华工松联电脑丛书

武汉松联公司
北 京 松 岗

从WIN3.1到WIN95
程序设计捷径

黄天浩 编著

华中理工大学出版社

·武汉松联环球电脑信息有限公司·北京松岗公司·

从WIN3.1到WIN95

程序设计捷径

黄天浩 编著

华中理工大学出版社

(鄂)新登字第 10 号

图书在版编目(CIP)数据

从 WIN31 到 WIN95 程序设计捷径/黄天浩编著
武汉:华中理工大学出版社,1997.6.
ISBN 7-5609-1530-2
Ⅰ从…
Ⅱ黄…
Ⅲ计算机软件—Win95 程序设计
Ⅳ TP316

从 WIN31 到 WIN95 程序设计捷径　　©黄天浩　编著

责任编辑:唐志宽等　　封面设计:梁书亭
责任校对:童兆丹　　监　印:张正林
出 版 者:华中理工大学出版社.　　(武汉市　邮编:430074)
发 行 者:华中理工大学出版社发行部　　(电话:(027)7547012)
印 刷 者:华中理工大学出版社印刷厂　　(邮编:430074)
开本:787×1092　1/16　印张:9　字数:200 000
版次:1997 年 6 月第 1 版　印次:1997 年 6 月第 1 次　印数:1-5 000
ISBN 7-5609-1530-2/TP・219　定价:19.00 元

序

许久以来，PC 机的操作系统一直在 16 位的领域中打转，虽然 Windows3.1 风靡全球，但是 Windows NT 这个 32 位的操作系统始终无法成为主流。Microsoft 公司为了引导全球电脑使用者接受 32 位 Windows NT 系统，隆重地推出了 Windows95。

Windows95 基本上是一个 32 位的过渡型产品，其目的在于让 Windows NT 可以得以顺利推广，由于 Windows95 对于硬件的需求较简单，因此，目前 Windows95 已经几乎成为 PC 操作系统的主流，全球软件公司都想尽办法推出 32 位的应用程序。要将一个庞大的应用程序由 16 位转换为 32 位并不是一项轻松的工作，程序设计人员必须先了解 16 位与 32 位 Windows 之间的差异，然后再逐步修改应用程序。

如何能够快速地将 16 位应用程序转换成 32 位应用程序，是许多程序设计师迫切想知道的。本书中，笔者尽量采用对比的方式，让每位读者了解 16 位与 32 位 Windows 之间的差异性，使人们可以快速了解如何修改现有的 Windows 应用程序。

本书虽然全部以 C 语言为探讨范例，但是对于 C++依然适用，即使是使用 Application framework，例如：MFC 或 OWL，本书也同样适用。对于 MFC 或 OWL 的程序而言，有许多问题已经被包装起来，但是无论如何，您的程序依旧会需要直接使用到一些 SDK 函数，所以，您的应用程序修改幅度会比使用 C 语言直接撰写的程序小，不过，您依旧必须了解 16 位与 32 位 Windows 之间的差异。

笔者十分感谢曾任教华夏工专电脑中心的陈张宗荣老师，及现任华夏工专电脑中心的林采端主任、汪忠光老师，他们当年的悉心教诲，使我能有今日之成就。还要感谢松岗电脑图书公司的伙伴们以及何志仁先生，由于他们的帮忙，使得本书得以顺利出版。

本书内容虽经多次校对，如有疏漏还请见谅。对本书的内容如有意见或疑问，可通过 internet 与笔者联系，笔者的 email address 为 emperor@cc. hwh. edu. tw

黄天浩

1996 年 10 月

出 版 说 明

本书中文繁体字版由台湾松岗电脑图书资料股份有限公司(下简称“松岗公司”)出版。本书中文简体字版经松岗公司授权由华中理工大学出版社出版。任何单位或个人未经出版者书面允许不得用任何手段复制或抄袭本书内容。

