后的可执行文件中。

Win32 环境中，程序链接的时候仍然要使用函数库来定位函数信息，只不过由于函数代码放在DLL文件中，库文件中只留有函数的定位信息和参数数目等简单信息，这种库文件叫做导入库， 一个DLL文件对应一个导入库， 如User32.dll文件用于编程的导入库是User32.lib，MASM32工具包中包含了所有DLL的导入库。

为了告诉链接程序使用哪个导入库，使用的语句是：

includelib 库文件名

或 includelib <库文件名>

与include的用法一样，在要包括让编译器混淆的文件名时加方括号。Win32 Hello world程序用到的两个API函数MessageBox和ExitProcess分别在User32.dll和Kernel32.dll中，那么在源程序使用的相应语句为：

includelib user32.lib

includelib kernel32.lib

和 include 语句的处理不同，includelib 不会把 .lib 文件插入到源程序中，它只是告诉链接器在链接的时候到指定的库文件中去找API函数的位置信息而已。

### 3.2.3 API参数中的等值定义

打开\masm32\include 目录下的 Windows.inc 查看一下，可以发现整个文件总共有两万六千多行，包括了几乎所有的Win32 API参数中的常量和数据结构定义。正是有了这个文件中详尽的定义， Win32ASM才得以流行起来， 试想一下， 哪个程序员愿意每使用一个API语句，就到函数手册中去看参数，然后到Microsoft发布的Visual C++的头文件中去找对应的数值，再应用到汇编源程序中？这样会有80%以上的时间花在做无用功上 （最后还是要骂Microsoft为什么不提供汇编格式的头文件，毕竟MASM32工具包不是Microsoft出的） 。

有时候由于版本的原因，当使用最新的 API 手册时，会发现有些参数使用的常量在Windows.inc中并没有定义，这下惨了，谁都不知道类似于MB\_XXXYYY的东西代表什么数值，Microsoft的《Microsoft Programmer's Reference》手册中从来就不会把参数对应的数值写进去。 遇到这种情况， 只有拿出最原始的办法了， 就是到最新的Visual C++或SDK的include目录中去，在C语言格式的 .h头文件中把定义找出来，然后自行增补到Windows.inc中去。如果这样也找不到定义值的话，那只好放弃使用这个API了。

## 3.3 标号、变量和数据结构

当程序中要跳转到另一位置时，需要有一个标识来指示新的位置，这就是标号，通过在目标地址的前面放上一个标号，可以在指令中使用标号来代替直接使用地址。

变量是计算机内存中已命名的存储位置，不同的变量有不同的类型和大小。按照定义的位置不同，MASM中的变量也分为全局变量和局部变量两种。

在MASM中标号和变量的命名规范是相同的，它们是：

1）可以用字母、数字、下划线、以及符号@、$和?。

2）$和?已经被作为特殊的标志符使用，因此不能单独用这两个符号来命名。

3）变量名不能和汇编器所定义的伪指令同名。

4）不能使用指令名等关键字。

5）第一个符号不能是数字。

6）长度不能超过240个字符。

7）在作用域内必须是唯一的。

### 3.3.1 标号

**1. 标号的定义**

标号既可以定义在目的指令同一行的头部，也可以在目的指令前一行单独用一行定义标号定义的格式是：

标号名: 目的指令 ;方法1

或

标号名:: 目的指令 ;方法2

常用的方法是使用方法1（标号后跟一个冒号） ，这时标号的作用域是当前的子程序，在单个子程序中的标号不能同名，否则编译器不知该用哪个地址，但在不同的子程序中可以有相同名称的标号，这意味着不能从一个子程序中用跳转指令跳到另一个子程序中。 需要从一个子程序中用跳转指令跳到另一个子程序中的标号时，可以用方法 2（标号后跟两个冒号）来定义，这时标号的作用域是整个程序，对任何其他子程序都是可见的。

在低版本的 MASM 中，标号在整个程序中是惟一的，子程序中的标号也可以从整个程序的任何地方转入（相当于任何标号都是用方法 2 定义的） 。而 Win32 汇编使用的高版本MASM中默认的标号作用域是当前的子程序，这是因为为了提供对局部变量和参数的支持，编译器会在子程序的入口处自动加上对堆栈的初始化指令，从子程序的中间进入相当于跳过了这些初始化指令，访问局部变量和参数时会出问题。

当然，如果程序员确认在子程序的中间进入不会引起问题（如用不到局部变量或参数） ，

那完全可以用方法2来定义作为入口的标号。

**2. MASM中的@@**

在 DOS 时代，为标号起名是个麻烦的事情，因为汇编指令用到跳转指令特别多，任何比较和测试等都要涉及跳转，所以在程序中会有很多标号，在整个程序范围内起个不重名的标号要费一番工夫，结果常常用 addr1 和 addr2 之类的标号一直延续下去，如果后来要在中间插一个标号，那么就常常出现addr1\_1和loop10\_5之类奇怪的标号。

实际上，很多标号只会使用一到两次，而且不一定非要起个有意义的名称，如汇编程序中下列代码结构很多：

mov cx,1234h

cmp flag,1

jz loc1

mov cx,1000h

loc1:

…

loop loc1

loc1在别的地方就再也用不到了， 对于这种情况， 高版本的MASM用@@标号去代替它：

mov cx,1234h

cmp flag,1

jz @F

mov cx,1000h

@@:

…

loop @B

当用@@做标号时， 可以用@F和@B来引用它， @F表示本条指令后的第一个@@标号，@B表示本条指令前的第一个@@标号，程序中可以有多个@@标号，但@B和@F只寻找匹配最近的一个。

|  |
| --- |
| warning.bmp不要在间隔太远的代码中使用@@标号，因为在以后的修改中，@@和@B，@F 中间可能会被无意中插入一个新的@@，这样一来，@B 或@F 就会引用到错误的地方去，源程序中@@标号和跳转指令之间的距离最好限制在编辑器能够显示的同一屏幕的范围内。 |

### 3.3.2 全局变量

**1. 全局变量的定义**

全局变量的作用域是整个程序，Win32汇编的全局变量定义在 .data或 .data?段内，可以同时定义变量的类型和长度，格式是：

变量名 类型 初始值1,初始值2，……

变量名 类型 重复数量 dup （初始值1,初始值2，……）

MASM中可以定义的变量类型相当多，具体如表3-2所示。

表3-2 数据类型

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 表示方式 | 缩写 | 长度（字节） |
| 字节（byte） | Byte | db | 1 |
| 字（word） | Word | dw | 2 |
| 双字（doubleword） | dword | dd | 4 |
| 三字（farword） | Fword | df | 6 |
| 四字（quadword） | Qword | dq | 8 |
| 十字节BCD码（tenbyte） | Tbyte | dt | 10 |
| 有符号字节（signbyte） | Sbyte |  | 1 |
| 有符号字（signword） | Sword |  | 2 |
| 有符号双字（signdword） | Sdword |  | 4 |
| 单精度浮点数 | Real4 |  | 4 |
| 双精度浮点数 | Real8 |  | 8 |
| 10字节浮点数 | Real10 |  | 10 |

所有使用到变量类型的情况中，只有定义全局变量的时候类型才可以用缩写。

**2. 全局变量的初始化值**

全局变量在定义中既可以指定初值，也可以只用问号预留空间，在 .data?段中，只能用问号预留空间，因为 .data?不能指定初始值，这里就有一个问题：既然可以用问号预留空间，那么在实际运行的时候，这个未初始化的值是随机的还是确定的呢？答案是 0，所以用问号指定的全局变量如果要以0为初始值的话，在程序中可以不必特地为它赋值。

### 3.3.3 局部变量

局部变量的作用域是单个子程序，在进入子程序的时候，通过修改堆栈指针 esp 来预留出需要的空间，在用ret指令返回主程序之前，同样通过恢复esp丢弃这些空间，这些变量就随之无效了。它的缺点就是因为空间是临时分配的，所以无法定义含有初始化值的变量，对局部变量的初始化一般在子程序中由指令完成。

在 DOS 时代，低版本的宏汇编本来无所谓全局变量和局部变量，所有的变量都是定义在数据段里面的，能被所有的子程序或主程序存取，就相当于现在所说的全局变量，用汇编语言在堆栈中定义局部变量是很麻烦的一件事情。要和高级语言做混合编程的时候，程序员

往往很痛苦地在边上准备一张表，表上的内容是局部变量名和ebp指针的位置关系。

**1. 局部变量的定义**

MASM用local伪指令提供了对局部变量的支持。定义的格式是：

local 变量名1[[重复数量]][:类型],变量名2[[重复数量]][:类型]……

local伪指令必须紧接在子程序定义的伪指令proc后、其他指令开始前，这是因为局部变量的数目必须在子程序开始的时候就确定下来，在一个local语句定义不下的时候，可以有多个local语句，语法中的数据类型不能用表3-2中的缩写，如果要定义数据结构，可以用数据结构的名称当做类型。Win32 汇编默认的类型是 dword，如果定义 dword 类型的局部变量，则类型可以省略。当定义数组的时候，可以 [] 括号括起来，不能使用定义全局变量的 dup伪指令。局部变量不能和已定义的全局变量同名。局部变量的作用域是当前的子程序，所以在不同的子程序中可以有同名的局部变量。

这里有几个定义局部变量的例子：

local loc1[1024]:byte ;例1

local loc2 ;例2

local loc3:WNDCLASS ;例3

下面是局部变量使用的一个典型的例子：

TestProc proc

local @loc1:dword,@loc2:word

local @loc3:byte

mov eax,@loc1

mov ax,@loc2

mov al,@loc3

ret

TestProc endp

这是一个名为TestProc的子程序，用local语句定义了3个变量，@loc1是dword类型，@loc2 是 word 类型，@loc3 是 byte 类型，在程序中分别有 3 句存取 3 个局部变量的指令，然后就返回了，编译成可执行文件后，再把它反汇编就得到了以下指令：

:00401000 55 push ebp

:00401001 8BEC mov ebp, esp

:00401003 83C4F8 add esp, FFFFFFF8

:00401006 8B45FC mov eax, dword ptr [ebp-04]

:00401009 668B45FA mov ax, word ptr [ebp-06]

:0040100D 8A45F9 mov al, byte ptr [ebp-07]

:00401010 C9 leave

:00401011 C3 ret

可以看到，反汇编后的指令比源程序多了前后两段指令，它们是：

:00401000 55 push ebp

:00401001 8BEC mov ebp, esp

:00401003 83C4F8 add esp, FFFFFFF8

…

:00401010 C9 leave

这些就是使用局部变量所必需的指令，分别用于局部变量的准备工作和扫尾工作。执行了 call 指令后，CPU 把返回的地址压入堆栈，再转移到子程序执行，esp 在程序的执行过程中可能随时用到，不可能用 esp 做指针来存取局部变量。大家一定有印象，在介绍寄存器的时候提到过 ebp 寄存器也是以堆栈段为默认数据段的，所以可以用 ebp 做指针，于是，在初始化前，先用一句push ebp指令把原来的ebp保存起来，然后把esp的值放到ebp中，供存取局部变量做指针用，再后面就是在堆栈中预留空间了，由于堆栈是向下增长的，所以要在esp 中加一个负值，FFFFFFF8 就是–8。慢着！一个 dword 加一个 word 再加一个字节不是 7吗，为什么是8呢？这是因为在80386处理器中，以dword为界对齐时存取内存速度最快，所以 MASM 宁可浪费一个字节。执行了这 3 句指令后，初始化完成，就可以进行正常的操作了，从指令中可以看出局部变量在堆栈中的位置排列，如表3-3所示。

表3-3 上例中局部变量排列的顺序

|  |  |
| --- | --- |
| ebp偏移 | 内容 |
| ebp+4 | 由call指令推入的返回地址 |
| ebp | push ebp指令推入的原ebp值，然后新的ebp=现在的esp |
| ebp-4 | 第一个局部变量 |
| ebp-6 | 第二个局部变量 |
| ebp-7 | 第三个局部变量 |

在程序退出的时候，必须把正确的esp设置回去，否则，ret指令会从堆栈中取出错误的地址返回，看程序可以发现，ebp 就是正确的初始 esp 值，因为子程序开始的时候已经有一句mov ebp,esp，所以要返回的时候只要先mov esp,ebp，然后再pop ebp，堆栈就是正确的了。

在 80386 指令集中有一条指令可以在一句中实现 mov esp,ebp 和 pop ebp 的功能，就是leave指令，所以，编译器在ret指令之前只使用了一句leave指令。

明白了局部变量使用的原理，就很容易理解使用时的注意点：ebp 寄存器是关键，它起

到保存原始 esp 的作用，并随时用做存取局部变量的指针基址，所以在任何时刻，不要尝试把ebp用于别的用途，否则会带来意想不到的后果。

|  |
| --- |
| 灯泡.BMPWin32汇编中局部变量的使用方法可以解释一个很有趣的现象： 在DOS汇编的时候， 如果在子程序中的push指令和pop指令不配对，那么返回的时候ret指令从堆栈里得到的肯定是错误的返回地址，程序也就死掉了。但在 Win32 汇编中，push 指令和 pop 指令不配对可能在逻辑上产生错误，却不会影响子程序正常返回， 原因就是在返回的时候esp不是靠相同数量的push和pop指令来保持一致的， 而是靠 leave 指令从保存在 ebp 中的原始值中取回来的，也就是说，即使把 esp 改得一塌糊涂也不会影响到子程序的返回，当然，“窍门”就在ebp，把ebp改掉，程序就玩完了！ |

**2. 局部变量的初始化值**

显然，局部变量是无法在定义的时候指定初始化值的，因为local伪指令只是简单地把空间给留出来，那么开始使用时它里面是什么值呢？和全局变量不一样，局部变量的起始值是随机的，是其他子程序执行后在堆栈里留下的垃圾。

|  |
| --- |
| warning.bmp在 API 函数使用的大量数据结构中，往往用 0 做默认值，如果用局部变量定义数据结构，初始化时只定义了其中的部分字段， 那么剩余字段的当前值可能是编程者预想不到的数值， 传给API函数后，执行的结果可能是意想不到的，这是初学者很容易忽略的一个问题。所以最好的办法是：在赋值前首先  将整个数据结构填 0，然后再初始化要用的字段，这样其余的字段就不必一个个地去填 0 了，RtlZeroMemory这个API函数就是实现填0的功能的。 |

### 3.3.4 数据结构

数据结构实际上是由多个字段组成的数据“样板”，相当于一种自定义的数据类型，数据结构中间的每一个字段可以是字节、字、双字、字符串或所有可能的数据类型。比如在API函数RegisterClass中要使用到一个叫做WNDCLASS的数据结构，Microsoft的手册中是如下定义的：

typedef struct \_WNDCLASS {

UINT style;

WNDPROC lpfnWndProc;

int cbClsExtra;

int cbWndExtra;

HINSTANCE hInstance;

HICON hIcon;

HCURSOR hCursor;

HBRUSH hbrBackground;

LPCTSTR lpszMenuName;

LPCTSTR lpszClassName;

} WNDCLASS, \*PWNDCLASS;

注意，这是 C 语言格式的，这个数据结构包含了 10 个字段，字段的名称是 style，lpfnWndProc和cbClsExtra等， 前面的UINT和WNDPROC等是这些字段的类型， 在汇编中，数据结构的写法如下：

结构名 struct

字段1 类型 ?

字段2 类型 ?

……

结构名 ends

上面的WNDCLASS结构定义用汇编的格式来表示就是：

WNDCLASS struct

style DWORD ?

lpfnWndProc DWORD ?

cbClsExtra DWORD ?

cbWndExtra DWORD ?

hInstance DWORD ?

hIcon DWORD ?

hCursor DWORD ?

hbrBackground DWORD ?

lpszMenuName DWORD ?

lpszClassName DWORD ?

WNDCLASS ends

定义了数据结构以后就可以多次在源程序中用这个“样板”当做数据类型来定义数据，使用数据结构在数据段中定义数据的方法如下：

.data?

stWndClass WNDCLASS <>

……

或者：

.data

stWndClass WNDCLASS <1,1,1,1,1,1,1,1,1,1>

……

这个例子定义了一个以WNDCLASS为结构的变量stWndClass，第一段的定义方法是未初始化的定义方法，第二段是在定义的同时指定结构中各字段的初始值，各字段的初始值用逗号隔开，在这个例子中10个字段的初始值都指定为1。

在汇编中，数据结构的引用方法有好几种，以上面的定义为例，如果要使用stWndClass中的lpfnWndProc字段，最直接的办法是：

mov eax,stWndClass.lpfnWndProc

它表示把 lpfnWndProc 字段的值放入 eax 中去，假设 stWndClass 在内存中的地址是403000h，这句指令会被编译成mov eax,[403004h]，因为lpfnWndProc是stWndClass中的第二个字段，第一个字段是dword，已经占用了4字节的空间。

在实际使用中，常常有使用指针存取数据结构的情况，如果使用esi寄存器做指针寻址，可以使用下列语句完成同样的功能：

mov esi,offset stWndClass

mov eax,[esi + WNDCLASS.lpfnWndProc]

如果要对一个数据结构中的大量字段进行操作，这种写法显然比较烦琐，MASM还有一个用法，可以用assume伪指令把寄存器预先定义为结构指针，再进行操作：

mov esi,offset stWndClass

assume esi:ptr WNDCLASS

mov eax,[esi].lpfnWndProc

…

assume esi:nothing

这样，使用寄存器也可以用逗点引用字段名，程序的可读性比较好。这样的写法在最后编译成可执行程序的时候产生同样的代码。注意：在不再使用 esi 寄存器做指针的时候要用assume esi:nothing取消定义。

结构的定义也可以嵌套，如果要定义一个新的 NEW\_WNDCLASS 结构，里面包含一个老的WNDCLASS结构和一个新的dwOption字段，那么可以如下定义：

NEW\_WNDCLASS struct

dwOption dword ?

oldWndClass WNDCLASS <>

NEW\_WNDCLASS ends

假设现在esi是指向一个NEW\_WNDCLASS的指针，那么引用里面嵌套的oldWndClass中的lpfnWndProc字段时，就可以用下面的语句：

mov eax,[esi].oldWndClass.lpfnWndProc

### 3.3.5 变量的使用

**1. 以不同的类型访问变量**

这个话题有点像C语言中的数据类型强制转换，在MASM中以不同的类型访问不会对变量造成影响。

举一个简单的例子，先以db方式定义一个缓冲区：

szBuffer db 1024 dup (?)

然后从其他地方取得了数据，但数据的格式是以字方式组织的，要处理数据，最有效的方法是两个字节两个字节地处理，但如果在程序中把szBuffer的值放入ax：

mov ax,szBuffer

编译器会报一个错：

error A2070: invalid instruction operands

意思是无效的指令操作，为什么呢？因为 szBuffer 是用db 定义的，而 ax 的尺寸是一个word，等于两个字节，尺寸不符合。MASM中，如果要用指定类型之外的长度访问变量，必须显式地指出要访问的长度，这样，编译器忽略语法上的长度检验，仅使用变量的地址。使用的方法是：

类型 ptr 变量名

类型可以是byte，word，dword，fword，qword，real8和real10。如：

mov ax,word ptr szBuffer

mov eax,dword ptr szBuffer

上述语句能通过编译，当然，类型必须和操作的寄存器长度匹配。在这里要注意的是，指定类型的参数访问并不会去检测长度是否溢出，看下面一段代码：

.data

bTest1 db 12h

wTest2 dw 1234h

dwTest3 dd 12345678h

…

.code

…

mov al,bTest1

mov ax,word ptr bTest1

mov eax,dword ptr bTest1

…

上面的程序片断，每一句执行后寄存器中的值是什么呢，mov al,bTest1这一句很显然使al等于12h， 下面的两句呢， ax和eax难道等于0012h和00000012h吗？实际运行结果很 “奇怪” ，竟然是3412h和78123412h，为什么呢？（留给读者自己思考）

如果程序员的本意是类似于C语言的强制类型转换， 想把bTest1的一个字节扩展到一个字或一个双字再放到ax或eax中，高位保持0而不是越界存取到其他的变量，可以用80386的扩展指令来实现。80386处理器提供的movzx指令可以实现这个功能，例如：

movzx ax,bTest1 ;例1

movzx eax,bTest1 ;例2

movzx eax,cl ;例3

movzx eax,ax ;例4

● 例1把单字节变量bTest1的值扩展到16位放入ax中。

● 例2把单字节变量bTest1的值扩展到32位放入eax中。

● 例3把cl中的8位值扩展到32位放入eax中。

● 例4把ax中的16位值扩展到32位放入eax中。

用movzx指令进行数据长度扩展是Win32汇编中经常用到的技巧， 该指令总是将扩展的数据位用0代替。使用另一条指令movsx可以完成带符号位的扩展，当被扩展数据的最高位为0时，效果和movzx指令相同；当最高位为1时，则扩展部分的数据位全部用1填充。

**2. 变量的尺寸和数量**

在源程序中用到变量的尺寸和数量的时候，可以用 sizeof 和 lengthof 伪指令来实现，格式是：

sizeof 变量名、数据类型或数据结构名

lengthof 变量名、数据类型或数据结构名

sizeof伪指令可以取得变量、数据类型或数据结构以字节为单位的长度，lengthof可以取得变量中数据的项数。假如定义了以下数据：

stWndClass WNDCLASS <>

szHello db 'Hello,world!',0

dwTest dd 1,2,3,4

…

.code

…

mov eax,sizeof stWndClass

mov ebx,sizeof WNDCLASS

mov ecx,sizeof szHello

mov edx,sizeof dword

mov esi,sizeof dwTest

执行后 eax 的值是 stWndClass 结构的长度 40，ebx 同样是 40，ecx 的值是 13，就是“Hello,world!”字符串的长度加上一个字节的0结束符，edx的值是一个双字的长度：4，而esi则等于4个双字的长度16。

如果把所有的sizeof换成lengthof，那么eax会等于1，因为只定义了1项WNDCLASS，而ecx同样等于13，esi则等于4，而lengthof WNDCLASS和lengthof dword是非法的用法，编译程序会报错。

要注意的是， sizeof和lengthof的数值是编译时候产生的， 由编译器直接替换到指令中去，上边的指令最后产生的代码就是：

mov eax,40

mov ebx,40

mov ecx,13

mov edx,4

mov esi,16

|  |
| --- |
| warning.bmp如果为了把Hello和World分两行定义，szHello是这样定义的：  szHello db 'Hello',0dh,0ah  db 'World',0  那么sizeof szHello是多少呢？注意！是7而不是13，MASM中的变量定义只认一行，后一行db 'World',0实际上是另一个没有名称的数据定义，编译器认为sizeof szHello是第一行字符的数量。虽然把szHello的地址当参数传给MessageBox 等函数显示时会把两行都显示出来，但严格地说这是越界使用变量。虽然在实际的应用中这样定义长字符串的用法很普遍，因为如果要显示一屏幕帮助，一行是不够的，但要注意的是：要用到这种字符串的长度时，千万不要用sizeof去表示，最好是在程序中用lstrlen函数去计算。 |

**3. 获取变量地址**

获取变量地址的操作对于全局变量和局部变量是不同的。

对于全局变量，它的地址在编译的时候已经由编译器确定了，它的用法大家都不陌生：

mov 寄存器,offset 变量名

其中 offset 是取变量地址的伪操作符，和 sizeof 伪操作符一样，它仅把变量的地址代到指令中去，这个操作是在编译时而不是在运行时完成的。

对于局部变量，它是用ebp来做指针操作的，假设ebp的值是40100h，那么局部变量1的地址是ebp-4即400FCh，由于ebp的值随着程序的执行环境不同可能是不同的，所以局部变量的地址值在编译的时候也是不确定的，不可能用offset伪操作符来获取它的地址。

80386处理器中有一条指令用来取指针的地址，就是lea指令，如：

lea eax,[ebp-4]

该指令可以在运行时按照ebp的值实际计算出地址放到eax中。

如果要在invoke伪指令的参数中用到一个局部变量的地址，该怎么办呢？参数中是不可能写入 lea 指令的，用 offset 又是不对的。MASM 对此有一个专用的伪操作符 addr，其格式为：

addr 局部变量名和全局变量名

当addr后跟全局变量名的时候，编译器自动按照offset的用法来使用；当addr后面跟局部变量名的时候，编译器会自动用lea指令先把地址取到eax中，然后用eax来代替变量地址使用。

要注意的是：对局部变量取地址的时候，addr伪操作符只能用在invoke的参数中，不能用在如下的 mov 指令中，这种限制很好理解，因为这种情况下，lea 指令如何能被代到语句里面呢：

mov eax,addr 局部变量名 ;注意：这是错误的用法

|  |
| --- |
| warning.bmp假设在一个子程序中有如下invoke指令：  invoke Test,eax,addr szHello  其中 Test 是一个需要两个参数的子程序，szHello 是一个局部变量，会发生什么结果呢？编译器会把invoke伪指令和addr翻译成下面这个模样：  lea eax,[ebp-4]  push eax ;参数2：addr szHello  push eax ;参数1：eax  call Test  发现了什么？到push第一个参数eax之前，eax的值已经被lea eax,[ebp-4]指令覆盖了！也就是说，要用到的eax的值不再有效，所以，当在invoke中使用addr伪操作符时，注意在它的左边不能用eax，否则eax 的值会被覆盖掉，当然 eax 用在 addr 右边的参数中用是可以的。幸亏 MASM 编译器对这种情况有如下错误提示：  error A2133: register value overwritten by INVOKE  否则，不知道又会引出多少莫名其妙的错误！ |

## 3.4 使用子程序

### 3.4.1 子程序的定义

子程序的定义方式如下所示。

子程序名 proc [距离][语言类型][可视区域][USES 寄存器列表][,参数:类型]...[VARARG]

local 局部变量列表

指令

子程序名 endp

proc 和 endp 伪指令定义了子程序开始和结束的位置， proc 后面跟的参数是子程序的属性和输入参数。子程序的属性有：

● 距离——可以是 NEAR，FAR，NEAR16，NEAR32，FAR16 或 FAR32，Win32 中只有一个平坦的段，无所谓距离，所以对距离的定义往往忽略。

● 语言类型——表示参数的使用方式和堆栈平衡的方式，可以是StdCall，C，SysCall，BASIC、FORTRAN和PASCAL，如果忽略，则使用程序头部 .model定义的值。

● 可视区域——可以是PRIVATE，PUBLIC和EXPORT。PRIVATE表示子程序只对本模块可见； PUBLIC表示对所有的模块可见 （在最后编译链接完成的 .exe文件中） ； EXPORT表示是导出的函数，当编写DLL的时候要将某个函数导出的时候可以这样使用。默认的设置

是PUBLIC。

● USES寄存器列表——表示由编译器在子程序指令开始前自动安排push这些寄存器的指令，并且在 ret 前自动安排 pop 指令，用于保存执行环境，但笔者认为不如自己在开头和结尾用pushad和popad指令一次保存和恢复所有寄存器来得方便。

● 参数和类型——参数指参数的名称，在定义参数名的时候不能跟全局变量和子程序中的局部变量重名。对于类型，由于Win32中的参数类型只有32位（dword）一种类型，所以可以省略。 在参数定义的最后还可以跟VARARG， 表示在已确定的参数后还可以跟多个数量不确定的参数，在Win32汇编中惟一使用VARARG的 API就是wsprintf，类似于C语言中的printf，其参数的个数取决于要显示的字符串中指定的变量个数。

完成了定义之后，可以用invoke伪指令来调用子程序，当invoke伪指令位于被调用的子程序代码之前的时候，编译器处理到invoke语句的时候还没有扫描到子程序的定义信息，所以会有以下错误信息：

error A2006: undefined symbol : 子程序名

这并不是说子程序的编写有错误，而是invoke伪指令无法得知子程序的定义情况，所以无法进行参数的检测。在这种情况下，为了让invoke指令能正常使用，必须在程序的头部用proto伪操作定义子程序的信息， “提前”告诉invoke语句关于子程序的信息，proto的用法见3.2.2节。当然，如果被调用的子程序定义在invoke语句前面的话，proto语句就可以省略了。

3.4.2 参数传递和堆栈平衡

在调用子程序时，参数的传递是通过堆栈进行的，也就是说，调用者把要传递给子程序的参数压入堆栈，子程序在堆栈中取出相应的值再使用，比如，如果要调用：

SubRouting(Var1,Var2,Var3)

经过编译后的最终代码可能 ．． 是（注意只是“可能” ） ：

push Var3

push Var2

push Var1

call SubRouting

add esp,12

也就是说，调用者首先把参数压入堆栈，然后调用子程序，在完成后，由于堆栈中先前压入的数不再有用，调用者或者被调用者必须有一方把堆栈指针修正到调用前的状态，这就叫堆栈的平衡。参数是最右边的先入堆栈还是最左边的先入堆栈、还有由调用者还是被调用者来修正堆栈都必须有个约定（称为调用约定） ，不然就会产生错误的结果，这就是在上述文字中使用“可能”这两个字的原因。由于各种语言默认的调用约定是不同的，所以在proc，以及 proto 语句的语言属性中确定语言类型后，编译器才可能将 invoke 伪指令翻译成正确的样子，不同语言的不同点如表3-4所示。

表3-4各种语言类型之间的属性差别

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 比较的项目\语言模式 | C | SYSCALL | STDCALL | BASIC | FORTRAN | PACAL |
| 子程序名要用大写 |  |  |  | √ | √ | √ |
| 参数是从左往右压栈的 |  |  |  | √ | √ | √ |
| 参数是从右往左压栈的 | √ | √ | √ |  |  |  |
| 堆栈是被调用者清除的 | √ |  |  |  |  |  |
| 堆栈是被子程序清除的 |  | √ | √ | √ | √ | √ |
| 可以使用VARARG属性 | √ | √ | √① |  |  |  |

在使用STDCALL语言模式的时候，如果在子程序里使用VARARG参数属性，那么子程序的堆栈清楚工作是由调用者完成的，否则由子程序来完成堆栈的清除工作。

现在来分析一下 ebp 指针和参数之间的对应关系，注意，这里是以 Win32 中的 StdCall 为例，不同的语言类型，指针的顺序可能是不同的。 假定在一个子程序中有两个参数，主程序调用时在 push 第一个参数前的堆栈指针 esp为X，那么压入两个参数后的esp为X-8，程序开始执行call指令，call指令把返回地址压入堆栈，这时候esp为X-C，接下去是子程序中用push ebp来保存ebp的值，esp变为X-10，再执行一句mov ebp,esp， 就可以开始用ebp存取参数和局部变量了， 图3-4说明了这个过程。

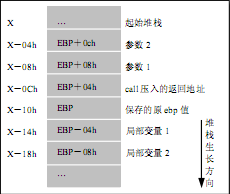


图3.4 ebp指针、参数和局部变量的关系

在子程序中使用参数，可以使用与存取局部变量同样的方法，因为这两者的构造原理几乎一模一样，所以，在子程序中有 invoke 语句时，如果要用到输入参数的地址当做 invoke的参数，同样要遵循局部变量的使用方式，不能用offset伪操作符，只能用addr来完成。同样，所有对局部变量使用的限制几乎都可以适用于参数。

## 3.5 高级语法

MASM中新引入了一系列的伪指令，涉及条件测试、分支和循环语句，利用它们，汇编语言有了与高级语言一样的结构，配合对局部变量和调用参数等高级语言中常见元素的支持，为使用Win32汇编编写大规模的应用程序奠定了基础。

### 3.5.1 条件测试语句

MASM条件测试的基本表达式是：

寄存器或变量 操作符 操作数

两个以上的表达式可以用逻辑运算符连接：

（表达式1）逻辑运算符（表达式2） 逻辑运算符（表达式3）…

允许的条件运算符为：>、<、>=、<=、==、!=。

允许的逻辑运算符为：&&、||、!。

其实，MASM 的条件测试采用的是和 C 语言相同的语法。

MASM的条件测试语句有几个限制，首先是表达式的左边只能是变量或寄存器，不能为常数；其次表达式的两边不能同时为变量，但可以同时是寄存器。这些限制来自于80x86的指令，因为条件测试伪操作符只是简单地把每个表达式翻译成 cmp 或 test 指令，80x86 的指令集中没有 cmp 0,eax 之类的指令，同时也不允许直接操作两个内存中的数，所以对这两个限制是很好理解的。

除了这些和高级语言类似的条件测试伪操作，汇编语言还有特殊的要求，就是程序中常常要根据系统标志寄存器中的各种标志位来做条件跳转，这些在高级语言中是用不到的，所以又增加了以下一些标志位的状态指示，它们本身相当于一个表达式：

CARRY? 表示Carry位是否置位

OVERFLOW? 表示Overflow位是否置位

PARITY? 表示Parity位是否置位

SIGN? 表示Sign位是否置位

ZERO? 表示Zero位是否置位

要测试eax等于ebx同时Zero位置位，条件表达式可以写为：

(eax==ebx) && ZERO?

要测试eax等于ebx同时Zero位清零，条件表达式可以写为：

(eax==ebx) && ! ZERO?

### 3.5.2 分支语句

分支语句用来根据条件表达式测试的真假执行不同的代码模块， MASM中的分支语句的语法如下：

.if 条件表达式1

表达式1为“真”时执行的指令

[.elseif 条件表达式2]

表达式2为“真”时执行的指令

[.elseif 条件表达式3]

表达式3为“真”时执行的指令

…

[.else]

所有表达式为“否”时执行的指令

.endif

注意：关键字 if/elseif/else/endif 的前面有个小数点，如果不加小数点，就变成宏汇编中的条件汇编伪操作了，结果可是天差地别。

### 3.5.3 循环语句

使用循环的语法是：

.while 条件测试表达式

指令

[.break [.if 退出条件]]

[.continue]

.endw

或

.repeat

指令

[.break [.if 退出条件]]

[.continue]

.until 条件测试表达式 （或.untilcxz [条件测试表达式]）

.while/.endw循环首先判断条件测试表达式，如果结果是“真” ，则执行循环体内的指令，结束后再回到 .while处判断表达式， 如此往复， 一直到表达式结果为 “假” 为止。 .while/.endw指令有可能一遍也不会执行到循环体内的指令，因为如果第一次判断表达式时就遇到结果为“假”的情况，那么就直接退出循环。

.repeat/.until循环首先执行一遍循环体内的指令，然后再判断条件测试表达式，如果结果为 “真” 的话， 就退出循环， 如果为 “假” ， 则返回 .repeat处继续循环， 可以看出， .repeat/.until不管表达式的值如何，至少会执行一遍循环体内的指令。

也可以把条件表达式直接设置为固定值，这样就可以构建一个无限循环，对于.while/.end直接使用TRUE，对 .repeat/.until直接使用FALSE来当表达式就是如此，这种情况下，可以使用 .break伪指令强制退出循环，如果 .break伪指令后面跟一个 .if测试伪指令的话，那么当退出条件为“真”时才执行 .break伪指令。

在循环体中也可以用 .continue 伪指令忽略以后的指令，遇到 .continue 伪指令时，不管下面还有没有其他循环体中的指令，都会直接回到循环头部开始执行。

.while/.endw和 .repeat/.until循环没有使用loop指令的优势，因为loop指令可以自动递减ecx的值来控制循环， 不使用loop将会在循环体内多设置一参数递减的指令，但不用 loop 指令的好处是带来更多的灵活性。为了弥补这个缺陷，可以使用 .repeat/.untilcxz伪指令， 编译器将会强制使用loop指令来完成循环， 当然， 在这种用法中，程序员必须在循环开始前正确设置ecx的值。

如果又想用loop指令来构成循环又要使用条件表达式怎么办，这时同样可以在 .untilcxz

伪指令后加条件测试语句，只不过这时候有很大的限制，第一只能是单个条件表达式，不能用&&或||来构成多项表达式了；第二即使是单个表达式中，也只能用==或!=操作符，不能用其他比较大小的操作符， 因为这时编译器的翻译方式是在一个比较指令后使用loopz或loopnz来构成循环，这个指令不能测试其他标志位。

|  |
| --- |
| warning.bmp在分支和循环的伪指令反汇编后可以发现，在使用>、>=、<和<=比较符时，MASM的伪指令总是将比较以后的跳转指令使用为 jb 和 jnb 等无符号数比较跳转的指令，这就意味着，MASM 的条件测试总是把操作数当做无符号数看待，这样，假设eax等于1，那么表达式（eax > -1）的值是“假”，因为-1表示为0ffffffffh，如果当做无符号数看，它反而是最大的数！如果程序中需要构造有符号数的比较分支或循环结构，那么必须另外用 jl 和 jg 等有符号数比较跳转的指令来完成，使用条件测试配合分支或循环伪指令可能会得到错误的结果！ |

## 3.6 代码风格

### 3.6.1 变量和函数的命名

**1. 匈牙利表示法**

在汇编语言中常用的类型前缀有：

b 表示byte

w 表示word

dw 表示dword

h 表示句柄

lp 表示指针

sz 表示以0结尾的字符串

lpsz 表示指向0结尾的字符串的指针

f 表示浮点数

st 表示一个数据结构

对于函数名，一般用“动词+名词”的形式。

**2. 对匈牙利表示法的补充**

使用匈牙利表示法已经基本上解决了命名的可读性问题，但相对于其他高级语言，汇编语言有语法上的特殊性，考虑下面这些汇编语言特有的问题：

● 对局部变量的地址引用要用lea指令或用addr伪操作，全局变量要用offset；对局部变量的使用要特别注意初始化问题。如何在定义中区分全局变量、局部变量和参数？

● 汇编的源代码占用的行数比较多，代码行数很容易膨胀，程序规模大了如何分清一个函数是系统的API还是本程序内部的子程序？

实际上上面的这些问题都可以归纳为区分作用域的问题。为了分清变量的作用域，命名中对全局变量、局部变量和参数应该有所区别，所以我们需要对匈牙利表示法做一些补充，以适应Win32汇编的特殊情况，下面的补充方法是笔者提出的，读者可以参考使用：

● 全局变量的定义使用标准的匈牙利表示法，在参数的前面加下划线，在局部变量的前面加@符号，这样引用的时候就能随时注意到变量的作用域。

● 在内部子程序的名称前面加下划线，以便和系统API区别。

### 3.6.2 代码的书写格式

**1. 排版方式**

程序的排版风格应该遵循以下规则。

首先是大小写的问题，汇编程序中对于指令和寄存器的书写是不分大小写的，但小写代码比大写代码便于阅读，所以程序中的指令和寄存器等最好采用小写字母，而用 equ 伪操作符定义的常量则使用大写，变量和标号使用匈牙利表示法，大小写混合。

其次是使用Tab的问题。汇编源程序中Tab的宽度一般设置为8个字符。在语法上，指令和操作数之间至少有一个空格就可以了，但指令的助记符长度是不等长的，用Tab隔开指令和操作数可以使格式对齐，便于阅读。如：

xor eax,eax

fistp dwNumber

xchg eax,ebx

上述代码的写法就不如下面的写法整齐：

xor eax,eax

fistp dwNumber

xchg eax,ebx

还有就是缩进格式的问题。程序中的各部分采用不同的缩进，一般变量和标号的定义不缩进，指令用两个Tab缩进，遇到分支或循环伪指令再缩进一格，如：

.data

dwFlag dd ?

.code

start:

mov eax,dwFlag

.if dwFlag == 1

call \_Function1

.else

call \_Function2

.endif

…

合适的缩进格式可以明显地表现出程序的流程结构，也很容易发现嵌套错误，当缩进过多的

时候，可以意识到嵌套过深，该改进程序结构了。

**2. 注释和空行**

没有注释的程序是很难维护的，但注释的方法也很有讲究，写注释要遵循以下的规则：

● 不要写无意义的注释，如“将1放到eax中”，“跳转到exit标号处”等。

● 修改代码同时修改相应的注释，以保证注释与代码的一致性。

● 注释以描写一组指令实现的功能为主，不要解释单个指令的用法，那是应该由指令手册来完成的，不要假设看程序的人连指令都不熟悉。

● 对于子程序，要在头部加注释说明参数和返回值，子程序可以实现的功能，以及调用时应该注意的事项。

由于汇编语言是以一条指令为一行的，实现一个小功能就需要好几行，没有分段的程序很难看出功能模块来，所以要合理利用空行来隔开不同的功能块，一般以在高级语言中可以用一句语句来完成的一段汇编指令为单位插入一个空行。

**3. 避免使用宏**

在 MASM 的宏功能中最好只使用条件汇编，用来选择编译不同的代码块来构建不同的版本，其他如宏定义和宏调用只会破坏程序的可读性，能够不用就尽量不用，虽然展开后只有一两句的宏定义不在此列，但既然展开后也只有一两句，那么和直接使用指令也就没有什么区别了。

在汇编中避免使用宏定义的理由是：汇编中随时要用到各个寄存器，宏定义不同于子程

序，可以有选择地通过push/pop指令保护现场，在使用中很容易忽略里面用了哪个寄存器，

从而对程序结构构成威胁。高级语言的宏定义则不会有这个问题。

### 3.6.3 代码的组织

程序中要注意变量的组织和模块的组织方式。

由于过多的全局变量会影响程序的模块化结构，所以不要设置没必要的全局变量，尽量把变量定义成局部变量。把仅在子程序中使用的变量设置为局部变量可以使子程序更容易封装成一个黑匣子，如果无法把全部变量设置为局部变量，则尽量把这些数据改为参数输入输出，如果无法改为参数，那么意味着这个子程序不能不经修改地直接放到别的程序中使用。

在主程序中使用比较频繁的部分，以及便于封装成黑匣子在别的程序上用的代码，都应该写成子程序，但一个子程序的规模不应该太大，行数尽量限制在几百行之内，功能则限于完成单个功能。对于子程序，定义参数的时候要尽可能精简，对可能引起程序崩溃的参数，如指针等，要进行合法性检测。

子程序中在使用完申请的资源的时候，注意在退出前要释放所用资源，包括申请的内存和其他句柄等，对于打开的文件则要关闭。

对于程序员来说，开发每一个软件都要从头做起是很浪费时间的，一般的做法是从自己以前做过的程序中拷贝相似的代码，但修改还是要花一定的时间，最好的办法就是尽量把子程序做成一个黑匣子，可以不经修改地直接拿过来用，这样，每次编程相当于只是编写新增的部分，随着代码的积累，开发任何程序都将是很快的事情。

# 第4章 第一个窗口程序

## 4.1 开始了解窗口

### 4.1.1 窗口是什么

**1. 使用窗口的原因**

Windows是多任务的操作系统，可以同时运行多个程序，同样，各个程序在屏幕上的显示不能互相干扰，而且，多个程序可以看成是“同时”运行的，在后台的程序也可能随时向屏幕输出内容，这中间的调度是由 Windows 完成的。Windows 采用的方法是给程序一块矩形的屏幕空间，这就是窗口。应用程序通过 Windows向属于自己的窗口显示信息，Windows判断该窗口是不是被别的窗口挡住，并把没有挡住的部分输出到屏幕上，这样屏幕上显示的东西就不会互相覆盖而乱套。对于应用程序来说，它只需认为窗口就是自己拥有的显示空间就可以了。

**2. 窗口和程序的关系**

二者没有一一对应关系。

任何GUI控件都可以看作是一个窗口，如状态栏、工具栏，对话框中的按钮、文本输入

框与复选框等都是窗口。

### 4.1.2 窗口界面



图4-1 一个典型的窗口

### 4.1.3 窗口程序是怎么工作的

**1. 窗口程序的运行模式**

对程序员来说，要了解的不仅是用户可以看到的部分，还必须了解隐藏在窗口底下的东西，了解用怎样的程序结构来实现窗口的行为模式。

DOS程序员熟悉的是顺序化的、按过程驱动的程序设计方法，这种程序有明显的开始、明显的过程和明显的结束，由程序运行的阶段来决定用户该做什么。

而窗口程序是事件驱动的，用户可能随时发出各种消息，如操作的过程中觉得窗口不够大了，就马上拖动边框，程序必须马上调整客户区的内容以适应新的窗口大小；用户觉得想先干别的事情，可能会把窗口最小化，关闭按钮也有可能随时被按下，这意味着程序要随时可以处理退出的请求。如果非要规定干活的时候不能移动窗口与调整大小，那么这些窗口就会呆在桌面上一动不动。

|  |
| --- |
| 灯泡.BMP再次提醒：这里是“窗口程序”而不是“Windows 程序”，因为和窗口有关的程序才是事件驱动的，其他的 Windows 可能并不这样工作，如控制台程序的结构还是同DOS 程序一样是顺序化的，但与窗口相关的 Windows 程序占了绝大多数，所以大部分书籍中讲到 Windows程序就认为是事件驱动的程序。 |

下面先来看一个地地道道的 Win32汇编窗口程序。

**2. HelloWorld源代码**

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; helloworld.asm

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 使用 nmake 或下列命令进行编译和链接:

; ml /c /coff FirstWindow.asm

; Link /subsystem:windows FirstWindow.obj

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.386

.model flat,stdcall

option casemap:none

include windows.inc

include gdi32.inc

includelib gdi32.lib

include user32.inc

includelib user32.lib

include kernel32.inc

includelib kernel32.lib

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 数据段

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.data?

hInstance dd ?

hwnd dd ?