由于海峡两岸计算机科学技术术语的译名不太相同,因此在出版中文简体字版时对文中的术语进行了转译。转译工作是由唐志宽副教授等完成的。转译内容力求做到表述准确贴切。

在中文简体字版中,对原中文繁体字版中某些仅适合台湾地区的内容经征得松岗公司驻北京代表叶权荣先生同意后作了删节,对原版书中个别错字、漏字也作了更正。

本书在中文简繁转译工作过程中得到了有关同志的大力帮助,谨致衷心感谢。

华中理工大学出版社

1997 年 5 月

内容简介

本书采用对比的方法,详细地介绍了 16 位与 32 位 Windows 之间的差异,并通过 C 语言程序中的窗口过程函数、数据类型、存储器、系统函数、INI 文件格式及动态连接程序库等具体问题讨论了 16 位应用程序转换成 32 位应用程序的捷径。

本书内容新颖、全面、简明、实用,是广大程序设计者的良师益友。

目 录

第一章

窗口过程函数

1-0 引言

应用程序的原始码，从 16 位的 Windows3. x 版转换到 32 位的 Windows95、Win32s 或 Windows NT 的过程中，有关主程序(WinMain)、窗口处理程序(Window Procedure)或是对话框处理程序(Dialog box procedure)，乍看之下似乎没有改变，其实其中的许多内容均已经变了，只不过有些问题被包装起来，因此单从原始程序上并不容易发现其中的变化。

本章将详细剖析 16 位以及 32 位程序中有关主程序、窗口处理函数以及对话框处理函数的转变，以及这些转变将产生什么样的影响与困扰。

1-1 主程序

对 16 位的 Windows 程序而言，应用程序的主程序写法如下：

```
int PASCAL WinMain (
  HINSTANCE hInstance.          /★  handle for this instance          ★/
  HINSTANCE hprevInstance.      /★  handle for previous instances      ★/
  LPSTR     lpszCmdLine.        /★  pointer to exec command line       ★/
  int       nCmdShow )          /★  for main window display            ★/
```

对 32 位的 Windows 程序而言，应用程序的主程序写法如下：

```
int APIENTRY WinMain (
    HINSTANCE hInstance.
    HINSTANCE hprevInstance.  /★ always NULL ★/
    LPSTR     lpszCmdLine.
    int       nCmdShow);
```

1-1-1 函数调用法

首先，我们可以发现主程序有关函数调用法(calling convention)的定义方法有所改变，我们分别从 Windows SDk 3. 1 的 WINDOWS. H 以及 Win 32 SDK 的 WINDEF. H 文件中找出其明确定义方式，其内容如下：

对 16 位的 Windows 而言，PASCAL 定义如下：

```
//define PASCAL   -pascal        /★ pascal 调用法 ★/
```

对 32 位的 Windows 而言，APIENTRY 定义如下：

```
//define WINAPI   -stdcall /★ standrad 调用法 ??? ★/
//define APIENTRY WINAPI
```

从上面的定义方式中我们可以轻易了解，在 16 位的 Windows 环境中，WinMain 函数采用-pascal（pascal 调用法）；而在 32 位 Windows 环境中，WinMain 函数改用--stdcall（standard 调用法）。pascal 调用法许多读者都多少有些了解，但是什么叫 standard 调用法？为了让您可充分了解 C 语言的各种函数调用方式，将先介绍 pascal 及 C 语言调用法，再介绍 standard 调用法。什么叫"pascal 调用法"？熟悉程序语言的读者多半知道，这种调用法对于函数参数乃是采用"由前往后"推入堆栈的方式进行，而且被调用的函数已经明确获知参数的数目，因此在函数结束时，会一并回复堆栈的状态。下面的范例可让人们进一步了解 pascal 调用法的类型

```
int PASCAL PascalCalling (int iparam1, intiparam2)
{
        :
        ;
}
TestFunc ()
{
    int i,j;
    PascalCalling (i,j);
}
```

上述的程序片段由 32 位 C 编译器编译后，可得到类似以下的组合语言程序码(不重要的部分将予以忽略)：

```
PascalCalling PROC
        :
        ;
    RET   8          ;返回时堆栈一并还原
                     ;PS:32 位环境下,int 为 4bytes
PascalCalling ENDP

_TestFunc       PROC
        :
        ;
    push  i          ;参数是"由前往后"推入堆栈
    push  j
    call PascalCalling
        :            ;返回后不需再处理堆栈
        ;
_TestFunc       ENDP
```