.const

szClassName db 'HelloWorld',0

szCaptionMain db 'Assemble Version',0

szText db 'Hello,world !',0

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 代码段

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.code

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; This function is called by the Windows function DispatchMessage()

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\_WindowProcedure proc uses ebx edi esi,\_hWnd,\_uMsg,\_wParam,\_lParam

local @stPs:PAINTSTRUCT

local @stRect:RECT

local @hDc

;handle the msg

mov eax,\_uMsg

.if eax == WM\_PAINT

invoke BeginPaint,\_hWnd,addr @stPs ;begin painting

mov @hDc,eax

invoke GetClientRect,\_hWnd,addr @stRect;get the size of client area

;show "hello world"

invoke DrawText,@hDc,addr szText,-1,\

addr @stRect,\

DT\_SINGLELINE or DT\_CENTER or DT\_VCENTER

invoke EndPaint,\_hWnd,addr @stPs ;end painting

.elseif eax == WM\_DESTROY

invoke PostQuitMessage,0 ;send a WM\_QUIT to the message queue

.else ;for msg that we don't deal with

invoke DefWindowProc,\_hWnd,\_uMsg,\_wParam,\_lParam

ret

.endif

xor eax,eax

ret

\_WindowProcedure endp

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; main function

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\_WinMain proc

local @stWndClass:WNDCLASSEX

local @stMsg:MSG

invoke GetModuleHandle,NULL

mov hInstance,eax

;The Window structure

push hInstance

pop @stWndClass.hInstance

mov @stWndClass.lpszClassName,offset szClassName

mov @stWndClass.lpfnWndProc,offset \_WindowProcedure

mov @stWndClass.style,CS\_HREDRAW or CS\_VREDRAW

mov @stWndClass.cbSize,sizeof WNDCLASSEX

;Use default icon and mouse-pointer

invoke LoadIcon,NULL,IDI\_APPLICATION

mov @stWndClass.hIcon,eax

mov @stWndClass.hIconSm,NULL

invoke LoadCursor,NULL,IDC\_ARROW

mov @stWndClass.hCursor,eax

mov @stWndClass.lpszMenuName,NULL ;No menu

mov @stWndClass.cbClsExtra,0 ;No extra bytes after the window class

mov @stWndClass.cbWndExtra,0 ;structure or the window instance

;Use Windows's default colour as the background of the window

mov @stWndClass.hbrBackground,COLOR\_WINDOW + 1

;Register the window class, and if it fails quit the program

invoke RegisterClassEx,addr @stWndClass

.if eax == 0

ret

.endif

;The class is registered, let's create the window

invoke CreateWindowEx,WS\_EX\_LEFT,\

offset szClassName,\

offset szCaptionMain,\

WS\_OVERLAPPEDWINDOW,\

CW\_USEDEFAULT,\

CW\_USEDEFAULT,\

CW\_USEDEFAULT,\

CW\_USEDEFAULT,\

NULL,\

NULL,\

hInstance,\

NULL

mov hwnd,eax

;Make the window visible on the screen

invoke ShowWindow,hwnd,SW\_SHOWNORMAL

;update the window

invoke UpdateWindow,hwnd

;Run the message loop. It will run until GetMessage() returns 0

.while TRUE

invoke GetMessage,addr @stMsg,NULL,0,0

.break .if eax == 0

;Translate virtual-key msg into character msg

invoke TranslateMessage,addr @stMsg

;Send message to \_WindowProcedure

invoke DispatchMessage,addr @stMsg

.endw

;The program return-value is 0 - The value that PostQuitMessage() gave

mov eax,@stMsg.wParam

ret

\_WinMain endp

start:

call \_WinMain

invoke ExitProcess,0

end start

**3. 窗口程序的运行过程**

在屏幕上显示一个窗口的过程一般有以下步骤，这就是主程序的结构流程：

（1）得到应用程序的句柄（GetModuleHandle，高级语言没有这一步）。

（2）注册窗口类（RegisterClassEx）。在注册之前，要先填写 RegisterClassEx 的参数WNDCLASSEX结构。

（3）创建窗口（CreateWindowEx）。

（4）显示窗口（ShowWindow）。

（5）刷新窗口（UpdateWindow）。

（6）消息循环。首先获取消息（GetMessage），如果有消息到达，则将消息分派到回调函数处理（DispatchMessage），如果消息是 WM\_QUIT，则退出循环。

程序的另一半\_ProcWinMain 子程序是用来处理消息的，它就是窗口的回调函数（Callback） ，也叫做窗口过程，之所以是回调函数是因为它是由 Windows 而不是我们自己调用的， 我们调用 DispatchMessage， 而DispatchMessage在自己的内部回过来调用窗口过程。

所有的用户操作都是通过消息来传给应用程序的，如用户按键、鼠标移动、选择了菜单和拖动了窗口等，应用程序中由窗口过程接收消息并处理，在例子程序中就是\_ProcWinMain。由于窗口过程构造了一个分支结构，对应不同的消息执行不同的代码，所以一个应用程序中几乎所有的功能代码都集中在窗口过程里。

窗口程序运行中消息传输的流程可以由图 4-4 来表示。

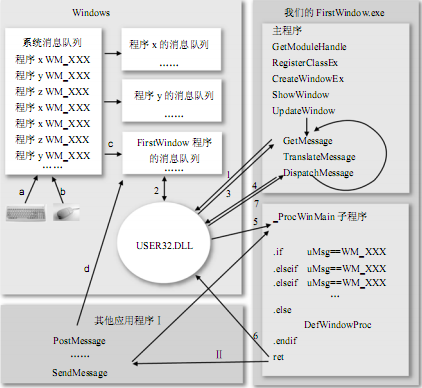


图4-4 窗口程序的运行过程

先来看看 Windows 对消息的处理。Windows 在系统内部有一个系统消息队列，当输入设备有所动作的时候，如用户按动了键盘、移动了鼠标、按下或放开了鼠标等，Windows都会产生相应的记录放在系统消息队列里，如图 4-4 中的箭头 a和b 所示，每个记录中包含消息的类型、发生的位置（如鼠标在什么坐标移动）和发生的时间等信息。

同时，Windows为每个程序（严格地说是每个线程）维护一个消息队列，Windows检查系统消息队列里消息的发生位置，当位置位于某个应用程序的窗口范围内的时候，就把这个消息派送到应用程序的消息队列里，如图 4-4 中的箭头 c所示。

当应用程序还没有来取消息的时候，消息就暂时保留在消息队列里，当程序中的消息循环执行到 GetMessage 的时候，控制权转移到 GetMessage所在的 USER32.DLL中（箭头 1） ，USER32.DLL从程序消息队列中取出一条消息（箭头 2） ，然后把这条消息返回应用程序（箭头 3） 。

应用程序可以对这条消息进行预处理，如可以用 TranslateMessage 把基于键盘扫描码的按键消息转换成基于 ASCII码的键盘消息， 以后也会用到 TranslateAccelerator把键盘快捷键转换成命令消息，但这个步骤不是必需的。

然后应用程序将处理这条消息，但方法不是自己直接调用窗口过程来完成，而是通过 DispatchMessage 间接调用窗口过程，Dispatch 的英文含义是“分派” ，之所以是“分派” ，是因为一个程序可能建有不止一个窗口，不同的窗口消息必须分派给相应的窗口过程。当控制权转移到 USER32.DLL 中的 DispatchMessage 时，DispatchMessage 找出消息对应窗口的窗口过程，然后把消息的具体信息当做参数来调用它（箭头 5） ，窗口过程根据消息找到对应的分支去处理，然后返回（箭头 6） ，这时控制权回到 DispatchMessage，最后 DispatchMessage 函数返回应用程序（箭头 7） 。这样，一个循环就结束了，程序又开始新一轮的 GetMessage。

|  |
| --- |
| 灯泡.BMP有个很常见的问题：为什么要由 Windows 来调用窗口过程，程序取了消息以后自己处理不是更简便吗？事实上并非如此，如果程序自己处理消息的“分派”，就必须自己维护本程序所属窗口的列表，当程序建立的窗口不止一个的时候，这个工作就变得复杂起来；另一个原因是：别的程序也可能用 SendMessage 通过 Windows 直接调用你的窗口过程；第三个原因：Windows并不是把所有的消息都放进消息队列，有的消息是直接调用窗口过程处理的，如 WM\_SETCURSOR 等实时性很强的消息，所以窗口过程必须开放给 Windows。 |

应用程序之间也可以互发消息， PostMessage 是把一个消息放到其他程序的消息队列中，如图 4-4 中箭头 d 所示，目标程序收到了这条消息就把它放入该程序的消息队列去处理；而SendMessage则越过消息队列直接调用目标程序的窗口过程（如图 4-4 中箭头 I所示） ，窗口过程返回以后才从 SendMessage 返回（如图 4-4 中箭头 II所示） 。 窗口过程是由 Windows 回调的，Windows 又是怎么知道往哪里回调呢？答案是我们在调用 RegisterClassEx 函数的时候已经把窗口过程的地址告诉了 Windows。

## 4.2 分析窗口程序

### 4.2.1 模块和句柄

**1. 模块的概念**

一个模块代表的是一个运行中的 exe 文件或 dll 文件，用来代表这个文件中所有的代码和资源，磁盘上的文件不是模块，装入内存后运行时就叫做模块。一个应用程序调用其他DLL中的 API时，这些 DLL文件被装入内存，就产生了不同的模块，为了区分地址空间中的不同模块，每个模块都有一个惟一的模块句柄来标识。

由于很多 API 函数中都要用到程序的模块句柄，以便利用程序中的各种资源，所以在程序的一开始就先取得模块句柄并存放到一个全局变量中可以省去很多的麻烦，在 Win32中，模块句柄在数值上等于程序在内存中装入的起始地址。 取模块句柄使用的 API函数是 GetModuleHandle，它的使用方法是：

invoke GetModuleHandle,lpModuleName

lpModuleName 参数是一个指向含有模块名称字符串的指针，可以用这个函数取得程序地址空间中各个模块的句柄，例如，如果想得到 User32.dll的句柄以便使用其中包含的图标资源，那么可以如下使用：

szUserDll db 'User32.dll',0

…

invoke GetModuleHandle,addr szUserDll

.if eax

mov hUserDllHandle,eax

.endif

…

如果使用参数 NULL调用 GetModuleHandle，那么得到的是调用者本模块的句柄，我们的源程序中就是这样使用的：

invoke GetModuleHandle,NULL

mov hInstance,eax

可以注意到， 把返回的句柄放到了取名为 hInstance的变量里而并不是放在 hModule中，为什么是 hInstance 呢? Instance 是“实例” ，它的概念来自于 Win16，Win16 中不同运行程序的地址空间并非是完全隔离的，一个可执行文件运行后形成“模块” ，多次加载同一个可执行文件时，这个“模块”是公用的，为了区分多次加载的“拷贝” ，就把每个“拷贝”叫做实例，每个实例均用不同的“实例句柄” （hInstance）值来标识它们。

但在 Win32中，程序运行时是隔离的，每个实例都使用自己私有的 4 GB空间，都认为自己是惟一的，不存在一个模块的多个实例的问题，实际上在 Win32 中，实例句柄就是模块句柄，但很多 API 原型中用到模块句柄的时候使用的名称还是沿用 hInstance，所以我们还是把变量名称取为 hInstance。

|  |
| --- |
| 灯泡.BMP在 C语言的编程中，hInstance通过 WinMain由系统传入，WinMain 的原型是： WinMain（hInstance,hPrevInstance,lpzCmdParam,nCmdShow），程序不用自己去获得hInstance，这个过程由 C 的初始化代码代劳了，但在 Win32 汇编中 hInstance 必须自己获取，如果不了解 hModule 就是 hInstance 的话，就无法得知如何得到 hInstance，因为并没有一个类似于 GetInstanceHandle之类的 API函数。 |

**2. 句柄是什么**

随着分析的深入，句柄（handle）一词也出现得频繁起来， “句柄”是什么呢？句柄只是一个数值而已，它的值对程序来说是没有意义的，它只是 Windows 用来表示各种资源的编号而已，可见只有 Windows才知道怎么使用它来引用各种资源。

下面举例说明。屏幕上已经有 10 个窗口，Windows把它们从 1 到10 编号，应用程序又建立了一个窗口，现在 Windows把它编号为 11，然后把 11当做窗口句柄返回给应用程序，应用程序并不知道 11 代表的是什么，但在操作窗口的时候，把 11 当做句柄传给 Windows，Windows自然可以根据这个数值查出是哪个窗口。当该窗口关闭的时候，11 这个编号作废。第二次运行的时候，如果屏幕上现有 5 个窗口，那么现在句柄可能就是 6 了，所以，应用程序并不用关心句柄的具体数值是多少。打个比方，可以把句柄当做是商场中寄放书包时营业员给的纸条，纸条上的标记用户并不知道是什么意思，但把它交还给营业员的时候，她自然会找到正确的书包。

Windows中几乎所有的东西都是用句柄来标识的，文件句柄、窗口句柄、线程句柄和模

块句柄等，同样道理，不必关心它们的值究竟是多少，拿来用就是了！

### 4.2.2 创建窗口

在创建窗口之前，先要谈到“类” 。 “类”的概念读者都不陌生，主要是为了把一组物体的相同属性归纳整理起来封装在一起，以便重复使用，在“类”已定义的属性基础上加上其他个性化的属性，就形成了各式各样的个体。

Windows中创建窗口同样使用这样的层次结构。首先定义一个窗口类，然后在窗口类的基础上添加其他的属性建立窗口。 不用一步到位的办法是因为很多窗口的基本属性和行都是一样的，如按钮、文本输入框和选择框等，对这些东西 Windows 都预定义了对应的类，使用时直接使用对应的类名建立窗口就可以了。 只有用户自定义的窗口才需要先定义自己的类，再建立窗口。这样可以节省资源。

**1. 注册窗口类**

建立窗口类的方法是在系统中注册，注册窗口类的 API函数是 RegisterClassEx，最后的“Ex”是扩展的意思，因为它是 Win16 中 RegisterClass 的扩展。一个窗口类定义了窗口的一些主要属性，如：图标、光标、背景色、菜单和负责处理该窗口所属消息的函数。这些属性并不是分成多个参数传递过去的，而是定义在一个 WNDCLASSEX 结构中，再把结构的地址当参数一次性传递给 RegisterClassEx，WNDCLASSEX 是 WNDCLASS结构的扩展。

WNDCLASSEX的结构定义为：

WNDCLASSEX STRUCT

cbSize DWORD ? ;结构的字节数

style DWORD ? ;类风格

lpfnWndProc DWORD ? ;窗口过程的地址

cbClsExtra DWORD ?

cbWndExtra DWORD ?

hInstance DWORD ? ;所属的实例句柄

hIcon DWORD ? ;窗口图标

hCursor DWORD ? ;窗口光标

hbrBackground DWORD ? ;背景色

lpszMenuName DWORD ? ;窗口菜单

lpszClassName DWORD ? ;类名字符串的地址

hIconSm DWORD ? ;小图标

WNDCLASSEX ENDS

在 FirstWindow程序中，注册窗口类的代码是：

local @stWndClass:WNDCLASSEX ；定义一个WNDCLASSEX结构

…

;The Window structure

push hInstance

pop @stWndClass.hInstance

mov @stWndClass.lpszClassName,offset szClassName

mov @stWndClass.lpfnWndProc,offset \_WindowProcedure

mov @stWndClass.style,CS\_HREDRAW or CS\_VREDRAW

mov @stWndClass.cbSize,sizeof WNDCLASSEX

;Use default icon and mouse-pointer

invoke LoadIcon,NULL,IDI\_APPLICATION

mov @stWndClass.hIcon,eax

mov @stWndClass.hIconSm,eax

invoke LoadCursor,NULL,IDC\_ARROW

mov @stWndClass.hCursor,eax

mov @stWndClass.lpszMenuName,NULL ;No menu

mov @stWndClass.cbClsExtra,0 ;No extra bytes after the window class

mov @stWndClass.cbWndExtra,0 ;structure or the window instance

;Use Windows's default colour as the background of the window

mov @stWndClass.hbrBackground,COLOR\_WINDOWFRAME

;Register the window class, and if it fails quit the program

invoke RegisterClassEx,addr @stWndClass

.if eax == 0

ret

.endif

程序定义了一个 WNDCLASSEX结构的变量@stWndClass，结构各字段的含义如下：

● hIcon——图标句柄，指定显示在窗口标题栏左上角的图标。Windows 已经预定义了一些图标， 同样， 程序也可以使用在资源文件中定义的图标， 这些图标的句柄可以用LoadIcon函数获得。如果这个成员为NULL，系统会提供默认的图标。

● hIconSm——图标句柄，指定显示在任务栏上的缩略图标。如果该成员为NULL，则系统会将hIcon指定的图标缩小到合适的大小，作为hIconSm指向的图标。

● hCursor——光标句柄，指定了鼠标在窗口中的光标形状。同样，Windows 也预定义了一些光标，可以用 LoadCursor获取它们的句柄，IDC\_ARROW是 Windows预定义的箭头光标，如果想使用自定义的光标，也可以自己在资源文件中定义。

● lpszMenuName——指定窗口上显示的默认菜单，它指向一个字符串，描述资源文件中菜单的名称，如果资源文件中菜单是用数值定义的，那么这里使用菜单资源的数值。窗口中的菜单也可以在建立窗口函数 CreateWindowEx的参数中指定。如果在两个地方都没有指定，那么建立的窗口上就没有菜单。

● hInstance——指定要注册的窗口类属于哪个模块，模块句柄在程序开始的地方已经用GetModuleHandle 函数获得。

● cbSize——指定WNDCLASSEX结构的长度， 用sizeof伪操作来获取。 很多Win32 API参数中的结构都有 cbSize字段，它主要是用来区分结构的版本，当以后新增了一个字段时，cbSize 就相应增大，如果调用的时候 cbSize 还是旧的长度，表示运行的是基于旧结构的程序，这样可以防止使用无效的字段。

● style——窗口风格。CS\_HREDRAW 和 CS\_VREDRAW 表示窗口的宽度或高度改变时是否重画窗口。比较重要的是 CS\_DBLCLKS风格，指定了它，Windows才会把在窗口中快速两次单击鼠标的行为翻译成双击消息 WM\_LBUTTONDBLCLK 发给窗口过程。笔者就曾经忘了指定它，结果怎么也搞不出双击消息来。 Windows已经有一些预定义的值，它们是以 CS（Class Style的缩写）开始的标识符。

● hbrBackground——窗口客户区的背景色。前面的 hbr 表示它是一个刷子（Brush）的句柄，“刷子”一词形象地表示了填充一个区域的着色模式。Windows预定义了一些刷子，如 BLACK\_BRUSH和 WHITE\_BRUSH等，可以用下列语句来得到它们的句柄：

invoke GetStockObject, WHITE\_BRUSH

但在这里也可以使用颜色值，Windows已经预定义了一些颜色值， 分别对应窗口各部分的颜色，如 COLOR\_BACKGROUND，COLOR\_HIGHLIGHT，COLOR\_MENU 和 COLOR\_WINDOW等，使用颜色值的时候，Windows规定必须在颜色值上加1，所以程序中的指令是：

mov @stWndClass.hbrBackground,COLOR\_WINDOW + 1

● lpszClassName——指定程序员要建立的类命名，以便以后用这个名称来引用它。这个字段是一个字符串指针，在程序里，它指向“HelloWorld”字符串。

● cbWndExtra 和 cbClsExtra——分别是在 Windows 内部保存的窗口结构和类结构中给程序员预留的空间大小， 用来存放自定义数据， 它们的单位是字节。 不使用自定义数据的话，这两个字段就是 0。

● lpfnWndProc——最重要的参数，它指定了基于这个类建立的窗口的窗口过程地址。通过这个参数，Windows就知道了在 DispatchMessage 函数中把窗口消息发到哪里去，一个窗口过程可以为多个窗口服务，只要这些窗口是基于同一个窗口类建立的。Windows 中不同应用程序中的按钮和文本框的行为都是一样的，就是因为它们是基于相同的 Windows 预定义类建立的，它们背后的窗口过程其实是同一段代码。

**2. 建立窗口**

接下来的步骤是在已经注册的窗口类的基础上建立窗口，使用“类”的原因是定义窗口的“共性” ，建立窗口时肯定还要指定窗口的很多“个性化”的参数——如 WNDCLASSEX结构中没有定义的外观、标题、位置、大小和边框类型等属性，这些属性是在建立窗口时才指定的。

与注册窗口类时用一个结构传递所有参数不同， 建立窗口时所有的属性都是用单个参数的方式传递的，建立窗口的函数是 CreateWindowEx（注意不要写成 CreateWindowsEx） ，同样，它是 Win16中 CreateWindow函数的扩展，主要表现在多了一个 dwExStyle（扩展风格）参数，原因是 Win32比Win16 中多了很多种窗口风格，原来的一个风格参数已经不够用了。CreateWindowEx函数的使用方法是：

invoke CreateWindowEx,dwExStyle,lpClassName,lpWindowName,dwStyle,\

x,y,nWidth,nHeight,hWndParent,hMenu,hInstance,lpParam

虽然这个函数的参数多达 12 个，但它们很好理解：

● lpClassName—建立窗口使用的类名字符串指针，在 HelloWorld.asm 程序中指向“HelloWord”字符串，表示使用“HelloWorld”类建立窗口，这正是我们自己注册的类，这样一来，这个窗口就有“HelloWorld”的所有属性，并且消息将被发到“MyClass”中指定的窗口过程中去，当然，这里也可以是 Windows预定义的类名，如编辑框就是“EDIT”。

● lpWindowName——指向表示窗口名称的字符串，该名称会显示在标题栏上。如果该参数空白，则标题栏上什么都没有。

● hMenu——窗口上要出现的菜单的句柄。在注册窗口类的时候也定义了一个菜单，那是窗口的默认菜单，意思是如果这里没有定义菜单（用参数 NULL）而注册窗口类时定义了菜单，则使用窗口类中定义的菜单；如果这里指定了菜单句柄，则不管窗口类中有没有定义都将使用这里定义的菜单；两个地方都没有定义菜单句柄，则窗口上没有菜单。另外，当建立的窗口是子窗口时 （dwStyle中指定了 WS\_CHILD） ， 这个参数是另一个含义， 这时 hMenu参数指定的是子窗口的 ID号（这样可以节省一个参数的位置，因为子窗口不会有菜单）。

● lpParam——这是一个指针，指向一个欲传给窗口的参数，这个参数在 WM\_CREATE消息中可以被获取，一般情况下用不到这个字段。

● hInstance——模块句柄，和注册窗口类时一样，指定了窗口所属的程序模块。

● hWndParent——窗口所属的父窗口，对于普通窗口（相对于子窗口），这里的“父子”关系只是从属关系，主要用来在父窗口销毁时一同将其“子”窗口销毁，并不会把窗口位置限制在父窗口的客户区范围内，但如果要建立的是真正的子窗口（dwStyle 中指定了WS\_CHILD 的时候），这时窗口位置会被限制在父窗口的客户区范围内，同时窗口的坐标（x，y）也是以父窗口的左上角为基准的。

● x，y——指定窗口左上角位置，单位是像素。默认时可指定为 CW\_USEDEFAULT，这样 Windows 会自动为窗口指定最合适的位置，当建立子窗口时，位置是以父窗口的左上角为基准的，否则，以屏幕左上角为基准。

● nWidth，nHeight—窗口的宽度和高度，也就是窗口的大小，同样是以像素为单位的。默认时可指定为 CW\_USEDEFAULT，这样 Windows 会自动为窗口指定最合适的大小。

窗口的两个参数 dwStyle 和 dwExStyle 决定了窗口的外形和行为，dwStyle 是从 Win16开始就有的属性，， 它们是一些以 WS （Windows Style的缩写）为开头的预定义值。

dwExStyle是 Win32 中扩展的，它们是一些以 WS\_EX\_开头的预定义值，主要定义了一

些特殊的风格。

用预定义的组合值 WS\_EX\_PALETTEWINDOW 可以很方便地构成浮在其他窗口前面

的工具栏。

下面看几种不同的窗口外形， 如图 4-5 所示， 窗口1 是 WS\_OVERLAPPED 类型的窗口，只有一个边框，没有控制按钮和图标；窗口 2 同时指定 WS\_MAXIMIZEBOX，WS\_MINIMIZEBOX 和 WS\_SYSMENU，在窗口 1 的基础上多了控制按钮和图标；窗口 3是 WS\_POPUPWINDOW 风格的，这是一个没有标题和控制按钮的弹出式窗口，常见的软件装入时的版权窗口就是这种风格；前面 3 个窗口都不能通过拖动边框改变大小，而窗口 4指定了 WS\_THICKFRAME风格，可以改变大小，它的边框显得厚了一点；窗口 5的风格是WS\_OVERLAPPEDWINDOW，是最常见的属性组合；窗口 6 在窗口 5 的基础上指定了WS\_EX\_CLIENTEDGE，它的客户区显得有立体感；窗口 7 是个工具栏，指定的是WS\_EX\_TOOLWINDOW风格， 可以看到它的标题栏要小得多； 窗口8指定了WS\_HSCROLL和 WS\_VSCROLL风格，窗口中多了垂直和水平滚动条。



图4-5 不同风格的窗口

HelloWorld.asm程序中建立窗口的相关代码是这样的：

invoke CreateWindowEx,WS\_EX\_CLIENTEDGE,\

offset szClassName,offset szCaptionMain,\

WS\_OVERLAPPEDWINDOW,\

100,100,600,400,\

NULL,NULL,hInstance,NULL

mov hWinMain,eax

invoke ShowWindow,hWinMain,SW\_SHOWNORMAL

invoke UpdateWindow,hWinMain

…

建立窗口以后，eax 中传回来的是窗口句柄，要把它保存起来以备后用，这时候，窗口虽已建立，但还没有在屏幕上显示出来，要用 ShowWindow 把它显示出来，ShowWindow也可以用在别的地方，主要用来控制窗口的显示状态（显示或隐藏） ，大小控制（最大化、最小化或原始大小）和是否激活（当前窗口还是背后的窗口） ，它用窗口句柄做第一个参数，第二个参数则是显示的方式。

窗口显示以后，用 UpdateWindow 绘制客户区，它实际上就是向窗口发送了一条WM\_PAINT消息。到此为止，一个顶层窗口就正常建立并显示了。

CreateWindowEx 也可以用来建立子窗口，Windows 中有很多预定义的子窗口类，如按钮和文本框的类名分别是“Button”和“Edit” 。要建立一个按钮，只要把 lpClassName 指向 “Button”字符串就可以了。下面举例说明建立一个按钮的方法，代码如下：

.data

szButton db 'button',0

szButtonText db '&OK',0

…

invoke CreateWindowEx,NULL,\

offset szButton,offset szButtonText,\

WS\_CHILD or WS\_VISIBLE,\

10,10,65,22,\

hWnd,1,hInstance,NULL

在 helloworld 的源程序中加入按钮类的定义字符串 szButton 和按钮文字字符串szButtonText，然后在窗口过程的 WM\_CREATE消息中加入建立按钮的代码，执行一下，窗口中就出现了一个按钮。建立按钮的时候，lpWindowName参数就是按钮上的文字，风格则一定要指定 WS\_CHILD，建立的按钮才会在我们的主窗口上，WS\_VISIBLE也要同时指定，否则按钮不会显示出来，hMenu 参数在这里用做表示子窗口 ID，将它设置为 1，在建立多个子窗口的时候，ID 应该有所区别。

### 4.2.3 消息循环

**1. 消息循环的一般形式**

程序中的以下代码就是通常的消息循环：

.while TRUE

invoke GetMessage,addr @stMsg,NULL,0,0

.break .if eax == 0

invoke TranslateMessage,addr @stMsg

invoke DispatchMessage,addr @stMsg

.endw

消息循环中的几个函数要用到一个 MSG结构，用来做消息传递：

MSG STRUCT

hwnd DWORD ?

message DWORD ?

wParam DWORD ?

lParam DWORD ?

time DWORD ?

pt POINT <>

MSG ENDS

它的各个字段的含义是：

● hwnd——消息要发向的窗口句柄。

● message——消息标识符， 在头文件中以WM\_开头的预定义值 （意思为Windows Message） 。

● wParam——消息的参数之一。

● lParam——消息的参数之二。

● time——消息放入消息队列的时间。

● pt——这是一个 POINT数据结构，表示消息放入消息队列时的鼠标坐标。

这个结构定义了消息的所有属性，GetMessage 函数就是从消息队列中取出这样一条消息的：

invoke GetMessage,lpMsg,hWnd,wMsgFilterMin,wMsgFilterMax

函数的 lpMsg 指向一个 MSG 结构，函数会在这里返回取到的消息，hWnd 参数指定要获取哪个窗口的消息，例子中指定为 NULL，表示获取的是所有本程序所属窗口的消息，wMsgFilterMin 和 wMsgFilterMax 为0表示获取所有编号的消息。

GetMessage函数从消息队列里取得消息，填写好 MSG结构并返回，如果获取的消息是WM\_QUIT消息，那么 eax 中的返回值是 0，否则 eax 返回非零值，所以用 .break .if eax == 0 来检查返回值，如果消息队列中有 WM\_QUIT则退出消息循环。

TranslateMessage 将 MSG 结构传给 Windows 进行一些键盘消息的转换，当有键盘按下和放开时，Windows 产生 WM\_KEYDOWN 和 WM\_KEYUP 或 WM\_SYSKEYDOWN 和WM\_SYSKEYUP消息，但这些消息的参数中包含的是按键的扫描码，转换成常用的 ASCII码要经过查表，很不方便，TranslateMessage 遇到键盘消息则将扫描码转换成 ASCII 码并在消息队列中插入 WM\_CHAR 或 WM\_SYSCHAR 消息，参数就是转换好的 ASCII 码，如此一来，要处理键盘消息的话只要处理 WM\_CHAR 消息就好了。遇到非键盘消息则TranslateMessage不做处理。

最后，由 DispatchMessage 将消息发送到窗口对应的窗口过程去处理。窗口过程返回后DispatchMessage函数才返回，然后开始新一轮消息循环。

**2. 其他形式的消息循环**

GetMessage函数是程序空闲的时候主动将控制权交还给Windows的一种方式， Windows是一个抢占式的多任务系统，任务之间每 20 ms切换一次，试想一下，如果窗口程序在主窗口中采用死循环等待，消息由 Windows直接发送到窗口过程，那么程序会是下列这种样子：

invoke CreateWindow,…

invoke ShowWindow,…

invoke UpdateWindow,…

.while dwQuitFlag == 0 ;要退出时在窗口过程中设置dwQuitFlag

.endw

invoke ExitProcess,…

但这样一来，即使程序在空闲状态，轮到自己的 20 ms 时间片的时候，CPU 时间就会全部消耗在 .while循环中，使用 GetMessage的时候，轮到应用程序时间片的时候，如果消息队列里还没有消息，那么程序还是停留在 GetMessage 内部，这时就可以由 Windows当家做主没收这 20 ms的时间片，这样保证了 CPU资源的合理应用。

如果应用程序想把所有时间充分利用回来，消息队列里没有消息的时候不让GetMessage 在 Windows 内部等待，拱手交出属于自己的 CPU 时间，那么消息循环可以是下列这种样子：

.while TRUE

invoke PeekMessage,addr @stMsg,NULL,0,0,PM\_REMOVE

.if eax

.break .if @stMsg.message == WM\_QUIT

invoke TranslateMessage,addr @stMsg

invoke DispatchMessage,addr @stMsg

.else

<做其他工作>

.endif

.endw

PeekMessage是一个类似于 GetMessage 的函数，区别在于当消息队列里有消息的时候，PeekMessage 取回消息，并在 eax 中返回非零值，没有消息的时候它会直接返回，并在 eax中返回零。 所以在返回非零值的时候， 程序检查消息是否是 WM\_QUIT， 是则结束消息循环，不是则用标准流程处理消息；返回零的时候，表示是空闲时间，程序就可以做其他工作了，但插入做其他工作的代码执行时间不能过长，以不超过 10 ms为好，否则会影响正常的消息处理，使窗口的反应看起来很迟钝。如果必须处理很长时间的工作，那么应该将它分成很多小部分处理，以便有足够的频率用 PeekMessage来检查消息。

PeekMessage 的前面 4 个参数和 GetMessage 是相同的，增加的最后一个参数表示在取回消息以后，对消息队列中的消息是否保留。当这个参数是 PM\_REMOVE时，消息被取回的同时也被从消息队列里删除，而用 PM\_NOREMOVE 的时候，被取回的消息不会从消息队列中删除，函数相当于“偷看”了这条消息。例子程序中用了 PM\_REMOVE，否则每次看到的都是队列中的第一条消息。

### 4.2.4 窗口过程

窗口过程是给 Windows 回调用的，它必须遵循规定的格式。对窗口过程的子程序名并没有规定，对 Windows 来说，窗口过程的地址才是惟一需要的，例子程序中的子程序名是\_ProcWinMain，读者可以改用任何名称。窗口过程子程序的参数格式为：

WindowProc proc hwnd,uMsg,wParam,lParam

第一个参数是窗口句柄，由于一个窗口过程可能为多个基于同一个窗口类的窗口服务，所以 Windows回调的时候必须指出要操作的窗口，否则窗口过程不知道要去处理哪个窗口，FirstWindow程序只建立了一个窗口， 所以每次传递过来的 hwnd 和用 CreateWindowEx函数返回的窗口句柄是一样的；第二个参数是消息标识，后面两个参数是消息的两个参数。这 4个参数和消息循环中 MSG结构中的前 4 个字段是一样的。

**1. 窗口过程的结构**

窗口过程一般有如下的结构：

WindowProc proc uses ebx edi esi hWnd,uMsg,wParam,lParam

mov eax,uMsg

.if eax == WM\_XXX

<处理WM\_XXX 消息>

.elseif eax == WM\_YYY

<处理WM\_YYY 消息>

.elseif eax == WM\_CLOSE

invoke DestroyWindow,hWinMain

invoke PostQuitMessage,NULL

.else

invoke DefWindowProc,hWnd,uMsg,wParam,lParam

ret

.endif

xor eax,eax

ret

WindowProc endp

该过程主要是对 uMsg 参数中的消息编号构成一个分支结构，对于需要处理的消息分别处理。不感兴趣的消息则交给 DefWindowProc来处理。

要注意的是，窗口过程中要注意保存 ebx，edi，esi 和 ebp 寄存器，高级程序中不用自己操心这一点，汇编中就要注意了，Windows内部将这 4 个寄存器当指针使用，如果返回时改变了它们的值，程序会马上崩溃。proc 后面的 uses 伪操作在子程序进入和退出时自动安插上 push 和 pop 寄存器指令，来保护这些寄存器的值。其实不仅是在窗口过程中是这样，所有由应用程序提供给 Windows 的回调函数都必须遵循这个规定，如定时器回调函数等，所有 Win32 API 也遵循这个规定，所以调用 API 后，ebx，edi，esi 和 ebp 寄存器的值总是不会被改变的，但 ecx和 edx 的值就不一定了。

uMsg 参数指定的消息有一定的范围，Windows 标准窗口中已经预定义的值在 0～03ffh 之间，用户可以自定义一些消息，通过 SendMessage 等函数传给窗口过程做自定义的处理工作， 这时可以使用的值是从 0400h 开始的， WM\_USER 就定义为 00000400h，当程序员定义多个用户消息的时候，一般使用 WM\_USER+1，WM\_USER+2，…之类的定义方法。

wParam 和 lParam 参数是消息所附带的参数，它随消息的不同而不同，对于不同的消息，它们的含义必须分别从手册中查明：如 WM\_MOUSEMOVE 消息中，wParam 是标志，lParam 是鼠标位置；而在 WM\_GETTEXT 消息中，wParam 是要获取的字符数，lParam 是缓冲区地址；而对于 WM\_COPY 消息来说，它不需要额外的信息，所以两个参数都没有定义。

处理了不同的消息，必须返回规定的值给 Windows，返回值也需要分别从手册中查明，比如，处理 WM\_CREATE消息的时候，返回 0表示成功；如果程序无法初始化，如申请内存失败，那么可以返回－1，Windows 就不会继续窗口的创建过程。一些消息的返回值则没有定义，但大部分的消息处理以后都以返回 0 表示成功，所以程序中把默认的返回语句放在最后，将 eax清 0 后返回，如果在处理某个消息的时候需要返回不同的值，可以在分支中将eax 赋值后直接用 ret 指令返回。对于 DefWindowProc 的返回值，我们不对它进行干涉，所以直接将 eax 不做修改地用 ret 返回。

WM\_CLOSE消息是按下了窗口右上角的“关闭”按钮后收到的，程序可以在这里处理和关闭窗口相关的事情，一般是相关资源的释放工作，如释放内存、保存工作和提示用户是否保存工作等，如记事本程序在未保存的时候单击“关闭”按钮，会有提示框提示是否先保存文件，单击“取消”按钮的话，记事本不会关闭，这个步骤就是在 WM\_CLOSE消息处理中完成的。如果处理 WM\_CLOSE消息时直接返回，那么窗口不会关闭，因为这个消息只是Windows通知窗口用户单击了“关闭”按钮而已，窗口采取什么样的行为是窗口的事。当窗口决定关闭的时候，需要程序自己调用 DestroyWindow 来摧毁窗口，并用 PostQuitMessage向消息循环发送WM\_QUIT消息来退出消息循环。 调用PostQuitMessage时的参数是退出码，就是 GetMessage收到 WM\_QUIT后 MSG结构 wParam字段中的东西，在这里使用 0。

|  |
| --- |
| 灯泡.BMPPostQuitMessage 是初学者容易遗漏的函数，如果没有这条语句，外观上窗口是被摧毁掉，从屏幕上消失了，但主程序中的消息循环却没有收到 WM\_QUIT，结果还在那里打转。常有人调试的时候丢了这条语句，结果再一次编译的时候就收到错误：LINK fatal error LNK1104: cannot open file "xxx.exe"，这就表示exe文件仍然被使用中。  Windows为什么不在窗口摧毁的时候自动发送一个WM\_QUIT消息， 而必须由用户程序自己通过PostQuitMessage 函数发送呢？其实很好理解：因为屏幕上可能不止一个窗口，Windows无法确定哪个窗口关闭代表着程序结束。试想一下，用户打开了一个输入参数的小窗口，单击“确定”按钮后关闭并回到主窗口， Windows却不分三七二十一自动发送了一个WM\_QUIT， 程序就会莫名其妙地退出了。 |

**2. 收到消息的顺序**

窗口过程收到消息是有一定顺序的，收到第一条消息并不是从消息循环开始以后，而是在 CreateWindowEx中就开始了，显示和刷新窗口的函数 ShowWindow和UpdateWindow也向窗口过程发送消息，这一点并不奇怪，因为 Windows 在 CreateWindowEx 前调用RegisterClassEx的时候就已经得到窗口过程的地址了。 并且在建立窗口的过程中需要窗口过程的配合。

在所有这些阶段的消息中，大部分的消息都不需要程序自己关心，Windows只是尽义务通知窗口过程而已，窗口过程转手就交给 DefWindowProc 去处理了。程序需要关心的消息有下面这些，可以根据需要选择使用：

● WM\_CREATE——放置窗口初始化代码，如建立各种子窗口（状态栏和工具栏等）。

● WM\_SIZE——放置位置安排的代码，因为建立的子窗口可能需要随窗口大小的改变而移动位置。

● WM\_PAINT——如果需要自己绘制客户区，则在这里安排代码。

● WM\_CLOSE——向用户确认是否退出， 如果退出则摧毁窗口并发送WM\_QUIT消息。

● WM\_DESTROY——窗口摧毁，在这里放置释放资源等扫尾代码。

在例子程序中，我们处理了 WM\_PAINT消息来绘制客户区，功能就是在窗口的中间写上一行字： “Win32 Assembly, Simple and powerful !” 。窗口过程先通过 BeginPaint 获取窗口客户区的“设备环境”句柄，然后通过 GetClientRect获取客户区的大小，最后通过 DrawText函数将字符串按照取得的屏幕大小居中写到“设备环境”中，也就是窗口上。如果不需要显示这个字符串，则连 WM\_PAINT消息也不用处理。

**3. 消息的默认处理——DefWindowProc**

Windows预定义的消息范围是 0～03ffh，共预留了 1 024 个消息编号，查看一下头文件Windows.inc，可以发现实际已定义的消息数目有几百个，这些消息中的大部分对于窗口的运行来说都是必需的，如果窗口过程要处理每一种消息，那么窗口过程中的 elseif语句就会绵延数千行，但是窗口的行为就是由处理这些消息的方法来表现的，不处理又不行，怎么办呢？

实际上，大部分窗口的行为都是差不多的，这意味着如果要窗口过程处理全部的消息，不同窗口的窗口过程代码应该是大同小异的， 完全可以用一个通用的模块来以默认的方式处理消息，Win32 中的 DefWindowProc函数实现的就是这个功能。

不要小看了这个 DefWindowProc，正是它用默认的方式处理了几百种消息，才使用户能用区区百来行代码写出一个全功能的窗口。也正是所有的窗口都用 DefWindowProc 默认处理程序自己不处理的消息，才使它们的行为看上去大同小异，因为它们背后实际上是同一块代码在处理。

在窗口过程的分支语句中，用户处理所有需要个性化处理的消息，对于表现行为是默认行为的消息，则在 else分支中用 DefWindowProc来处理。由于对于 Windows来说，它并不关心消息在窗口过程中是程序用自己的代码处理的还是用 DefWindowProc 处理的，它只看eax 中的返回值来了解处理结果，所以不管消息是谁处理的，都必须在 eax中返回正确的值。DefWindowProc 返回时 eax 中就是它对消息的处理结果，程序只要直接把 eax 传回给Windows就行了，所以在例子程序中，DefWindowProc后面直接用一句 ret 指令返回。

|  |
| --- |
| warning.bmp注意：例子中 DefWindowProc 后面直接使用的这句 ret 非常重要，如果丢失了这一句，那么相当于处理大多数消息时没有返回正确的值，窗口将不会正常工作。 |

表 4-9 中列出了 DefWindowProc中对一些消息的处理方法，如果与用户期望的不同，就必须在窗口过程中自己处理。

表4-9 DefWindowProc对一些消息的默认处理方式

|  |
| --- |
| 消 息 DefWindowProc的处理方式 |
| WM\_PAINT 发送WM\_ERASEBKGND消息来擦除背景  WM\_ERASEBKGND 用窗口类结构中的hbrBackground刷子来绘制窗口背景  WM\_CLOSE 调用DestroyWindow来摧毁窗口  WM\_NCLBUTTONDBLCLK 这是非客户区（如标题栏）鼠标双击消息，DefWindowProc测试 鼠标的位置，然后再采取相应的措施，如标题栏双击将最大化和恢复窗口  WM\_NCLBUTTONUP 这是非客户区鼠标释放消息，同样，DefWindowProc测试鼠标的位置然后再采取相应的措施，如鼠标在“关闭”按钮的位置释放将导致发送WM\_CLOSE消息  WM\_NCPAINT 非客户区绘制消息，DefWindowProc将绘制边框和客户区 |

从这些默认的处理方法可以看出，想要一个窗口和别的窗口看起来不一样，比如，想要窗口看起来像苹果机的窗口一样，并且把关闭按钮移到标题栏最左边去，那么可以自己处理WM\_NCPAINT消息， 把非客户区画成苹果机窗口的样子， 并把关闭按钮画到标题栏左边去，并且自己处理 WM\_NCLBUTTONUP 消息，当检测到鼠标按下的位置在自己的关闭按钮上的时候，则发送 WM\_CLOSE消息。对别的消息的处理思路也可以按这种方法类推。

另外，可以发现 DefWindowProc 对 WM\_CLOSE 的默认处理是调用 DestroyWindow 摧毁窗口， DestroyWindow会引发一个 WM\_DESTROY消息， WM\_CLOSE和WM\_DESTROY的不同之处是：WM\_CLOSE 代表用户有关闭窗口的意向，窗口过程有权不“服从” ，但收到 WM\_DESTROY 的时候窗口已经在关闭过程中了，不管窗口过程愿不愿意，窗口的关闭已经是不可挽回的事了。

对于这两个消息，窗口过程必须处理其中的一个，因为必须有个地方发送 WM\_QUIT消息来结束消息循环，例子程序中处理 WM\_CLOSE 消息，在其中用 DestroyWindow 摧毁窗口，再调用 PostQuitMessage 结束消息循环；程序也可以不处理 WM\_CLOSE 消息，让DefWindowProc 以默认处理的方式摧毁窗口，但这时候必须处理 WM\_DESTROY 消息，在其中调用 PostQuitMessage发送 WM\_QUIT以结束消息循环。

## 4.3 窗口间的通信

### 4.3.1 窗口间的消息互发

在介绍消息循环的时候，已经知道在不同应用程序之间的窗口中可以互发消息（如图4-4 所示） ，方法是通过 SendMessage或者 PostMessage 函数，这两个函数的使用语法是相同的：

invoke PostMessage,hWnd,Msg,wParam,lParam

invoke SendMessage,hWnd,Msg,wParam,lParam

对于不同的 Msg，wParam和 lParam的含义是不同的，如对于 WM\_SETTEXT是：

wParam = 0; ;未定义，必须为0

lParam = (LPARAM)(LPCTSTR)lpsz ;要设置的字符串地址

想一想就会发现一个问题：Windows中不同应用程序的地址空间是隔离的（如图 1-6所示） ，假设程序 1 要用 SendMessage 调用程序 2 所属窗口的窗口过程，但程序 2 窗口过程的代码并不在程序 1 的地址空间中，那么 SendMessage如何调用它呢？其实很简单，当程序 1调用 SendMessage 函数的时候，Windows 会先保存 wParam 和 lParam 参数并等待，等轮到程序 2 的时间片的时候再去调用它的窗口过程，并把保存的 wParam和 lParam参数发给它，等窗口过程返回的时候，Windows记下返回值并等待，再等轮到程序 1 的时间片的时候把返回值当做 SendMessage 的返回值传给程序 1，这样程序 1 看上去就像自己直接在调用程序 2的窗口过程一样。