了解 pascal 调用法后，我们先来回忆什么叫“C 语言调用法”？熟悉 C 语言的读者多半知道，在 Microsoft 的 C 编译器上，使用“_cdecl”定义这种调用法，当然，如果您没有定义调用法，C 编译器会自动采用 C 语言调用法。C 语言调用法对于函数参数的处理恰好与 pascal 调用法相反，它采用“由后往前”推入堆栈的方式进行，而且被调用的函数无法明确获知参数的数目，因此在函数结束时，不会恢复堆栈的状态。下面的范例可让您更了解 C 语言调用法的型态：

```
int _cdecl C_Calling (int iparam1, int iparam2)
{
        :
        :
}

TestFunc ()
{
      int i,j;

      C_Calling (i,j);
}
```

上述的程序片段由 32 位 C 编译器编译后，可得到类似以下的组合语言程序码(不重要的部分将予以忽略)：

```
_C_Calling PROC
        :
        :
      RET              ;返回时堆栈不会还原
_C_Calling ENDP
_TestFunc       PROC
        :
        :
      push j          ;参数是“由后往前”推入堆栈
      push i
      call _C_Calling
      add esp, 8      ;返回后再还原堆栈
                       ;PS:32 位环境下,int 为 4 bytes
        :
        :
_TestFunc          ENDP
```

了解了 pascal 及 C 语言调用法后，我们可以再进一步了解 32 位 Windows 所采用的 standard 调用法。其实 Windows 的应用程序虽然号称以 C 或 C++ 语言为主要开发工具，而且语法上也完全采用 C 及 C++ 语言之定义，但是许多使用习惯或当初 C 或 C++ 建议的函数库标准却没有被完全采用。例如：16 位 Windows 系统所提供的函数，几乎都是采用 pascal 调用法，而不是惯用的 C 语言调用法；32 位 Windows 采用 standard 调用法。基本上，所谓的 standard 调用法乃是一种融合 pascal 调用法及 C 语言调用法的一种函数调用方式，编译器会根据原型函数(prototype)的定义状态决定采用 pascal 还是 C 语言调用法。例如：

```
int _stdcall StdCalling1 (int iparam. iparam2. iparam3);
int _stdcall StdCalling2 (int iparam,…);

TestFunc()
{
    int i,j,k;

    StdCalling1 (i,j,k);
    StdCalling2 (i,j,k);
}
```

上述的程序片段由 32 位 C 编译器译编后，可得到类似以下的组合语言程序码(不重要的部分将予以忽略)：

```
_TestFunc     PROC
        :
        :
      push i          ;参数是"由前往后"推入堆栈
      push j
      push k
      call _StdCalling1
                      ;返回后不需再处理堆栈

      push k          ;参数是"由后往前"推入堆栈
      push j
      _push i
      call _StdCalling2
      add esp. 12     ;返回后再还原堆栈
                      ;PS:32 位环境下. int 为 4 bytes
        :
        :
 TestFunc       ENDP
```

从上面的例子我们可以得知，大部分的情形下，函数的参数数目若是固定的，编译器会采用 pascal 调用法，对于少数使用变动参数数目的函数而言，系统则使用 C 语言调用法。此种调用法的优点在于 pascal 会自动恢复堆栈状态，因此可以使程序码有效地变小；如果有使用变动参数的需要时，又可自动使用 C 语言调用法，保持程序设计的弹性。了解 C 语言所使用的各种函数调用法后，我们可以深刻了解原型函数定义在 Windows 程序中的重要性。缺乏原型函数定义的程序在 16 位的 Windows 环境中还不至于发生太大的困扰；但是在 32 位的 Windows 环境中，恐怕就有可能发生。从整体看来，原型函数定义对于 C++ 语言原本就是必备的，对 C 语言而言则是可有可无，虽然在 ANSI C 制定时已经将函数原型定义列为建议，但是对于具有丰富经验的 C 语言程序设计师而言，还是习惯采用不含原型函数定义的 K&R 标准。在此建议读者务必采用 ANSI C 的标准及建议，ANSI C 虽然对程序设计过程增加了一些负担，却可确保程序的可信度，并且对于未来的程序维护工作有所助益，具有绝对正面的意义。