但又一个问题出现了：Windows在做“牵线红娘”的时候传递了 wParam和 lParam，以及返回值，如果参数指向一个字符串呢，比如说上面的 WM\_SETTEXT 消息中的 lParam指向一个字符串，假设程序 1 中lParam指向字符串的地址为 xxxxxxxx，把这个地址传给程序2 的时候，程序 2 不可能访问到程序 1 的地址空间，在程序 2 中 xxxxxxxx 指向的可能是其他内容，也可能是不可访问的，这又该如何处理呢？

Windows 在处理 SendMessage 的时候要检查消息的类型，并对不同的消息做不同的处理，当消息的参数是一个普通的 32 位数时，仅仅将该数值传递给目标窗口过程；而当消息的参数是一个指针的时候，Windows对指针指向的内容进行了一些处理，以便数据能够正常地传递到目标进程中。

Windows 首先创建一块共享内存，并将 WM\_SETTEXT 消息 lParam指向的字符串拷贝到该内存中，然后再发送消息到其他进程，并将共享内存在目标进程中的地址发送给目标窗口过程，目标窗口过程处理完消息后，函数返回，共享内存被释放。共享内存使用的是第10 章介绍的内存映射文件技术， 相当于用图 1.6 的方法将同一块物理内存映射到不同进程的不同线性地址上去。

虽然当消息传递到目标窗口过程的时候lParam的取值会有所变化， 但在WM\_SETTEXT消息中，lParam的数值是多少并不重要，重要的是它指向的字符串是否正确。

|  |
| --- |
| warning.bmp在用户自定义的消息中（WM\_USER等），不要在消息参数中传递指针，这只会引发非法访问内存，因为 Windows 不知道用户的意图，它只会把 lParam 和 wParam 当两个普通的数值传递，而不会帮用户把指针指向的内容复制到一块共享内存中。 |

### 4.3.2 在窗口间传递数据

在 WM\_SETTEXT 这一类的消息中，Windows 可以将参数所指的字符串传递到目标窗口过程中， 但是这些消息都有它们的本职工作， 并且传递的数据也只限于以 0 结尾的字符串。为了能够在不同进程的窗口间自由地拷贝任意类型的数据， Windows提供了一个特殊的窗口消息——WM\_COPYDATA。

WM\_COPYDATA 消息用一个 COPYDATASTRUCT 结构来描述要拷贝的数据的长度和

位置：

COPYDATASTRUCT STRUCT

dwData DWORD ? ; 附加字段

cbData DWORD ? ; 数据长度

lpData DWORD ? ; 数据位置指针

COPYDATASTRUCT ENDS

其中的 dwData字段是一个备用的字段，可以存放任何值，例如，读者有可能向另外的进程发送数据的同时用一个数字来说明数据的类型，那么就可以把这个字段用上去；cbData 字段规定了发送的字节数，lpData 字段是指向待发送数据的指针。填充好数据结构后，用SendMessage函数就可以将数据发送给目标窗口过程：

.data

stCopyData COPYDATASTRUCT <>

.code

...

invoke SendMessage,hDestWnd,WM\_COPYDATA,hWnd,addr stCopyData

例句中的 hDestWnd 为目标窗口句柄；wParam 指定为 hWnd，是当前窗口的句柄；lParam指向已经填充完毕的 COPYDATASTRUCT结构。

Windows收到 WM\_COPYDATA消息后， 会根据 cbData字段的长度创建一块共享内存，并把 lpData 所指的数据拷贝到共享内存中，然后定位该共享内存在目标进程中的地址，把该地址作为新的地址添加到 COPYDATASTRUCT结构的 lpData 字段中，最后将经过处理的COPYDATASTRUCT 结构发送给目标窗口过程，目标窗口过程就可以根据结构中的字段来定位数据了。目标窗口过程返回后，Windows释放掉共享内存，SendMessage 函数返回。

### 4.3.3 SendMessage和PostMessage函数的区别

从逻辑上看，SendMessage函数相当于直接调用其他窗口的窗口过程来处理某个消息，并等待窗口过程的返回，在函数返回后，目标窗口过程必定已经处理了该消息。PostMessage函数则将消息放入目标窗口的消息队列中并直接返回，函数返回后，目标窗口过程可能还没有处理到该消息。

对于普通的消息来说， 两个函数除了在处理速度上有所区别外， 其他的表现都一模一样，但是对于 WM\_SETTEXT，WM\_COPYDATA等在参数中用到指针的消息来说，两者就有所不同了。读者可以尝试将前面例子中的 SendMessage函数改为 PostMessage 函数，就会发现Receive程序根本不会接收到 WM\_SETTEXT或者WM\_COPYDATA消息。事实上，当消息参数中用到指针时，用 PostMessage 函数来发送消息都不会成功，PostMessage 的参考文档中明确地说明，该函数不能用于任何参数中用到指针的消息。

# 第5章 使用资源

## 5.1 菜单和加速键

### 5.1.1 菜单和加速键的组成

如图5.1所示，在窗口中菜单位于标题栏下面，这个菜单称为“主菜单”或“顶层菜单”，菜单中的菜单项有“文件”、“查看”和“帮助”；单击主菜单上的项目后，可以弹出下一层菜单，叫做“弹出式菜单”或“子菜单”，子菜单中可以继续包含下一层子菜单，如单击“查看”弹出一个子菜单后，再单击其中的“工具栏”可以继续弹出一个子菜单，在子菜单中可以继续弹出下一层子菜单的菜单项最右边用一个三角箭头来表示。

有的程序在窗口的客户区单击鼠标右键也可以弹出一个菜单，单击标题栏图标也可以弹出一个系统菜单，这些菜单都属于弹出式菜单。

菜单中的菜单项有好几种，从资源定义的角度来看，分隔用的横线也是一个菜单项，除横线外其他菜单项可以供用户选择，也可以设置为“禁止”或“灰化”状态暂时停用，如图5.1中“被禁用的菜单项”和“被灰化的菜单项”所示。“禁用”的菜单项看上去和普通菜单项相同，但无法在上面单击鼠标，“灰化”的菜单项从外观上就已经表示是不可用的。菜单项也可以在左边显示选中标记，如图5.1中的“大图标”前的圆点和“状态栏”前的对钩，圆点表示选中标记是互斥的，对钩表示是不互斥的。



图5.1 菜单示意图

加速键就是菜单项的快捷键，如图5.1中的“字体”菜单项右边有个说明，表示当窗口是激活的时候，不必打开菜单，直接按“Alt”加“F”的组合键就相当于选择了“字体”菜单项，同样，直接按下“Ctrl”加“Alt”加“B”键等于选择了“背景色”菜单项，加速键也是资源的组成部分，一般将最常用的菜单项定义为加速键，以减少打开菜单的操作。加速键的定义要遵循惯例，如“Ctrl＋C”和“Ctrl＋V”一般定义为“拷贝”和“粘贴”，“Ctrl＋X”定义为“剪切”等。当然加速键的定义并不是必需的，不定义加速键并不会影响程序的功能。

不管程序中是否定义加速键，Windows总是定义了几个默认的加速键，如“F10”键会打开窗口的主菜单，“Alt＋空格”会打开系统菜单，“Alt＋F4”等于单击了“关闭”按钮等。

### 5.1.2 菜单和加速键的资源定义

这里有一个资源脚本文件的例子，运行后产生如图5.1所示的菜单：

//filename:menu.rc

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

#include <resource.h>

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

#define ICO\_MAIN 0x1000 //图标

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

#define IDM\_MAIN 0x2000 //菜单

#define IDA\_MAIN 0x2000 //加速键

#define IDM\_OPEN 0x4101

#define IDM\_OPTION 0x4102

#define IDM\_EXIT 0x4103

#define IDM\_SETFONT 0x4201

#define IDM\_SETCOLOR 0x4202

#define IDM\_INACT 0x4203

#define IDM\_GRAY 0x4204

#define IDM\_BIG 0x4205

#define IDM\_SMALL 0x4206

#define IDM\_LIST 0x4207

#define IDM\_DETAIL 0x4208

#define IDM\_TOOLBAR 0x4209

#define IDM\_TOOLBARTEXT 0x4210

#define IDM\_INPUTBAR 0x4211

#define IDM\_STATUSBAR 0x4212

#define IDM\_HELP 0x4301

#define IDM\_ABOUT 0x4302

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

ICO\_MAIN ICON "Main.ico"

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

IDM\_MAIN menu discardable

BEGIN

popup "文件(&F)"

BEGIN

menuitem "打开文件(&O)...", IDM\_OPEN

menuitem "关闭文件(&C)...", IDM\_OPTION

menuitem separator

menuitem "退出(&X)", IDM\_EXIT

END

popup "查看(&V)"

BEGIN

menuitem "字体(&F)...\tAlt+F",IDM\_SETFONT

menuitem "背景色(&B)...\tCtrl+Alt+B",IDM\_SETCOLOR

menuitem separator

menuitem "被禁用的菜单项", IDM\_INACT, INACTIVE

menuitem "被灰化的菜单项", IDM\_GRAY, GRAYED

menuitem separator

menuitem "大图标(&G)", IDM\_BIG

menuitem "小图标(&M)", IDM\_SMALL

menuitem "列表(&L)", IDM\_LIST

menuitem "详细资料(&D)", IDM\_DETAIL

menuitem separator

popup "工具栏(&T)"

BEGIN

menuitem "标准按钮(&S)", IDM\_TOOLBAR

menuitem "文字标签(&C)", IDM\_TOOLBARTEXT

menuitem "命令栏(&I)", IDM\_INPUTBAR

END

menuitem "状态栏(&U)", IDM\_STATUSBAR

END

popup "帮助(&H)" ,HELP

BEGIN

menuitem "帮助主题(&H)\tF1", IDM\_HELP

menuitem separator

menuitem "关于本程序(&A)...", IDM\_ABOUT

END

END

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

IDA\_MAIN accelerators

BEGIN

VK\_F1, IDM\_HELP, VIRTKEY

"B", IDM\_SETCOLOR,VIRTKEY,CONTROL,ALT

"F", IDM\_SETFONT,VIRTKEY,ALT

END

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

编译上述文件使用的makefile文件如下：

NAME = Menu

OBJS = $(NAME).obj

RES = $(NAME).res

LINK\_FLAG = /subsystem:windows

ML\_FLAG = /c /coff

$(NAME).exe: $(OBJS) $(RES)

Link $(LINK\_FLAG) $(OBJS) $(RES)

.asm.obj:

ml $(ML\_FLAG) $<

.rc.res:

rc $<

clean:

del \*.obj

del \*.res

为了编译资源文件，makefile中比以前多了一个资源编译的隐含规则：

.rc.res:

rc $<

同时在exe文件的依赖文件中增加了Menu.res文件。

在rc文件中，各种语句使用的是C语言的格式，因为资源编译器Rc.exe根本上就是Visual C++附带的（这一点在第2章中就提及过），所以在定义等值语句的时候用的是#define，包含语句使用#include<文件名>，用到16进制数值的时候并不是用汇编的语法在后面加h，而是用前面加Ox的方法，如1234h写为Ox1234，注释也要用前面加//的方法。这一点在书写的时候一定要注意，以免引起语法错误。

在脚本文件的头部，首先要把MASM32软件包中的resource.h文件包含进来，这个文件中包括了资源定义中很多的预定义值，如窗口属性与加速键的键值等。资源在程序中的引用往往用一个数值来表示，称为资源的ID值，但在定义的时候直接使用数值不是很直观，所以往往用#define语句将数值定义为容易记忆的字符串。

**1. 菜单的定义**

在资源脚本文件中菜单的定义格式是：

菜单ID MENU [DISCARDABLE]

BEGIN

菜单项定义

…

END

“菜单ID MENU [DISCARDABLE]”语句用来指定菜单的ID值和内存属性，菜单ID可以是16位的整数，范围是1～65 535，在Menu.rc文件中，定义的菜单ID是2000h，但菜单ID也可以用字符串表示，如下面的定义：

MainMenu menu

begin

menuitem …

end

表示菜单的ID是字符串型的“MainMenu”，但这样定义的话，在程序中引用的时候就要用字符串指针代替16进制的菜单ID值，显得相当不便，所以在实际应用中通常使用16进制数值当做菜单ID。

数值型ID的范围限制在1～65 535之间的原因是字符串在内存中的线性地址总是大于10000h，API函数检测参数时发现小于10000h时就可以把它认为是数值型的，大于10000h时就当做字符串指针处理。

menu关键字后面的DISCARDABLE是菜单的内存属性，表示菜单在不再使用的时候可以暂时从内存中释放以节省内存，这是一个可选属性。菜单项的定义语句必须包含在begin和end关键字之内，这两个关键字也可以用花括号{ 和 } 代替。

菜单项目的定义方法有3类：

MENUITEM 菜单文字,命令ID [，选项列表] （用法1）

或 MENUITEM SEPARATOR （用法2）

或 POPUP 菜单文字 [，选项列表] （用法3）

BEGIN

item-definitions

...

END

下面分别就这3类详细说明，用法1定义的是普通菜单项，图5.1中的“字体”与“背景色”等菜单项都是这样定义的，它的组成部分如下：

● 菜单文字——显示在菜单项中的字符串。如果需要字符串中某个字母带下横线，那么可以在字母前面加&符号，如“字体(F)...”就要写成“字体(&F)...”，带下横线的字母可以被系统自动当做快捷键（跟加速键没关系）：在这里，当菜单打开的时候按下F键，那么就相当于用鼠标选择了“字体”选项。在同一个弹出菜单中要注意不同的菜单项快捷键应该有所区别。另外，如果要把加速键的提示信息显示在菜单项的右边，如“字体”菜单项中的“Alt＋F”字符，可以在两者中间加\t（表示插入一个Tab字符），写为“字体(&F)...\tAlt+F”，这样Tab后面的字符在显示的时候会右对齐。

● 命令ID——用来分辨不同的菜单项。当菜单被选中的时候，Windows会向窗口过程发送WM\_COMMAND消息，消息的参数就是这个命令ID。用命令ID可以分辨用户究竟选中了哪个菜单项，所以不同的菜单项应该定义不同的ID值，除非想让两个菜单项的功能相同。

● 选项——用来定义菜单项的各种属性，它可以是下列数值：

CHECKED——表示打上选定标志（对钩）。

GRAYED——表示菜单项是灰化的。

INACTIVE——表示菜单项是禁用的。

MENUBREAK或MENUBARBREAK——表示将这个菜单项和以后的菜单项列到新的列中。

读者可以做个实验，当把例子中的“详细资料”一句的定义语句改为：

menuitem "详细资料(&D)",IDM\_DETAIL，MENUBARBREAK

那么显示出来的菜单如图5.2所示。“详细资料”以及以后的菜单项都另起一列了！



图5.2 使用MENUBARBREAK的效果

用法2定义的是菜单项之间的分隔线，显然，分隔线是不需要字符串和选项的。

方法3定义的是弹出式菜单，顶层菜单是由多个弹出式子菜单组成的，所以在Menu.rc文件中，主菜单是由“文件”、“查看”和“帮助”3个顺序定义的弹出式菜单组成的，弹出式菜单的定义也可以嵌套，如“查看”菜单中的“工具栏”又是一个弹出式菜单，在嵌套的时候要注意像写C的源程序一样把begin和end（或者{ 和 }）正确地配对。popup菜单的选项列表可以是以下的值：

● GRAYED——表示菜单项是灰化的。

● INACTIVE——表示菜单项是禁用的。

● HELP——表示本项和以后的菜单项是右对齐的，如图5.1中的“帮助”菜单。

popup菜单项选中的时候会自动将弹出式菜单弹出来，不需要向程序发送消息，所以在定义的参数中不需要命令ID。

有些选项是可以同时定义的，如果要指定超过一个的选项，中间要用逗号隔开，但是也有些小小的限制：GRAYED和INACTIVE不能同时使用，MENUBREAK和MENUBARBREAK也是不能同时使用的。

**2. 加速键的定义**

和菜单的定义相比，加速键的定义要简单得多，具体的语法如下：

加速键ID ACCELERATORS

BEGIN

键名, 命令ID [，类型] [，选项]

...

END

加速键ID同样可以是一个字符串或者是1～65 535之间的数字，整个定义内容也是用begin和end（或花括号）包含起来，中间是多个加速键的定义项目，每个键占据一行，各字段的含义如下所示。

● 键名——表示加速键对应的按键，可以有4种方式定义：

"^字母"：表示Ctrl键加上字母键。

"字母"：表示字母。

虚拟键：虚拟键，这时类型必须指明是VIRTKEY。

数值：表示ASCII码为该数值的字母，这时类型必须指明为ASCII。

● 命令ID——按下加速键后，Windows向程序发送的命令ID。如果想把加速键和菜单项关联起来，这里就是要关联菜单项的命令ID。

● 类型——用来指定键的定义方式，可以是VIRTKEY和ASCII，分别用来表示“键名”字段定义的是虚拟键还是ASCII码。

● 选项——可以是Alt，Control或Shift中的单个或多个，如果指定多个，则中间用逗号隔开，表示加速键是按键加上这些控制键的组合键。这些选项只能在类型是VIRTKEY的情况下才能使用。

在键名的定义中，系统按键如F1，F2，BackSpace和Esc等都是用虚拟键的方法定义的，Resource.h中已经包括所有的预定义，它们是以VK\_带头的一些值，如VK\_BACK，VK\_TAB，VK\_RETURN，VK\_ESCAPE，VK\_DELETE，VK\_F1和VK\_F2等，读者可以查看Resource.h文件。下面是加速键定义的一些例子：

"^C", ID ；Ctrl＋C

"K", ID ；Shift＋K

"k", ID，ALT ；Alt＋k

98, ID，ASCII ；b（字符b的ASCII码为98）

66, ID，ASCII ；B (Shift b)

"g", ID ；g

VK\_F1, ID，VIRTKEY ；F1

VK\_F1, ID，VIRTKEY，CONTROL ；Ctrl＋F1

VK\_F2, ID，VIRTKEY，ALT，SHIFT ；Alt＋Shift＋F2

在一个资源脚本文件中，可以定义多个菜单和多个加速键表，当然也有其他各式各样的资源，有位图、图标与对话框等，这就涉及为这些资源取ID值的问题，取值的时候要掌握的原则是：

（1）对于同类别的多个资源，资源ID必须为不同的值，如定义了两个菜单，那么它们的ID就必须用不同的数值表示，否则将无法分辨。

（2）对于不同类别的资源，资源ID在数值上可以是相同的，如可以将菜单和加速键的ID都定义为1，同时也可以有ID为1的位图或图标等，Windows并不会把它们搞混。

### 5.1.3 使用菜单和加速键

下面是Menu.asm源代码：

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; Menu.asm

; 菜单资源的使用例子

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; 使用 nmake 或下列命令进行编译和链接:

; ml /c /coff Menu.asm

; rc Menu.rc

; Link /subsystem:windows Menu.obj Menu.res

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.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

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; Equ 等值定义

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

ICO\_MAIN equ 1000h ;图标

IDM\_MAIN equ 2000h ;菜单

IDA\_MAIN equ 2000h ;加速键

IDM\_OPEN equ 4101h

IDM\_OPTION equ 4102h

IDM\_EXIT equ 4103h

IDM\_SETFONT equ 4201h

IDM\_SETCOLOR equ 4202h

IDM\_INACT equ 4203h

IDM\_GRAY equ 4204h

IDM\_BIG equ 4205h

IDM\_SMALL equ 4206h

IDM\_LIST equ 4207h

IDM\_DETAIL equ 4208h

IDM\_TOOLBAR equ 4209h

IDM\_TOOLBARTEXT equ 4210h

IDM\_INPUTBAR equ 4211h

IDM\_STATUSBAR equ 4212h

IDM\_HELP equ 4301h

IDM\_ABOUT equ 4302h

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; 数据段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.data?

hInstance dd ?

hWinMain dd ?

hMenu dd ?

hSubMenu dd ?

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; 数据段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.const

szClassName db 'Menu Example',0

szCaptionMain db 'Menu',0

szMenuHelp db '帮助主题(&H)',0

szMenuAbout db '关于本程序(&A)...',0

szCaption db '菜单选择',0

szFormat db '您选择了菜单命令：%08x',0

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; 代码段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.code

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

\_DisplayMenuItem proc \_dwCommandID

local @szBuffer[256]:byte

pushad

invoke wsprintf,addr @szBuffer,addr szFormat,\_dwCommandID

invoke MessageBox,hWinMain,addr @szBuffer,offset szCaption,MB\_OK

popad

ret

\_DisplayMenuItem endp

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

\_Quit proc

invoke DestroyWindow,hWinMain

invoke PostQuitMessage,NULL

ret

\_Quit endp

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

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

local @stPos:POINT

local @hSysMenu

mov eax,uMsg

.if eax == WM\_CREATE

invoke GetSubMenu,hMenu,1

mov hSubMenu,eax

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 在系统菜单中添加菜单项

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

invoke GetSystemMenu,hWnd,FALSE

mov @hSysMenu,eax

invoke AppendMenu,@hSysMenu,MF\_SEPARATOR,0,NULL

invoke AppendMenu,@hSysMenu,0,IDM\_HELP,offset szMenuHelp

invoke AppendMenu,@hSysMenu,0,IDM\_ABOUT,offset szMenuAbout

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 处理菜单及加速键消息

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif eax == WM\_COMMAND

invoke \_DisplayMenuItem,wParam

mov eax,wParam

movzx eax,ax

.if eax == IDM\_EXIT

call \_Quit

.elseif eax >= IDM\_TOOLBAR && eax <= IDM\_STATUSBAR

mov ebx,eax

invoke GetMenuState,hMenu,ebx,MF\_BYCOMMAND

.if eax == MF\_CHECKED

mov eax,MF\_UNCHECKED

.else

mov eax,MF\_CHECKED

.endif

invoke CheckMenuItem,hMenu,ebx,eax

.elseif eax >= IDM\_BIG && eax <= IDM\_DETAIL

invoke CheckMenuRadioItem,hMenu,IDM\_BIG,IDM\_DETAIL,eax,MF\_BYCOMMAND

.endif

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 处理系统菜单消息

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif eax == WM\_SYSCOMMAND

mov eax,wParam

movzx eax,ax

.if eax == IDM\_HELP || eax == IDM\_ABOUT

invoke \_DisplayMenuItem,wParam

.else

invoke DefWindowProc,hWnd,uMsg,wParam,lParam

ret

.endif

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 按下右键时弹出一个POPUP菜单

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif eax == WM\_RBUTTONDOWN

invoke GetCursorPos,addr @stPos

invoke TrackPopupMenu,hSubMenu,TPM\_LEFTALIGN,@stPos.x,@stPos.y,NULL,hWnd,NULL

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif eax == WM\_CLOSE

call \_Quit

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.else

invoke DefWindowProc,hWnd,uMsg,wParam,lParam

ret

.endif

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

xor eax,eax

ret

\_ProcWinMain endp

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

\_WinMain proc

local @stWndClass:WNDCLASSEX

local @stMsg:MSG

local @hAccelerator

invoke GetModuleHandle,NULL

mov hInstance,eax

invoke LoadMenu,hInstance,IDM\_MAIN

mov hMenu,eax

invoke LoadAccelerators,hInstance,IDA\_MAIN

mov @hAccelerator,eax

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 注册窗口类

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

invoke RtlZeroMemory,addr @stWndClass,sizeof @stWndClass

invoke LoadIcon,hInstance,ICO\_MAIN

mov @stWndClass.hIcon,eax

mov @stWndClass.hIconSm,eax

invoke LoadCursor,0,IDC\_ARROW

mov @stWndClass.hCursor,eax

push hInstance

pop @stWndClass.hInstance

mov @stWndClass.cbSize,sizeof WNDCLASSEX

mov @stWndClass.style,CS\_HREDRAW or CS\_VREDRAW

mov @stWndClass.lpfnWndProc,offset \_ProcWinMain

mov @stWndClass.hbrBackground,COLOR\_WINDOW + 1

mov @stWndClass.lpszClassName,offset szClassName

invoke RegisterClassEx,addr @stWndClass

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 建立并显示窗口

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

invoke CreateWindowEx,WS\_EX\_CLIENTEDGE,\

offset szClassName,offset szCaptionMain,\

WS\_OVERLAPPEDWINDOW,\

100,100,400,300,\

NULL,hMenu,hInstance,NULL

mov hWinMain,eax

invoke ShowWindow,hWinMain,SW\_SHOWNORMAL

invoke UpdateWindow,hWinMain

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 消息循环

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.while TRUE

invoke GetMessage,addr @stMsg,NULL,0,0

.break .if eax == 0

invoke TranslateAccelerator,hWinMain,@hAccelerator,addr @stMsg

.if eax == 0

invoke TranslateMessage,addr @stMsg

invoke DispatchMessage,addr @stMsg

.endif

.endw

ret

\_WinMain endp

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

start:

call \_WinMain

invoke ExitProcess,NULL

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

end start

**1. 加载菜单**

在窗口中加载菜单的方法在第4章已经提及，方法有两个：一是在注册窗口类的时候指定类的默认菜单；二是在建立窗口的时候在参数中指定菜单句柄。Menu.asm程序中用的是第2种方法：

invoke CreateWindowEx,WS\_EX\_CLIENTEDGE,\

offset szClassName,offset szCaptionMain,\

WS\_OVERLAPPEDWINDOW,\

100,100,400,300,\

NULL,hMenu,hInstance,NULL

在参数中指出了hMenu。不管用哪种方法，首先都必须使用LoadMenu函数来获取菜单句柄hMenu，如下面的语句：

invoke LoadMenu,hInstance,IDM\_MAIN

mov hMenu,eax

这个函数的第1个参数是用GetModuleHandle获取的实例句柄，第2个参数指定需要装入的菜单资源ID，函数返回菜单句柄。在得到菜单句柄以后，我们先把它放入hMenu变量保存起来以便后用。

当资源文件中用字符串为名称定义菜单而不是用数值的时候，例如：

MainMenu menu //定义菜单名为字符串“MainMenu”

begin

...

end

那么在程序中就必须用字符串指针代替菜单ID做参数：

szMenu "MainMenu",0 ；在数据段中定义菜单名称字符串

...

invoke LoadMenu,hInstance,addr szMenu ；在程序中装载

mov hMenu,eax

|  |
| --- |
| 灯泡.BMP用字符串为名称定义资源，在资源装载函数LoadXXXX中用字符串指针做参数装入，这实际上是一个通用的方法，不仅适用于菜单资源，对于其他类别的资源也是适用的。在其他资源的介绍中就不再另外说明了。 |

**2. 加载加速键**

和菜单一样，加速键在使用前也要装入，参数同样是在资源脚本文件中定义的加速键ID，程序中对应的语句是：

invoke LoadAccelerators,hInstance,IDA\_MAIN

mov @hAccelerator,eax

其实我们自己在程序中也可以很方便地实现加速键功能，方法是：在WM\_KEYDOWN消息中判断键盘消息并按照自定义的逻辑进行处理，使用加速键实际上是让Windows替我们完成这个功能，Windows实现的方法正是在消息循环中检查WM\_KEYDOWN和WM\_SYSKEYDOWN消息。下面是使用加速键时消息循环的代码，请注意粗体字部分：

.while TRUE

invoke GetMessage,addr @stMsg,NULL,0,0

.break .if eax == 0

invoke TranslateAccelerator,hWinMain,@hAccelerator,addr @stMsg

.if eax == 0

invoke TranslateMessage,addr @stMsg

invoke DispatchMessage,addr @stMsg

.endif

.endw

TranslateAccelerator函数是实现加速键功能的核心，它的参数为目标窗口、加速键句柄和GetMessage取得的消息结构。该函数检查消息结构中的消息，如果遇到WM\_KEYDOWN和WM\_SYSKEYDOWN消息则检测加速键资源，看按键是否符合某个加速键，符合的话则向目标窗口发送WM\_COMMAND或WM\_SYSCOMMAND消息，并返回TRUE，不符合的话不进行任何处理并返回FALSE。

由于加速键的键码并不是用户真正想输入窗口的，比如用户在写字板中输入文字，按Ctrl＋C键是为了“拷贝”，而并不是想把Ctrl＋C键对应的字符输入文档，所以这个Ctrl＋C的键码在完成加速键的使命后就应该丢弃，也就是说符合加速键的键盘消息不应该再发送给窗口，TranslateMessage和DispatchMessage函数前的逻辑判断就是这样的意图：只有TranslateAccelerator没有转换的消息（返回值eax为0）才继续处理。

另外，TranslateAccelerator的参数中有个“目标窗口”，例子中是程序的主窗口hWinMain，为什么要设置这样一个参数而不像DispatchMessage函数一样使用MSG结构中的hwnd呢？这是因为键盘消息可以产生于不同窗口中——既可能是主窗口，也可能是其他子窗口，如果用@stMsg.hwnd做目标窗口，就必须在所有子窗口的窗口过程中都设置处理加速键消息的代码，这显然是一种浪费，所以一般把所有的加速键消息都发送到主窗口，然后集中在主窗口的窗口过程中处理WM\_COMMAND消息，这样有利于精简代码。

**3. 菜单和加速键消息**

当用户选择了一个菜单项的时候，Windows向菜单所属的窗口发送WM\_COMMAND消息；而用户按下了一个加速键的时候，Windows向TranslateAccelerator函数指定的目标窗口发送WM\_COMMAND消息。一般这两种情况对应的窗口都是主窗口，所以可以在主窗口中的窗口过程中集中处理WM\_COMMAND消息，而不必考虑它究竟是菜单引发的还是加速键引发的。

WM\_COMMAND消息的两个参数是这样定义的：

wParam的高位 ＝ wNotifyCode ；通知码

wParam的低位 ＝ wID ；命令ID

lParam ＝ hwndCtl ；发送WM\_COMMAND的子窗口句柄

除了菜单和加速键，WM\_COMMAND消息也可以由其他子窗口引发，如主窗口中的按钮或工具栏等，lParam参数指定了引发消息的子窗口句柄，对于菜单和加速键引发的WM\_COMMAND消息，lParam的值为零。wParam参数的低16位是命令ID，也就是资源脚本文件中菜单项的命令ID或加速键的命令ID，高16位是通知码，菜单消息的通知码是0，加速键消息的通知码为1。

在需要处理菜单和加速键消息的窗口过程中，一般需要增加一个WM\_COMMAND分支来处理对应的消息，这个分支的一般结构为：

.elseif eax == WM\_COMMAND ；eax中为wMsg

mov eax,wParam

movzx eax,ax

.if eax == 命令ID1

...

.elseif eax == 命令ID2

...

.endif

其中movzx eax,ax指令将16位的ax扩展到32位的eax，相当于将eax的高16位填零，作用就是当消息由加速键引起时，将高16位中的1忽略，这样下面的分支就可以同时处理菜单和加速键消息，当然读者也可以去掉这一句，这样下面的比较语句中就要使用ax而不是eax。

在例子程序中，mov eax,wParam前面还有一句invoke \_DisplayMenuItem,wParam，作用是在处理WM\_COMMAND消息前将wParam的值通过一个对话框显示出来，读者可以和资源脚本文件中定义的命令ID值对比一下，在正常使用的程序中可以去掉这一句。

读者可以发现，资源文件中定义的“字体”菜单项的ID为Ox4201，当选中“字体”菜单项的时候，对话框中显示的wParam数值正是00004201，而按下加速键Alt＋F的时候，显示出来的值就是00014201了，它们的区别就是高16位中的通知码不同。

**4. 菜单项的修改**

在程序的运行中也可以动态修改菜单项，包括添加、删除和修改操作，这些操作是通过几个API函数来完成的：

invoke AppendMenu,hMenu,uFlags,uIDNewItem,lpNewItem ；添加菜单项

invoke InsertMenu,hMenu,uPosition,uFlags,uIDNewItem,lpNewItem ；插入菜单项

invoke ModifyMenu,hMenu,uPosition,uFlags,uIDNewItem,lpNewItem ；修改菜单项

invoke DeleteMenu,hMenu,uPosition,uFlags ；删除菜单项

invoke RemoveMenu,hMenu,uPosition,uFlags ；删除菜单项

其中AppendMenu用来在一个菜单的最后添加菜单项，InsertMenu则在中间插入菜单项，ModifyMenu可以修改一个菜单项的文字，DeleteMenu和RemoveMenu则可以删除一个菜单项。

这些函数中的参数都是雷同的，hMenu参数指要操作的菜单句柄；uPosition用来定位要操作的菜单项，定位的方法有两种：用命令ID定位或用位置索引，用哪一种方法取决于后面的uFlags参数，当uFlags为MF\_BYCOMMAND时，uPostion为菜单项的命令ID，而当uFlags为MF\_BYPOSITION的时候，uPostion表示菜单项的位置索引，索引是从0开始的，也就是说第一个菜单项的索引值为0。

AppendMenu函数总是在菜单的最后添加新的菜单项，所以不需要uPostion参数。

对于AppendMenu和InsertMenu，会有一个新的菜单项产生，uIDNewItem就表示这个新菜单项的命令ID，lpNewItem指向新菜单项的文字字符串，ModifyMenu函数可以修改一个菜单项的命令ID或文字字符串，所以也有uIDNewItem和lpNewItem参数。而用来删除菜单项的DeleteMenu和RemoveMenu显然用不着uIDNewItem和lpNewItem参数。

uFlags参数除了指定MF\_BYCOMMAND还是MF\_BYPOSITION外，还可以组合指定菜单项的其他属性，如MF\_CHECKED，MF\_DISABLED，MF\_ENABLED，MF\_GRAYED，MF\_MENUBARBREAK，MF\_MENUBREAK，MF\_SEPARATOR和MF\_UNCHECKED等，从其字面上就可以看出这些属性的含义。

DeleteMenu和RemoveMenu的不同之处在于对popup菜单项的处理。当它们用于popup属性的菜单项时，DeleteMenu不仅删除菜单项，而且将这个popup菜单项的所有子项目全部删除，这样，这个popup菜单就不能在别的地方继续使用；而RemoveMenu仅从菜单中移去这个popup菜单项，整个popup菜单在内存中还是存在的。以Menu.asm程序为例，按鼠标右键弹出的菜单实际上是主菜单中的“查看”菜单项，假如用DeleteMenu删除主菜单中的“查看”项目，那么按右键也就弹不出菜单了，而用RemoveMenu删除主菜单中的“查看”项目，按鼠标右键仍然可以弹出菜单。对于非popup属性的菜单项，DeleteMenu和RemoveMenu的效果是同样的。

**5. 使用系统菜单**

系统菜单指按下了标题栏图标后弹出的菜单，和窗口菜单不同，选中系统菜单的菜单项后，Windows向窗口发送的是WM\_SYSCOMMAND消息而非WM\_COMMAND消息。默认的系统菜单中已经有“还原”、“移动”、“大小”、“最大化”、“最小化”和“关闭”等菜单项，这些菜单项的命令ID已经预定义为SC\_RESTORE，SC\_MOVE，SC\_SIZE，SC\_MAXIMIZE，SC\_MINIMIZE和SC\_CLOSE等，如果读者要自己处理它们，可以在WM\_SYSCOMMAND消息中建立一个比较分支对它们进行处理，一般在程序中并不自己处理WM\_SYSCOMMAND消息，而是交给DefWindowProc处理。

如何在系统菜单中添加自己的菜单项呢？方法就是使用上面介绍的AppendMenu（当然也可以用InsertMenu），在添加前必须用GetSystemMenu函数首先获取系统菜单的句柄。例子程序在窗口初始化的时候在系统菜单尾添加了一个分隔线和两个菜单项：“帮助主题”和“关于本程序”：

.if eax == WM\_CREATE

...

invoke GetSystemMenu,hWnd,FALSE

mov @hSysMenu,eax

invoke AppendMenu,@hSysMenu,MF\_SEPARATOR,0,NULL

invoke AppendMenu,@hSysMenu,0,IDM\_HELP,offset szMenuHelp

invoke AppendMenu,@hSysMenu,0,IDM\_ABOUT,offset szMenuAbout

在窗口过程中处理系统菜单消息的分支结构为：

.elseif eax == WM\_SYSCOMMAND

mov eax,wParam

.if ax == 自定义ID1

...

.elseif ax == 自定义ID2

...

.else

invoke DefWindowProc,hWnd,uMsg,wParam,lParam

ret

.endif

和处理WM\_COMMAND消息不同的是，在WM\_SYSCOMMAND消息中处理了自定义的菜单命令ID后，必须把其他命令ID交给DefWindowProc处理，并直接把返回值返回给Windows，不然的话会发现窗口不能移动，不能关闭，不能最小化……因为它相当于屏蔽了所有SC\_RESTORE，SC\_MOVE，SC\_SIZE，SC\_MAXIMIZE，SC\_MINIMIZE和SC\_CLOSE等消息的处理。

**6. 右键弹出菜单**

例子程序的一个功能是当用户在窗口客户区按下鼠标右键的时候弹出一个菜单，这个功能是用TrackPopupMenu函数实现的。TrackPopupMenu函数的用法：

invoke TrackPopupMenu,hMenu,uFlags,x,y,nReserved,hWnd,lpRect

这个函数本身很简单，执行后在参数指定的x，y位置弹出一个属于hWnd窗口（也就是说WM\_COMMAND消息发到这个窗口）的菜单，菜单句柄是hMenu。函数中的坐标是以整个屏幕左上角为基准的，所以弹出菜单的位置不一定在窗口的客户区内，它可以是屏幕的任何一个地方。

uFlags参数指定一些和位置相关的选项，表示（x，y）坐标是代表弹出菜单位置的中间、左上角还是右上角等等，默认为TPM\_LEFTALIGN or TPM\_TOPALIG，（x,y）是菜单的左上角。uFlags中同时还可以指定用鼠标左键还是右键选定菜单项，定义值可以是TPM\_LEFTBUTTON或TPM\_RIGHTBUTTON，如果选择TPM\_RIGHTBUTTON的话，对在菜单项上面按鼠标左键是没有反应的。

lpRect已经废弃，永远为NULL。

在使用TrackPopupMenu之前，有几个准备工作是要做的：为了在客户区中按下鼠标右键弹出菜单，我们当然要处理鼠标右键消息，也就是说在WM\_RBUTTONDOWN消息中调用TrackPopupMenu函数，一般的习惯是在鼠标按下的地方弹出菜单，所以还要首先获取鼠标光标的位置，然后在此位置弹出菜单。

要获取鼠标位置，可以用GetCursorPos函数：

invoke GetCursorPos,lpPoint

参数lpPoint指向一个POINT数据结构，这个结构只有两个字段：

POINT STRUCT

x DWORD ?

y DWORD ?

POINT ENDS

该结构用来表示一个点的（x，y）坐标，GetCursorPos将当前的鼠标位置返回到这个结构中，程序中的相关代码是：

local @stPos:POINT ；首先定义一个POINT结构

...

invoke GetCursorPos,addr @stPos ；获取鼠标位置

invoke TrackPopupMenu,hSubMenu,\

TPM\_LEFTALIGN,@stPos.x,@stPos.y,NULL,hWnd,NULL

用GetCursorPos获取的鼠标位置是一个POINT结构，但TrackPopupMenu输入坐标的方法是用x，y两个参数，而不是一个POINT结构，所以要用结构中的两个字段@stPos.x和@stPos.y分别输入。

使用TrackPopupMenu时要注意的是，弹出的菜单句柄必须是popup类型的，而在资源文件中定义并且可以用LoadMenu函数装入的菜单并不是popup类型的，popup菜单（如例子中的“文件”与“查看”等）只能在第二层中才能定义，在程序中用GetSubMenu得到的第二层子菜单的句柄才是popup类型的。GetSubMenu函数的用法是：

invoke GetSubMenu,hMenu,nPos

.if eax

mov hSubMenu,eax

.endif

nPos参数指定要获取的菜单的位置索引，GetSubMenu的返回值是获取的子菜单句柄。

例子用invoke GetSubMenu,hMenu,1取得第二个子菜单（“文件”子菜单为0，“查看”子菜单为1，……）的句柄，然后在TrackPopupMenu中使用，这个菜单句柄就是主菜单中的“查看”菜单，所以按鼠标右键弹出的菜单和下拉菜单中的“查看”菜单是一模一样的。

**7. 菜单状态的检测和设置**

对菜单项状态的检测可以用GetMenuState函数来完成，用法是：

invoke GetMenuState,hMenu，uPosition，uFlags

参数hMenu是菜单的句柄，uPosition用来定位要检测的菜单项，当uFlags是MF\_BYCOMMAND的时候，uPosition表示菜单项的命令ID，默认为MF\_BYCOMMAND；当uFlags是MF\_BYPOSITION的时候，uPosition的值是位置索引，函数执行后的返回值为－1时表示失败，否则会是MF\_CHECKED，MF\_DISABLED，MF\_GRAYED，MF\_HILITE，MF\_MENUBARBREAK，MF\_MENUBREAK和MF\_SEPARATOR的组合值，它们分别表示菜单项的状态是选中、禁用、灰化、高亮显示以及3种分隔线，读者可以用test指令测试相应的数据位来分辨菜单项处于哪种状态，一般的测试代码如下：

invoke GetMenuState,hMenu,IDM\_XXX,MF\_BYCOMMAND

.if eax & MF\_CHECKED

；表示IDM\_XXX菜单项现在是选中状态

.endif

同样，读者也可以用eax & MF\_DISABLED和eax & MF\_GRAYED等条件测试其他状态。

设置菜单项的状态可以用下列3个函数来实现不同的功能：

invoke EnableMenuItem,hMenu,uIDEnableItem,uEnable

invoke CheckMenuItem,hMenu,uIDCheckItem,uCheck

invoke CheckMenuRadioItem,hMenu,idFirst,idLast,idCheck,uFlags

EnableMenuItem函数将菜单项在禁用、可用和灰化状态之间切换，uEnable可以为MF\_BYCOMMAND或MF\_BYPOSITION和MF\_DISABLED，MF\_ENABLED或MF\_GRAYED的组合。

CheckMenuItem函数将菜单项在非互斥的选定状态和非选定状态之间切换（即前面是否有对钩），uCheck可以是MF\_BYCOMMAND或MF\_BYPOSITION和MF\_CHECKED或MF\_UNCHECKED的组合。

CheckMenuRadioItem将菜单项在互斥的选定状态和非选定状态之间切换（即前面是否有圆点标志），由于互斥的菜单项在一个范围内只有一个是可以选定的，当选定另一个的时候，原来的选定应该撤销，idFirst和idLast就指定了这个互斥范围。函数在选定idCheck指定的菜单项的同时将自动清除idFirst和idLast范围内的其他选定。所以uFlags中无需指定状态，只需指定MF\_BYCOMMAND或MF\_BYPOSITION定位方法。

在例子程序中，当选中IDM\_TOOLBAR和IDM\_STATUSBAR之间的菜单项的时候，程序先用invoke GetMenuState,hMenu,ebx,MF\_BYCOMMAND获取当前的状态，检查是否选定，并将选定状态反转后用CheckMenuItem重新设置：

.elseif eax >= IDM\_TOOLBAR && eax <= IDM\_STATUSBAR

mov ebx,eax

invoke GetMenuState,hMenu,ebx,MF\_BYCOMMAND

.if eax == MF\_CHECKED

mov eax,MF\_UNCHECKED

.else

mov eax,MF\_CHECKED

.endif

invoke CheckMenuItem,hMenu,ebx,eax

当选中IDM\_BIG和IDM\_DETAIL之间的菜单项的时候，程序用CheckMenuRadioItem将原先IDM\_BIG和IDM\_DETAIL范围内的互斥选定撤销并将当前选定的菜单项加圆点标记。

.elseif eax >= IDM\_BIG && eax <= IDM\_DETAIL

invoke CheckMenuRadioItem,hMenu,IDM\_BIG,IDM\_DETAIL,eax,MF\_BYCOMMAND

最后，修改菜单状态的时机是什么时候呢？在程序中似乎不应该随时去检测状态并设置，这显然是很浪费资源的。Windows考虑到了这一点：在菜单将要激活的时候，也就是用户在菜单上按动鼠标的时候，Windows在菜单弹出之前会向窗口过程发送WM\_INITMENU消息，我们可以从容不迫地在这里进行各种检测，并设置对应的菜单项。

|  |
| --- |
| 灯泡.BMP读者可以注意到，在状态参数中指定MF\_BYCOMMAND或MF\_BYPOSITION将决定位置参数用命令ID还是位置索引表示，这个规则在所有的菜单函数中都是适用的，MF\_BYCOMMAND是默认值（它的定义值是0），如果两者都不定义的话，位置参数代表的就是命令ID。 |