32 位 Windows 改变了在 16 位 Windows 环境中使用最广的 pascal 调用法，是否会造成全球 Windows 应用程序的改版困扰呢？这点 Microsoft 已经考虑到了。因为 32 位 Windows 并不直接支援 pascal 调用法的定义，在 WINDEF. H 文件中，pascal 定义已经被改成 --stdcall 定义。下面我们来看 16 及 32 位 Windows 对于 pascal 定义的规定。

对 16 位的 Windows SDK，WINDOWS. H 定义如下：

```
//define PASCAL   pascal
```

对 32 位的 Windows SDK，WINDEF. H 定义如下：

```
//undef pascal
//define pascal     _stdcall
//define PASCAL     _stdcall
```

一个简单的定义式就解决了大部分的问题，Windows SDK 尽量隐藏 C 语言的原始数据型态，全面采用延伸定义的数据型态，虽然对于刚开始从事 Windows 应用程序设计的人存在适应性的问题，不过在操作平台的移转过程中，却可充分显示出其优点。

1-1-2 HINSTANCE 参数

在 16 位的 Windows 环境中，WinMain()函数的前两个参数分别代表目前及前一个事例的代码，而事例代码其实就是应用程序的数据段代码(handle)。在 32 位的 Windows 环境中，由于存储器的处理方式已经彻底改变，事例代码的含义已经不再是应用程序的数据段代码，而是由应用程序的代码表(handle table)所配置出来，其含意与模块代码(HMODULE)完全相同。事例代码的改变对少数应用程序会产生相当大的冲击，某些应用程序在发现前一个事例处于执行状态时，会透过 GetInstanceData()函数取得前一个事例的数据，这种作法在 32 位 Windows 环境下是行不通的。比较快的修改方式是不再向前一个事例取得数据，如果一定要取得前一个事例的数据，恐怕仅能改为采用动态数据交换管理模块(Dynamic Data Exchange Management Library，DDEML)或存储器映射文件(memory-map file)的方式解

```
/★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★/
/★ 在 16 位 Windows 环境中                              ★/
/★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★/
int PASCAL WinMain(
    HANDLE hInstance.       /★ handle for this instance         ★/
    HANDLE hprevInstance.   /★ handle for previous instances    ★/
    LPSTR lpszCmdLine.      /★ long pointer to command line     ★/
    int nCmdShow )          /★ Show code for window display     ★/
{
        :
        :
    if (hprevInstance)
    {
      /★ Active previous instance and shutdown this ★/
      GetInstanceData (hprevInstance. (NPSTR)&hWndMain.
                       sizeof(hWndMain));
      SetActiveWindow (hWndMain);
      ShowWindow (hWndMain. SW_RESTORE):
      return (FALSE);
    }
        :
        :
}
```

```
/★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★/
/★ 在 32 位 Windows 环境中                              ★/
/★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★/
int APIENTRY WinMain (
    HINSTANCE hInstance.
    HINSTANCE hPrevInstance.      /★★★ always is NULL ★★★/
    IPSTR lpszCmdLine.
int nCmdShow)
{
        :
        :
    hWndPrev=FindWindow (szClassName. szWindowName);
    if ( hWndPrev )
    {
      SetActiveWindow(hWndPrev);
      ShowWindow (hWndPrev. SW_RESTORE);
      return (FALSE);
    }
        :
        :
}
```

决。对于大部分的应用程序而言，要改变事例代码通常仅需要稍微改动程序。在 16 位的 Windows 环境中，应用程序可运用 WinMain()函数的 hPrevInstance 参数判别是否有其他

事例处于执行状态，如果应用程序不希望有多个事例同时执行，可运用 GetInstanceData() 函数，取得前一个事例的窗口代码(HWND)，将主控权交给前一个事例，并且结束目前的事例。这个作法在 32 位 Windows 环境中已经完全不适用，首先，hPrevInstance 参数已经不再具有任何意义，它永远是 NULL；其次，您无法再通过 GetInstanceData()取得前一个事例的任何数据，此函数已经被取消。下面的程序片段列出了从 16 位的 Windows 环境转换到 32 位的 Windows 环境的差异性，此程序片段会在使用者企图执行第二个程序事例时，调出第一个事例，并结束第二个事例。