**8. 其他菜单函数**

除了前面介绍的一些函数之外，还有一些不太常用的菜单函数，在这里作一个简单的介绍。

菜单不一定非要在资源文件中定义，在程序中也可以用代码来建立菜单，不过比较麻烦一点，方法是先用CreateMenu建立一个菜单，CreateMenu函数没有参数，调用后返回一个没有任何菜单项的菜单句柄，接下来就可以用AppendMenu在上面一条条地添加菜单项了。

同样，CreatePopupMenu也可以建立一个没有任何菜单项的菜单句柄，但它建立的是popup类型的菜单句柄，可以在TrackPopupMenu中直接使用。

如果要获取一个窗口当前使用的菜单句柄，那么可以使用GetMenu函数：

invoke GetMenu,hWnd

mov hMenu,eax

一个菜单的总项数可以用GetMenuItemCount函数获取:

invoke GetMenuItemCount,hMenu

不过GetMenuItemCount函数的返回值不包括子菜单展开以后的项数，而是指最上层菜单的项数，比如在例子程序中对hMenu统计的结果是3，因为Menu.rc中定义的最上层的菜单项是“文件”、“查看”和“帮助”，总共3个，如果要统计全部展开后的项数，那么只好用GetSubMenu一层层地统计下去了。

对菜单中各个菜单项当前的文字和命令ID也是可以查询的，方法是用GetMenuString和GetMenuItemID，读者可以参考命令手册。

建立窗口时指定了菜单句柄后并不是不能改变的，我们常常见到一些编辑软件，没有打开文件之前菜单只有寥寥几项，一打开文件以后功能菜单就全部出来了，实际上这是用SetMenu函数完成的：

invoke SetMenu,hWnd,hMenu

可以在资源文件中预定义几个不同的菜单，在使用的时候根据不同情况用SetMenu设置不同的菜单句柄。

使用菜单后，要涉及清除的问题，和窗口相连的菜单句柄在窗口摧毁的时候会由Windows自动释放，不需要手工释放，但没有和窗口相连的菜单就要由程序自己来释放了，方法是使用DestroyMenu函数，比如没有和窗口相连而仅用TrackPopupMenu弹出的菜单句柄：

invoke DestroyMenu,hMenu

## 5.2 图标和光标

### 5.2.1 图标和光标的资源定义

和菜单、加速键等资源不同，在资源脚本文件中定义图标和光标时并不是一个个像素地定义，而是指定图标和光标的文件名，由资源编译器将像素数据读入再转换成二进制格式，所以在资源定义之前要用其他工具先创建图标和光标文件。图标和静态光标文件的扩展名分别是ico和cur，还有一种扩展名为ani的动态光标文件。

光标和图标在资源文件中的定义语句是：

图标ID ICON [DISCARDABLE] 图标文件名 ；定义图标

光标ID CURSOR [DISCARDABLE] 光标文件名 ；定义光标

DISCARDABLE关键字是内存选项，表示在不用的时候可以从内存暂时卸掉，当文件名包含空格时，两边要用双引号引起来，图标ID和光标ID同样也可以用16位的整数或字符串表示，这里是几个定义的例子：

MyIcon icon “1.ico” ；把1.ico定义为ID为“MyIcon”的图标资源

1000 icon discardable 2.ico ；把2.ico定义为ID为1000的图标资源

1001 icon “big icon.ico” ；把big icon.ico定义为ID为1001的图标资源

1002 cursor “big arrow.ani” ；把big arrow.ani定义为ID为1002的光标资源

GoodCursor cursor arrow.cur ；把arrow.cur定义为ID为“GoodCursor”的光标资源

|  |
| --- |
| 灯泡.BMP资源文件中定义的图标可以不止一个，但Windows在“我的电脑”中列出文件的时候总是使用资源中的第一个图标当做文件的图标，所以在资源脚本文件中要把想用做程序图标的图标定义语句排在最前面。 |

**1. 装入图标和光标**

在WM\_CREATE消息中，程序从资源节区中装入所有的图标和光标资源，装入图标是用LoadIcon函数来完成的：

invoke LoadIcon,hInstance,lpIconName

.if eax

mov hIcon,eax

.endif

hInstance参数指定实例句柄，表示图标资源定义在哪个可执行文件中，lpIconName参数指定图标资源的名称，它就是资源文件中定义的图标ID值，如果调用成功的话，函数返回图标句柄。

除了可以装入资源文件中定义的图标资源之外，当参数hInstance为NULL的时候，用LoadIcon还可以用预定义的lpIconName参数装入Windows预定义的图标。

装入光标的函数有两个。装入在资源中定义的光标的函数是LoadCursor，它的语法和LoadIcon几乎一样：

invoke LoadCursor,hInstance,lpCursorName

.if eax

mov hCursor,eax

.endif

LoadCursor的用法也和LoadIcon相似，lpCursorName是光标资源的ID，LoadCursor也可以用指定hInstance为NULL的办法装入Windows的预定义光标。

|  |
| --- |
| 灯泡.BMP读者可以注意到，预定义的图标和光标都是Windows系统中常用的，预定义图标常用在消息框中，预定义光标就是Windows鼠标属性中的光标。使用预定义图标和光标的好处是它们的形状会随着系统设置值的不同自动改变，如改变“控制面板”→“鼠标”→“指针”中的设置后，装入的光标会自动改变。 |

另一个光标装入函数是LoadCursorFromFile，这个函数从磁盘光标文件中装入光标：

invoke LoadCursorFromFile,lpCursorFileName

.if eax

mov hCursor,eax

.endif

在Windows 9x中，静态光标文件\*.cur既可以定义在资源文件中，也可以使用LoadCursorFromFile函数装入，但是动态光标文件\*.ani只能通过文件方式装入。在Windows 2000及XP中，两种光标文件都可以通过资源装入。为了在不同的操作系统上都可以使用，例子文件使用LoadCursorFromFile函数来装入动态光标文件。

**2. 使用图标和光标**

现在来看如何使用图标。图标一般使用在对话框中或者程序窗口的标题栏中，要在标题栏中设置图标可以用向窗口发送WM\_SETICON消息的办法实现：

invoke SendMessage,hWnd,WM\_SETICON,ICON\_BIG,hIcon

消息的wParam参数可以是ICON\_BIG或ICON\_SMALL，用来指定图标的分辨率为32×32还是16×16，Windows将大图标显示在ALT+TAB的对话框上，将小图标显示在标题栏。

要将窗口的光标设置为新的光标不能使用WM\_SETCURSOR，这个消息是通知窗口重新刷新光标而不是让它设定指定的光标。Windows中有个SetCursor函数可以用来设置窗口光标，但这只能将新的光标维持很短一段时间，因为当Windows向窗口重新发送WM\_SETCURSOR消息的时候，光标就会被设置为原来的样子，在第4章的实验中读者已经看到，WM\_SETCURSOR是最频繁的消息之一，所以SetCursor并不能用来永久地改变窗口的光标。

如果要改变窗口的光标，正确的办法是用SetClassLong函数改变窗口类的属性，这个函数的使用方法如下：

invoke SetClassLong,hWnd,nIndex,dwNewLong

这个函数用来改变窗口类的属性，所以可以改变类中的光标设定，hWnd用来指定一个用这个类建立的某个窗口句柄，nIndex参数指定要改变窗口类的哪个属性，可以指定为GCL\_HBRBACKGROUND，GCL\_HCURSOR，GCL\_HICON，GCL\_HMODULE，GCL\_MENUNAME，GCL\_STYLE或GCL\_WNDPROC等，它们分别表示要改变的窗口类的背景色、光标、图标、hInstance、菜单、风格或窗口过程地址，读者可以用这个函数来改变一个窗口类的所有属性，程序中通过这个函数将窗口的光标在不同的光标句柄之间切换：

invoke SetClassLong,hWnd,GCL\_HCURSOR,hCur1或hCur2

## 5.3 位图

### 5.3.1 位图简介

位图（Bitmap）是Windows操作系统存储图像的方式，图像中的每个像素对应存储器中的一个或多个数据位，如单色位图每个像素对应1位，16色位图每个像素对应4位，256色为8位，全彩色为24位等，所有的像素数据按照一行行的顺序排列在存储器中，每个像素对应的位数称为颜色深度。

对于存放在磁盘上或别的地方的位图数据来说，它的颜色深度有可能和屏幕颜色深度不同，为了准确描述它的颜色信息，必须有像素数据的属性说明以及色彩表，在使用这个位图的时候，可以根据这些信息将像素数据转换到需要的颜色深度。色彩表和位图数据合在一起就叫做设备无关位图（DIB），因为它转换后可以用在不同颜色深度的设备上。Windows有函数专门用来处理DIB。使用DIB惟一的问题是当将高颜色数的DIB转换到低颜色数的设备 上时，由于色彩只能被转换成设备所能表示的最相近的颜色，所以可能会有很大的颜色失真。

DIB可以存放在磁盘上的位图文件中，位图文件一般以bmp为扩展名，它的内容包括一个bitmap文件头和DIB数据，bitmap文件头可以用来验证整个文件的有效性。所以简单地讲，DIB是位图数据的超集，位图文件又是DIB的超集。

### 5.3.2 在资源中定义位图

Windows对bmp文件的支持有两种方法，一种是打开bmp文件读入DIB部分的数据，然后用函数将DIB数据转换到位图数据；另一种方法就是在资源文件中用和ico，cur文件类似的方法定义位图资源，然后在程序中装入后使用。

在资源脚本文件中定义位图资源的语法是：

位图ID BITMAP [DISCARDABLE] 位图文件名

在程序中可以用LoadBitmap函数装入位图资源：

invoke LoadBitmap,hInstance,lpBitmapName

.if eax

mov hBitmap,eax

.endif

LoadBitmap函数返回一个位图句柄，在程序退出的时候，位图句柄必须用DeleteObject函数释放。对位图资源的大部分操作涉及GDI的内容，这在第7章中再详细介绍。

## 5.4 对 话 框

### 5.4.1 对话框简介

顾名思义，对话框完成的是“对话”功能，应用程序一般建立一个主窗口用做工作界面，大部分的工作会在主窗口的客户区完成，但程序往往需要和用户交互，如输入参数和输入文本等，这些界面不必要全部放在主窗口中，习惯的做法是通过选择菜单项弹出一个窗口，然后在这个窗口中完成对话，这个窗口就是“对话框”，对话框中的按钮、文本框和图标等称为“子窗口控件”。

为了提示用户，习惯于在会引出对话框的菜单项后面加上省略号。如“文件”菜单中的“另存为...”表示会引出一个选择文件名的对话框，所以“另存为”3个字后面加了个省略号。对话框最典型的例子就是写字板“查找”菜单弹出的窗口以及IE浏览器中选择“Internet选项”菜单项弹出的设置窗口。

**1. 对话框的类型**

对话框分两类：modal对话框和modeless对话框，翻译成中文就是“模态的”和“非模态的”，它们之间的区别在于是否允许用户在不同窗口间进行切换：当显示非模态对话框时，用户可以随意在这个对话框和其他窗口之间切换；而显示一个模态对话框时，用户在关闭对话框之前不允许切换到同一程序的其他窗口中，但可以切换到其他程序的窗口中（还有一种很少用的模态对话框，称为“系统模态”的对话框，比如关机时出现的那个对话框，这时切换到其他任何程序的窗口都是不允许的）。

Windows在资源文件中定义对话框，然后在程序中利用这个模板创建对话框，模态对话框和非模态对话框的资源定义是相同的，只是创建时调用的函数不同而已。

**2. 对话框的工作原理**

图5.4显示了对话框和普通窗口之间的不同之处。普通的窗口在建立之前需要用RegisterClass注册一个窗口类，然后用CreateWindow建立窗口；建立对话框的时候并不使用CreateWindow函数，取而代之，建立模态对话框的函数是DialogBoxParam，建立非模态对话框的函数是CreateDialogParam，调用这两个函数创建对话框窗口之前不需要注册对话框的窗口类。

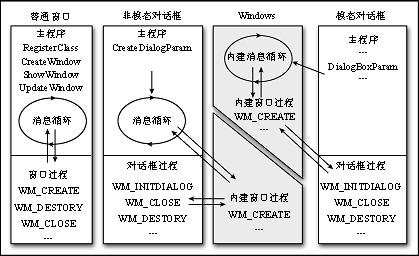


图5.4 对话框和普通窗口工作方式的区别

Windows在这两个函数的内部调用CreateWindowEx来建立对话框，使用的风格、大小和位置等参数取自资源中定义的对话框模板，使用的窗口类则是Windows内部定义的类，如果读者用一些工具去查看，会发现类名是“＃32770”之类的字符串，在这个名字奇特的窗口类中，窗口过程被定义到了Windows内部的“对话框管理器”代码中，Windows在这里处理对话框的大部分消息，如维护客户区的刷新，键盘接口（按Tab键在不同子窗口之间切换、按回车调用默认按钮等），对话框管理器在初始化对话框时会根据对话框模板中定义的子窗口控件建立对话框中所有的子窗口。

用户程序中的对话框过程是由对话框管理器调用的，在处理消息前，对话框管理器会先调用用户指定的对话框过程，再根据对话框过程的返回值决定是否处理它们。

Windows对模态对话框和非模态对话框的处理有些不同。在创建并显示模态对话框后，Windows会为它在内部建立一个消息循环，在这个消息循环中把消息发送给对话框管理器，对话框管理器在处理消息的过程中会调用用户定义的对话框过程，当对话框关闭的时候，Windows退出内建的消息循环，并从DialogBoxParam函数返回。而对于非模态对话框，CreateDialogParam函数在创建对话框后直接返回，对话框窗口的消息是通过用户程序中的消息循环派送的。

由于模态对话框的特征，使得用它来做小程序的主窗口非常方便，因为用一句DialogBoxParam函数就可以搞定了，既不用注册窗口类，也不用写消息循环，这对看到创建窗口的几十句代码就烦的读者来说可真是个福音，笔者也很喜欢用模态对话框做程序的主窗口。这种方法的缺点就是无法使用依赖消息循环来完成的功能，很明显，加速键就不能用了。

在接下来的内容中，以一个最简单的例子来讲解如何实现模态对话框，运行的结果如图5.5所示。

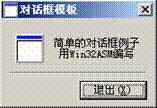


图5.5 例子对话框的运行结果

### 5.4.2 对话框的资源定义

**1. 对话框资源定义的语法**

在资源脚本中定义对话框的语法是：

对话框ID DIALOG [DISCARDABLE] x坐标, y坐标, 宽度, 高度

[可选属性]

BEGIN

子窗口控件

...

END

对话框中的子窗口控件语句定义在BEGIN/END（当然也可以用花括号）之中，在这之前，可以定义对话框的一些可选属性，每种属性单独用一行定义，常用的可选属性如表5.3所示。

表5.3 对话框的可选属性

|  |  |  |
| --- | --- | --- |
| 属性 | 定义语法 | 说明 |
| 标题文字  窗口类  窗口风格  扩展风格  字体  菜单 | CAPTION "文字"  CLASS "类名"  STYLE 风格组合  EXSTYLE 风格组合  FONT 大小，"字体名"  MENU 菜单ID | 定义显示在窗口标题栏上的文字  定义对话框窗口使用的窗口类，如果不定义，则使用Windows内建的类  定义对话框的窗口风格，同CreateWindowEx中的dwStyle参数  定义对话框的扩展窗口风格，同CreateWindowEx中的dwExStyle参数  定义对话框包括子窗口控件使用的字体  对话框中使用的菜单，菜单ID在同一个资源脚本文件中定义 |

在本节的例子中，资源脚本文件Dialog.rc是这样定义的：

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

#include <resource.h>

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

#define ICO\_MAIN 0x1000 //图标

#define DLG\_MAIN 1

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

ICO\_MAIN ICON "Main.ico"

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

DLG\_MAIN DIALOG 50, 50, 113, 64

STYLE DS\_MODALFRAME | WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU

CAPTION "对话框模板"

FONT 9, "宋体"

BEGIN

ICON ICO\_MAIN, -1, 10, 11, 18, 21

CTEXT "简单的对话框例子\n用Win32ASM编写", -1, 36, 14, 70, 19

DEFPUSHBUTTON "退出(&X)", IDOK, 58, 46, 50, 14

CONTROL "", -1, "Static", SS\_ETCHEDHORZ | WS\_CHILD | WS\_VISIBLE, 6, 39, 103, 1

END

脚本文件中除了定义图标以外，另外还定义了一个ID为1的对话框，对话框中有4个子窗口控件，分别是图标、文本、按钮和一个横线，按钮的ID为IDOK，其他的子窗口控件由于是静态控件，不会向对话框过程发送命令，所以ID就设置为−1，这些控件的具体用法将在后面的内容中详细介绍。

定义中还指定了一些可选属性，STYLE语句定义了对话框窗口的风格，CAPTION语句把标题定义为“对话框模板”，FONT语句指定了对话框使用的字体是大小为9的宋体。

对话框的位置为（50，50），大小为宽113单位、高64单位，读者可能已经注意到：这个对话框的大小好像比宽113像素、高64像素的窗口要大，事实上的确如此，这也正是大小是“单位”而不是“像素”的原因。对话框的位置、大小以及所有子窗口控件的度量单位是根据系统字体的大小来决定的，横向（x坐标和宽度）每单位为系统字符平均宽度的1/4，纵向（y坐标和高度）每单位为字符平均高度的1/8，由于系统字体的字符高度大致为宽度的两倍，所以虽然这种计算方法有些费解，但横向和纵向的数值在视觉上还是成比例的，但和以“像素”为单位在数值上肯定是不同的。如果读者一定要知道这个值换算成像素后是多少，那么可以用GetDialogBaseUnits函数来获取系统字体的高度和宽度再进行计算。

|  |
| --- |
| 灯泡.BMP当一些英文版的软件在中文Windows上运行的时候，对话框中有些文本往往被砍掉了尾巴，原因就是这些程序是在英文Windows上调试的，文本框的尺寸是以英文Windows系统字符的大小来度量的，到了其他语言的Windows上后，系统字符的大小可能改变，对话框的大小也随着改变，结果就是原来刚好的宽度可能会变得不够，这也算是对话框尺寸度量方法的缺点吧！  使用文本编辑器直接书写对话框脚本定义不是很直观，所以在创建对话框资源时最好使用可视化的资源编辑器，如VC++或ResourceWorkshop等。 |

在子窗口控件的ID定义中有两个特殊的ID值——IDOK和IDCANCEL，在Resource.h中它们的值定义为1和2，IDOK是默认的“确定”ID，IDCANCEL是默认的“取消”ID。如果一个按钮的ID是IDOK，当焦点没有停留在其他按钮上的时候，在任何地方按下回车键就相当于按下了这个按钮，而按下Esc键的时候，就相当于按下了ID为IDCANCEL的按钮。

**2. Tab停留位和组**

对话框中可以定义多个子窗口控件，有的子窗口控件可以拥有输入焦点（如按钮、文本框与组合框等），有些则不能（如图标与文本等），当对话框中有多个允许拥有输入焦点的子窗口控件时（有WS\_TABSTOP风格），用户可以用Tab键将输入焦点切换到下一个有WS\_TABSTOP风格的子窗口控件上，也可以用Shift＋Tab键切换到上一个，Tab键切换的顺序就叫做Tab停留位。

Tab停留位并不是系统根据子窗口控件的坐标位置自动排列的，而是按照子窗口控件在资源脚本文件中的定义顺序来排列的，所以读者在定义的时候最好根据子窗口控件的位置适当排列语句的先后，以免按动Tab键切换的时候焦点上下左右无规则地跳来跳去。如果使用可视化的资源编辑器，那么菜单中一般会有“Tab停留位”菜单项，在编辑完成后也要进到这个菜单项中设置一下，资源编辑器会根据设置调整rc文件中定义语句的先后顺序。

对话框中往往有一些排列在一起的同类子窗口控件，如几个单选钮，几个单选钮之间的选中标记是互斥的，在对话框的其他地方可能又有一组互斥的单选钮用来代表其他功能，在对话框中规定所有的单选钮都是互斥的显然不现实，解决的方法就是将不同的子窗口控件 “分组”，这就是“组”的含义。使用中可以选择一些子窗口控件定义WS\_GROUP属性，两个有WS\_GROUP属性的子窗口控件之间的所有子窗口控件同属同一组。

### 5.4.3 使用对话框

使用对话框的代码分为创建部分和对话框过程两个部分。先看Dialog.asm的源代码，再分析具体的使用过程，源代码如下：

.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

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; Equ 等值定义

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

ICO\_MAIN equ 1000h ;图标

DLG\_MAIN equ 1

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; 数据段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.data?

hInstance dd ?

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; 代码段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.code

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

\_ProcDlgMain proc uses ebx edi esi hWnd,wMsg,wParam,lParam

mov eax,wMsg

.if eax == WM\_CLOSE

invoke EndDialog,hWnd,NULL

.elseif eax == WM\_INITDIALOG

invoke LoadIcon,hInstance,ICO\_MAIN

invoke SendMessage,hWnd,WM\_SETICON,ICON\_BIG,eax

.elseif eax == WM\_COMMAND

mov eax,wParam

.if ax == IDOK

invoke EndDialog,hWnd,NULL

.endif

.else

mov eax,FALSE

ret

.endif

mov eax,TRUE

ret

\_ProcDlgMain endp

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

start:

invoke GetModuleHandle,NULL

mov hInstance,eax

invoke DialogBoxParam,hInstance,DLG\_MAIN,NULL,offset \_ProcDlgMain,NULL

invoke ExitProcess,NULL

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

end start

读者可以发现，相对于普通窗口的使用，对话框的使用显得特别简单，最明显的区别在于主程序中的一大堆代码不见了，换成了一个DialogBoxParam语句。

**1. 创建模态对话框**

创建模态对话框的函数是DialogBoxParam，它的使用方法是：

invoke DialogBoxParam,hInstance,lpTemplateName,hWndParent,\

lpDialogFunc,dwInitParam

函数的各参数说明如下：

● hInstance和lpTemplateName—函数从hInstance参数指定的模块中装入lpTemplateName参数指定的对话框资源，然后显示对话框窗口。例子程序中的lpTemplateName参数用的就是我们定义的DLG\_MAIN。

● hWndParent——对话框的父窗口，对话框关闭之前将无法切换到父窗口所属的其他窗口中，例子中用对话框做主窗口，所以父窗口句柄是NULL，在其他程序中使用时，这个参数设置为主窗口的句柄。

● lpDialogFunc——指定了对话框过程的地址，例子程序中是\_ProcDlgMain。

● dwInitParam——当做WM\_INITDIALOG消息的lParam传给对话框过程，读者可以用它来做自定义的用途。

要结束模态对话框，必须在对话框过程的WM\_CLOSE消息中使用EndDialog函数：

invoke EndDialog,hDlg,dwResult

不能使用通常的DestroyWindow函数，参数中的hDlg就是对话框窗口的句柄，dwResult参数是退出时的返回值，这个值最后由DialogBoxParam函数返回到主程序中。

**2. 创建非模态对话框**

创建非模态对话框的函数是CreateDialogParam，它的参数定义和DialogBoxParam一模一样：

invoke CreateDialogParam,hInstance,lpTemplateName,hWndParent,\

lpDialogFunc,dwInitParam

mov hDlg,eax

CreateDialogParam和DialogBoxParam在使用中有几个不同点：

● CreateDialogParam在创建对话框后，会根据对话框模板的风格是否定义了WS\_VISIBLE来决定是否显示对话框窗口。如果定义了则显示，没有的话，则程序需要在以后自行调用ShowWindow来显示它；而DialogBoxParam函数不管是否定义了WS\_VISIBLE风格都会显示对话框。

● CreateDialogParam在建立对话框窗口后直接返回，返回值是对话框窗口的句柄；而DialogBoxParam要在对话框关闭后才返回，返回值是EndDialog中的dwResult参数。

● 在CreateDialogParam返回后，应用程序在自己的消息循环中获取对话框消息，所以如果要用非模态对话框做程序的主窗口，消息循环的代码还是要写的；而DialogBoxParam是使用Windows为它内建的消息循环。

● 关闭非模态对话框使用DestroyWindow函数，注意在这里不要用EndDialog函数。

**3. 对话框过程**

Windows在“对话框管理器”——也就是为对话框内建的窗口过程中处理对话框消息，在处理前会首先调用用户定义的对话框过程，程序可以在这里选择是否自行处理某些消息。读者在理解时可以把“对话框管理器”看成是对话框的DefWindowsProc，凡是自己不想处理的消息都由它来处理。

和窗口过程一样，对话框过程是一个“回调”子程序，它由程序定义，Windows来调用，模态对话框和非模态对话框的对话框过程是一样的。

对话框过程和窗口过程的输入参数是一样的，也是：

DialogProc proc hwndDlg,uMsg,wParam,lParam

在程序里面一般编写对话框过程的分支结构如下：

\_ProcDlgMain proc uses ebx edi esi hWnd,wMsg,wParam,lParam

mov eax,wMsg

.if eax == WM\_CLOSE

；模态对话框用EndDialog关闭

；非模态对话框用DestroyWindow关闭

.elseif eax == WM\_INITDIALOG

；初始化代码

.elseif eax == WM\_COMMAND

；子窗口控件发送的消息

；wParam的低16位为子窗口控件ID

.elseif eax == WM\_XXXX

；处理其他需要处理的消息

.else

mov eax,FALSE

ret

.endif

mov eax,TRUE

ret

\_ProcDlgMain endp

注意对话框过程和普通的窗口过程在使用上有以下区别：

● 窗口过程对应于不同的消息有各种不同含义的返回值，而对话框过程返回BOOL类型的值，返回TRUE表示已经处理了某条消息，返回FALSE表示没有处理。“对话框管理器”代码会根据返回值决定是否继续处理某一条消息（惟一的例外是WM\_INITDIALOG消息）。

● 对于不处理的消息，不需要调用DefWindowProc来处理，这事情由“对话框管理器”来做。

“对话框管理器”不会把WM\_CREATE消息转发给对话框过程，取而代之，它会以WM\_INITDIALOG消息来调用对话框过程，程序可以在这里进行一些初始化的操作，WM\_INITDIALOG消息的返回值有点特殊，如果程序想自行设置输入焦点，那么可以用SetFocus函数把输入焦点设置到需要的子窗口控件上，然后返回FALSE；如果返回TRUE的话，那么Windows会自动将输入焦点设置到第一个有WS\_TABSTOP的子窗口控件上。

对话框过程在WM\_COMMAND消息中处理子窗口控件发送的命令，当用户在对话框中按下了按钮，输入文字或选择复选框等操作时，子窗口控件会向对话框过程发送WM\_COMMAND消息，wParam是子窗口控件的ID，如例子程序中处理“退出”按钮的消息，在里面用EndDialog函数关闭对话框。

对话框窗口的标题栏上默认没有定义图标，如果要像普通窗口一样显示一个图标，那么可以像例子程序中那样，在WM\_INITDIALOG中用WM\_SETICON消息来设置。

### 5.4.4 在对话框中使用子窗口控件

子窗口控件是一些Windows预定义类，它们实际上就是一个个以对话框为父窗口的子窗口。对于程序员来说，在对话框中使用它们的时候并不需要手工去逐一创建，只需要在对话框中定义就可以了，“对话框管理器”会在初始化对话框的时候，根据定义语句自动建立所有的子窗口。

**1. 子窗口控件的定义**

子窗口控件定义的一般语法是：

CONTROL 文本,ID,类,风格, x, y,宽度,高度[,扩展风格]

“文本”指控件的初始化值，“ID”是子窗口向对话框过程发送WM\_COMMAND中用的ID值，“类”可以是按钮（Button）、静态（Static）、编辑（Edit）、滚动条（ScrollBar）、列表框（ListBox）和组合框（ComboBox），这些类都是Windows系统中已经预定义的，“对话框管理器”在初始化的时候把每一条控件定义语句转换成下面的CreateWindow命令：

invoke CreateWindow，类名，文本，风格, x，y，宽度，高度，\

对话框窗口句柄，ID，hInstance，NULL

正因为如此，所有可以用CreateWindow建立的子窗口都可以在资源中定义，只要知道要使用的类和风格就可以了。所以除了上面这些基本的类之外，对话框中还可以使用一些通用控件，如“日期”（SysDateTimePick32）、“月历”（SysMonthCal32）、“热键”（msctls\_hotkey32）和“列表”（SysListView32）等，括号内是它们的类名，只要把定义语句的“类”写成对应的名称就可以了。

基于同一个预定义类的控件根据风格属性的不同，外表可能完全不同，如单选钮、复选框和分组框使用的类都是Button类，文本、图标框、位图框和线条等都是Static类。使用CONTROL语句定义的时候可能不是很直观，所以Rc.exe资源编译器允许使用另一种语法来书写控件定义：

控件名称 [文本,] ID,x,y,宽度,高度[,风格][,扩展风格]

这里使用“控件名称”而不是“类”是因为这个名称只是Rc.exe使用的缩写，并不是真正的Windows类的名称，“控件名称”由Rc.exe解释成“类”名，同时为它使用了几种默认的风格，定义语句中风格属性实际上是附加在默认风格上的，表5.4列出了每种控件使用的类和默认属性，除了表中列出的默认属性外，每种控件还被默认定义了WS\_CHILD和WS\_VISIBLE属性。

表5.4 资源脚本中使用的控件名称

|  |  |  |  |
| --- | --- | --- | --- |
| 控件名称 | 说明 | 基于的类 | 默认窗口风格 |
| PUSHBUTTON  DEFPUSHBUTTON  CHECKBOX  AUTOCHECKBOX  STATE3  AUTO3STATE  RADIOBUTTON  AUTORADIOBUTTON  GROUPBOX  SCROLLBAR  CTEXT  LTEXT  RTEXT  ICON  EDITTEXT  COMBOBOX  LISTBOX | 按钮  默认按钮  复选框  自动复选框  3态复选框  自动3态复选框  单选钮  自动单选钮  分组框  滚动条  居中文本  左对齐文本  右对齐文本  图标框  文本编辑  组合框  列表框 | Button  Button  Button  Button  Button  Button  Button  Button  Button  ScrollBar  Static  Static  Static  Static  Edit  ComboBox  ListBox | BS\_PUSHBUTTO，WS\_TABSTOP  BS\_DEFPUSHBUTTON，WS\_TABSTOP  BS\_CHECKBOX，WS\_TABSTOP  BS\_AUTOCHECKBOX，WS\_TABSTOP  BS\_3STATE，WS\_TABSTOP  BS\_AUTO3STATE，WS\_TABSTOP  BS\_RADIOBUTTON，WS\_TABSTOP  BS\_AUTORADIOBUTTON，WS\_TABSTOP  BS\_GROUPBOX  SBS\_HORZ  SS\_CENTER，WS\_GROUP  SS\_LEFT，WS\_GROUP  SS\_RIGHT，WS\_GROUP  SS\_ICON  ES\_LEFT，WS\_BORDER，WS\_TABSTOP  CBS\_SIMPLE，WS\_TABSTOP  LBS\_NOTIFY，WS\_BORDER |

看下面的例子：

GROUPBOX "选项", -1, 55, 5, 120, 100

PUSHBUTTON "退出", IDCANCEL, 255, 115, 50, 14

这两条语句和下面的语句编译后产生的二进制资源文件是一模一样的：

CONTROL "选项", -1, "Button", BS\_GROUPBOX | WS\_TABSTOP, 55, 5, 120, 100

CONTROL "退出", IDCANCEL, "Button", BS\_PUSHBUTTON | WS\_TABSTOP, 255, 115, 50, 14

第一种语句的用法比第二种语句不但要直观许多，而且不必书写默认的窗口风格。

当用到的控件没有缩写语法时，那就必须用CONTROL定义了，下面的两句分别定义了一条横线和一个图片框，它们并没有缩写的用法：

CONTROL "", -1, "Static", SS\_ETCHEDHORZ | WS\_CHILD | WS\_VISIBLE, 60, 65, 110, 1

CONTROL BMP\_ID, -1, "Static", SS\_BITMAP | WS\_CHILD | WS\_VISIBLE, 5, 5, 40, 95

下面以一个例子来演示各种子窗口控件的用法，其中的Control.rc文件如下：

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

#include <resource.h>

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

#define ICO\_MAIN 0x1000 //图标

#define DLG\_MAIN 1

#define IDB\_1 1

#define IDB\_2 2

#define IDC\_ONTOP 101

#define IDC\_SHOWBMP 102

#define IDC\_ALOW 103

#define IDC\_MODALFRAME 104

#define IDC\_THICKFRAME 105

#define IDC\_TITLETEXT 106

#define IDC\_CUSTOMTEXT 107

#define IDC\_BMP 108

#define IDC\_SCROLL 109

#define IDC\_VALUE 110

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

ICO\_MAIN ICON "Main.ico"

IDB\_1 BITMAP "Picture1.bmp"

IDB\_2 BITMAP "Picture2.bmp"

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

DLG\_MAIN DIALOG 193, 180, 310, 134

STYLE DS\_MODALFRAME | WS\_POPUP | WS\_VISIBLE | WS\_CAPTION | WS\_SYSMENU | WS\_THICKFRAME

CAPTION "对话框子窗口控制示例"

FONT 9, "宋体"

{

GROUPBOX "选项", -1, 55, 5, 120, 100

AUTOCHECKBOX "总在最前面", IDC\_ONTOP, 65, 20, 100, 12

AUTOCHECKBOX "显示图片", IDC\_SHOWBMP, 65, 35, 100, 12

AUTOCHECKBOX "允许更换图片", IDC\_ALOW, 65, 50, 100, 12

CONTROL "", -1, "Static", SS\_ETCHEDHORZ | WS\_CHILD | WS\_VISIBLE, 60, 65, 110, 1

AUTORADIOBUTTON "模态边框(&Modal Frame)", IDC\_MODALFRAME, 65, 70, 100, 12, WS\_TABSTOP

AUTORADIOBUTTON "可变边框(&Thick Frame)", IDC\_THICKFRAME, 65, 85, 100, 12, WS\_TABSTOP

GROUPBOX "标题栏文字", -1, 180, 5, 125, 100, BS\_GROUPBOX

COMBOBOX IDC\_TITLETEXT, 190, 20, 105, 70, CBS\_DROPDOWNLIST | WS\_TABSTOP

LTEXT "自定义文字：", -1, 190, 40, 105, 10

EDITTEXT IDC\_CUSTOMTEXT, 190, 55, 105, 12

LTEXT "请在此选择显示在标题栏上面的文字，或者选择“自定义”后自行输入", -1, 191, 73, 105, 26, WS\_BORDER

CONTROL "", -1, "Static", SS\_ETCHEDHORZ | WS\_CHILD | WS\_VISIBLE, 5, 110, 300, 1

DEFPUSHBUTTON "更换图片(&C)", IDOK, 200, 115, 50, 14

PUSHBUTTON "退出(&X)", IDCANCEL, 255, 115, 50, 14

CONTROL IDB\_1, IDC\_BMP, "Static", SS\_BITMAP | WS\_CHILD | WS\_VISIBLE, 5, 5, 40, 95

SCROLLBAR IDC\_SCROLL, 6, 118, 125, 10

LTEXT "0", IDC\_VALUE, 138, 119, 50, 8

}

//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

编译后的Control.exe运行后的界面如图5.6所示。

图5.6中的子窗口控件分别是GROUPBOX①，AUTOCHECBOX②，SS\_ETCHEDHORZ风格的Static类③，AUTORADIOBUTTON④，COMBOBOX⑤，EDITTEXT⑥，WS\_BORDER风格LTEXT⑦，DEFPUSHBUTTON⑧，SS\_BITMAP风格的Static类⑨和滚动条⑩。



图5.6 子窗口控件使用的例子

程序有这些功能：按下“更换图片”按钮⑧可以切换图片框⑨的图片；在组合框⑤中可以选择更换标题栏的文字，选“自定义”的时候可以激活文本编辑框⑥并输入自定义文字；默认状态下对话框是可以调整大小的，如果将单选钮④切换到“模态边框”，那么大小就无法调整。Control.asm源代码如下：

.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 gdi32.inc

includelib gdi32.lib

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; Equ 等值定义

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

ICO\_MAIN equ 1000h

DLG\_MAIN equ 1

IDB\_1 equ 1

IDB\_2 equ 2

IDC\_ONTOP equ 101

IDC\_SHOWBMP equ 102

IDC\_ALOW equ 103

IDC\_MODALFRAME equ 104

IDC\_THICKFRAME equ 105

IDC\_TITLETEXT equ 106

IDC\_CUSTOMTEXT equ 107

IDC\_BMP equ 108

IDC\_SCROLL equ 109

IDC\_VALUE equ 110

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; 数据段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.data?

hInstance dd ?

hBmp1 dd ?

hBmp2 dd ?

dwPos dd ?

.const

szText1 db 'Hello, World!',0

szText2 db '嘿，你看到标题栏变了吗?',0

szText3 db '自定义',0

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

; 代码段

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

.code

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

\_ProcDlgMain proc uses ebx edi esi hWnd,wMsg,wParam,lParam

local @szBuffer[128]:byte

mov eax,wMsg

.if eax == WM\_CLOSE

invoke EndDialog,hWnd,NULL

invoke DeleteObject,hBmp1

invoke DeleteObject,hBmp2

.elseif eax == WM\_INITDIALOG

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 设置标题栏图标

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

invoke LoadIcon,hInstance,ICO\_MAIN

invoke SendMessage,hWnd,WM\_SETICON,ICON\_BIG,eax

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 初始化组合框

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

invoke SendDlgItemMessage,hWnd,IDC\_TITLETEXT,CB\_ADDSTRING,0,addr szText1

invoke SendDlgItemMessage,hWnd,IDC\_TITLETEXT,CB\_ADDSTRING,0,addr szText2

invoke SendDlgItemMessage,hWnd,IDC\_TITLETEXT,CB\_ADDSTRING,0,addr szText3

invoke SendDlgItemMessage,hWnd,IDC\_TITLETEXT,CB\_SETCURSEL,0,0

invoke GetDlgItem,hWnd,IDC\_CUSTOMTEXT

invoke EnableWindow,eax,FALSE

invoke LoadBitmap,hInstance,IDB\_1

mov hBmp1,eax

invoke LoadBitmap,hInstance,IDB\_2

mov hBmp2,eax

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 初始化单选钮和复选框

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

invoke CheckDlgButton,hWnd,IDC\_SHOWBMP,BST\_CHECKED

invoke CheckDlgButton,hWnd,IDC\_ALOW,BST\_CHECKED

invoke CheckDlgButton,hWnd,IDC\_THICKFRAME,BST\_CHECKED

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 初始化滚动条

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

invoke SendDlgItemMessage,hWnd,IDC\_SCROLL,SBM\_SETRANGE,0,100

.elseif eax == WM\_COMMAND

mov eax,wParam

.if ax == IDCANCEL

invoke EndDialog,hWnd,NULL

invoke DeleteObject,hBmp1

invoke DeleteObject,hBmp2

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 更换图片

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif ax == IDOK

mov eax,hBmp1

xchg eax,hBmp2

mov hBmp1,eax

invoke SendDlgItemMessage,hWnd,IDC\_BMP,STM\_SETIMAGE,IMAGE\_BITMAP,eax

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 设置是否“总在最前面”

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif ax == IDC\_ONTOP

invoke IsDlgButtonChecked,hWnd,IDC\_ONTOP

.if eax == BST\_CHECKED

invoke SetWindowPos,hWnd,HWND\_TOPMOST,0,0,0,0,\

SWP\_NOMOVE or SWP\_NOSIZE

.else

invoke SetWindowPos,hWnd,HWND\_NOTOPMOST,0,0,0,0,\

SWP\_NOMOVE or SWP\_NOSIZE

.endif

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 演示隐藏和显示图片控件

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif ax == IDC\_SHOWBMP

invoke GetDlgItem,hWnd,IDC\_BMP

mov ebx,eax

invoke IsWindowVisible,ebx

.if eax

invoke ShowWindow,ebx,SW\_HIDE

.else

invoke ShowWindow,ebx,SW\_SHOW

.endif

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 演示允许和灰化“更换图片”按钮

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif ax == IDC\_ALOW

invoke IsDlgButtonChecked,hWnd,IDC\_ALOW

.if eax == BST\_CHECKED

mov ebx,TRUE

.else

xor ebx,ebx

.endif

invoke GetDlgItem,hWnd,IDOK

invoke EnableWindow,eax,ebx

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif ax == IDC\_MODALFRAME

invoke GetWindowLong,hWnd,GWL\_STYLE

and eax,not WS\_THICKFRAME

invoke SetWindowLong,hWnd,GWL\_STYLE,eax

.elseif ax == IDC\_THICKFRAME

invoke GetWindowLong,hWnd,GWL\_STYLE

or eax,WS\_THICKFRAME

invoke SetWindowLong,hWnd,GWL\_STYLE,eax

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 演示处理下拉式组合框

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif ax == IDC\_TITLETEXT

shr eax,16

.if ax == CBN\_SELENDOK

invoke SendDlgItemMessage,hWnd,IDC\_TITLETEXT,CB\_GETCURSEL,0,0

.if eax == 2

invoke GetDlgItem,hWnd,IDC\_CUSTOMTEXT

invoke EnableWindow,eax,TRUE

.else

mov ebx,eax

invoke SendDlgItemMessage,hWnd,IDC\_TITLETEXT,CB\_GETLBTEXT,ebx,addr @szBuffer

invoke SetWindowText,hWnd,addr @szBuffer

invoke GetDlgItem,hWnd,IDC\_CUSTOMTEXT

invoke EnableWindow,eax,FALSE

.endif

.endif

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 在文本框中输入文字

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif ax == IDC\_CUSTOMTEXT

invoke GetDlgItemText,hWnd,IDC\_CUSTOMTEXT,addr @szBuffer,sizeof @szBuffer

invoke SetWindowText,hWnd,addr @szBuffer

.endif

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; 处理滚动条消息

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.elseif eax == WM\_HSCROLL

mov eax,wParam

.if ax == SB\_LINELEFT

dec dwPos

.elseif ax == SB\_LINERIGHT

inc dwPos

.elseif ax == SB\_PAGELEFT

sub dwPos,10

.elseif ax == SB\_PAGERIGHT

add dwPos,10

.elseif ax == SB\_THUMBPOSITION || ax == SB\_THUMBTRACK

mov eax,wParam

shr eax,16

mov dwPos,eax

.else

mov eax,TRUE

ret

.endif

cmp dwPos,0

jge @F

mov dwPos,0

@@:

cmp dwPos,100

jle @F

mov dwPos,100

@@:

invoke SetDlgItemInt,hWnd,IDC\_VALUE,dwPos,FALSE

invoke SendDlgItemMessage,hWnd,IDC\_SCROLL,SBM\_SETPOS,dwPos,TRUE

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.else

mov eax,FALSE

ret

.endif

mov eax,TRUE

ret

\_ProcDlgMain endp

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

start:

invoke GetModuleHandle,NULL

mov hInstance,eax

invoke DialogBoxParam,hInstance,DLG\_MAIN,NULL,offset \_ProcDlgMain,NULL

invoke ExitProcess,NULL

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

end start

**2. 子窗口控件的通用使用方法**

由于子窗口控件实际上就是窗口，大部分窗口函数对它们都是适用的，如可以用EnableWindow在灰化和允许状态之间切换，可以用ShowWindow在显示和隐藏之间切换，可以用GetWindowText和SetWindowText来改变上面的文字，也可以用MoveWindow来改变大小和移动位置等。在Control.asm中用“显示图片”复选框切换图片框的隐藏和显示，用的就是ShowWindow函数，处理“允许更换图片”复选框时切换“更换图片”按钮的状态，用的是EnableWindow函数。

除了可以用对子窗口控件使用窗口的通用函数外，还可以使用针对它们的专用函数。下面介绍一些常用的函数。

在资源脚本文件中定义的是控件的ID，当这些子窗口控件被创建以后同样会有一个窗口句柄，但既然它们不是由我们自己创建的，那么怎么知道它们的窗口句柄呢？有一个函数可以从ID中获取子窗口句柄：

invoke GetDlgItem,hDlg,dwIDDlgItem

mov hDlgItem,eax

函数的输入参数是对话框句柄和ID值，返回值是子窗口句柄；反过来，有两种方法可以从子窗口句柄获取ID：

（1） invoke GetDlgCtrlID,hWndCtrl ；输入子窗口句柄，返回值是控件ID

（2） invoke GetWindowLong,hWndCtrl,GWL\_ID

当需要向控件发送消息的时候，当然可以先用GetDlgItem获取子窗口句柄再用SendMessage函数，但有一个函数更为简便：

invoke SendDlgItemMessage,hDlg,dwIDDlgItem,Msg,wParam,lParam

这个函数可以直接向控件发送消息，只需要在参数中指定对话框句柄和子窗口ID（注意：并没有PostDlgItemMessage这样的函数！）。

如果要想知道在一个控件上按下了Tab键或Shift＋Tab键会跳到哪一个控件上去，也就是说下一个或上一个Tab停留位在哪里，可以使用GetNextDlgTabItem函数：

invoke GetNextDlgTabItem,hDlg,hCtl,bPrevious

.if eax

mov hWinNext,eax

.endif

其中的bPrevious参数指定了搜索的方向；与之相似，使用GetNextDlgGroupItem 函数可以返回下一个分组的位置：

invoke GetNextDlgGroupItem,hDlg,hCtl,bPrevious

.if eax

mov hWinNext,eax

.endif

**2. 子窗口控件的通用使用方法**