从前面的程序片段中，我们可以得知，由于 32 位 Windows 不再支持 hPrevInstance 参数，因此程序改为使用搜寻窗口的方式找寻是否有前一个程序事例的存在，如果有，则将前一个事例的窗口设定为活动窗口，并且结束目前的应用程序。这种作法在 16 位 Windows 环境中也是可以用的，不必担心必须维护两套原始程序码。

```
/★★★★★★★★★★★★★★★★★★★★★/
/★   16 位 Windows SDK. 定义如下：   ★/
/★★★★★★★★★★★★★★★★★★★★★/
//define FAR                _far
//define NEAR               _near
typedef unsigned int      UINT;

//ifdef STRICT
typedef const void NEAR★      HANDLE;
//define DECLARE _HANDLE(name) struct name// //_{ int unused; };\
                        typedef const struct name// //_NEAR★name
//else   /★ STRICT ★/
typedef UINT
//define DECLARE _HANDLE(name)typedef UNIT name
//endif /★ ISTRICT ★/

DECLARE _HANDLE(HINSTANCE);
```

```
/★★★★★★★★★★★★★★★★★★★★★/
/★   32 位 Windows SDK. 定义如下：   ★/
/★★★★★★★★★★★★★★★★★★★★★/
typedef void★PVOID;

//ifdef STRICT
typedef void★HANDLE;
//define DECLARE _HANDLE(name)struct name// //_{ int unused; };\
                        typedef struct name// //_★name
//else
typedef PVOID HANDLE;
//define DECLARE _HANDLE(name)typedef HANDLE name
//endif

DECLARE _HANDLE(HINSTANCE);
```

HINSTANCE 除了内容在 16 与 32 位 Windows 环境中的含义有所改变外，其实质内

容亦有所改变。以下是分别从 Windows SDK3.1 版的 WINDOWS.H 文件及 Win32SDK 的 WINNT.H、WINDEF.H 文件中找出来的定义式：

从上面的式子可以看出，在 16 位环境中，HINSTANCE 可有两种类型，一种是一个 16 位的短指标(near pointer)；另一种是一个无号整数(unsigned int)。在 32 位环境中，HINSTANCE 一律是一个指标，而且在 32 位 Windows 应用程序中，没有所谓的长指标(far point)或短指标，所有的指标一律是 32 位，详细状态我们将在下一章讨论。

1-1-3 命令行参数

无论是 16 还是 32 位 Windows 环境，基本上命令行参数的内容并没有改变，不过在 Windows NT 环境中则有些例外，由于 UNICODE 的使用与否会导致系统判断命令行的方式有所改变，因此系统无法辨识命令行参数。为了确保应用程序可在 Windows NT 环境中正确执行，32 位 Windows 应用程序应该使用 GetCommandLine()函数取得命令行参数，不要直接使用 WinMain()函数的第三个参数。命令行参数的定义虽然没有改变，但其实质定义已经不一样，在 16 位 Windows 环境中，此参数是一个长指标，包括 16 位的段地址及偏移地址，共 32 位；在 32 位的 Windows 环境中，此参数依旧是一个指标，而系统的指标仅有一种，一律是 32 位的偏移地址。以下是 16 及 32 位 Windows SDK 内的定义。

对 16 位的 Windows SDK，WINDOWS.H 定义如下：

```
//define FAR              _far
typedef char FAR★       LPSTR;
```

对 32 位的 Windows SDK，WINNT.H 定义如下：

```
typedef char        CHAR;
typedef CHAR        ★LPSTR, ★PSTR;      /★不再具有 FAR 描述★/
```

1-2 窗口过程函数

窗口过程函数是 Windows 应用程序不可或缺的函数之一，它用来处理窗口的各种信息。对 16 位的 Windows 应用程序而言，窗口过程函数具有下列两种写法：

```
/★ Windws3.0 版时期的写法★/
LONG FAR PASCAL WndProc (
    HWND    hWnd.       /★ handle of window ★/
    WORD    Msg.        /★ message ID ★/
    WORD    wParam.
    LONG    lParam)
```

```
/★ Windows3.1 版时期的写法★/
LRESULT WINAPI WndProc (
    HWND      hWnd.       /★ handle of window ★/
    UINT      Msg.        /★ message ID ★/
    WPARAM    wParam.
    LPARAM    lParam)
```

对 32 位的 Windows 应用程序而言，窗口过程函数写法如下：

```
LRESULT APIENTRY WndProc (
    HWND      hWnd.      /★ handle of window ★/
    UINT      Msg.       /★ message ID ★/
    WPARAM    wParam.
    LPARAM    lParam)
```

1-2-1 函数调用方式

乍看之下，好像从 Windows3.0 至 3.1 版的窗口过程函数就已经有所改变了，而 32 位的窗口过程函数反而没有变化，要了解实际上的变化情形，我们必须先了解 16 及 32 位 Windows 对各种型态的定义。我们分别从 Windows SDK 3.1 的 WINDOWS.H 以及 Win 32 SDK 的 WINDEF.H、WINNT.H 文件中找出其明确定义式，其内容如下。

对 16 位的 Windows 而言，定义如下：

```
//define LONG       long          /★ 长整数                    ★/
typedef LONG        LRESULT;      /★ LRESULT 与 LONG 相同      ★/
//define FAR        _far          /★ 长程指标或远程调用        ★/
//define PASCAL     _pascal       /★ pascal 调用法             ★/
//define WINAPI     _far _pascal  /★ 与 FAR PASCAL 相同        ★/
```

对 32 位的 Windows 而言，APIENTRY 定义如下：

```
typedef long        LONG;         /★ 长整数                    ★/
typedef LONG        LRESULT;      /★ LRESULT 与 LONG 相同      ★/
//define WINAPI     _stdcall      /★standard 调用法            ★/
//define APIENTRY   WINAPI       /★ 与 WINAPI 相同            ★/
```

从上面的定义内容可以轻易看出，其实 Windows 3.0 版与 3.1 版之间并没有改变，只不过定义式的写法改变了。但是 16 位与 32 位 Windows 之间则有一项明显的变化——32 位 Windows 不再使用远程调用，这也是受到存储器处理方式改变的影响，详细情形我们将在下一章的内容中探讨。

1-2-2 参数类型

在 Windows 3.0 与 3.1 版之间，虽然参数定义方式不相同，不过其内容并没有真的改变，下面是从 Windows SDK 3.1 的 WINDOWS.H 文件中取出的内容：

16 位的 Windows SDK，WINDOWS.H 定义如下：

```
DECLARE_HANDLE(HWND);                  /★ 详见 1-1-2 ★/
typedef unsigned short      WORD;      /★ 16 位 ★/
typedef signed long         LONG;      /★ 32 位 ★/
typedef unsignde int        UNIT;      /★ 16 位 ★/
typedef UNIT                WPARAM;    /★ 16 位 ★ /
typedef LONG                LPARAM;    /★ 32 位 ★/
```

与之相对的，虽然 32 位 Windows 的窗口过程函数的参数与 16 位的 Windows3.1 版的

写法完全相同，不过实际状态则不相同，下面是从 Win 32 SDK 的 WINNT.H 与 WINDEF.H 文件中取出的内容。

32 位的 Windows SDK，定义如下：

```
DECLARE_HANDLE(HWND);          /★ 详见 1-1-2 ★/
typedef unsigned short    WORD;     /★ 16 位 ★/
typedef long              LONG;     /★ 32 位 ★/
typedef unsigned int      UINT;     /★ 32 位 ★/
typedef UINT              WPARAM;   /★ 32 位 ★/
typedef LONG              LPARAM;   /★ 32 位 ★/
```

除了 1-1-2 节已经讨论过的 DECLARE-HANDLE()巨集内容不相同外，另外有很重要的一点，即在两种 Windows 环境下，整数的含义也已经变了，如果您的应用程序是遵照 Windows.3.1 版的规范所撰写，即可免去修改动作，若是依照 3.0 版的规范所撰写，则必须修改应用程序。

1-3 对话框函数

如同窗口过程函数，对话框函数是 Windows 应用程序非常重要的函数之一，它用来处理对话框的各种信息。

对 16 位的 Windows 应用程序而言，对话框函数具有下列两种写法：

```
/★ Windows 3.0 版时期的写法 ★/
BOOL FAR PASCAL DlgProc (
    HWND    hDlg.       /★ handle of dialog box ★/
    WORD    Msg.        /★ message ID ★/
    WORD    wParam.
    LONG    lParam)
```

```
/★ Windows 3.1 版时期的写法 ★/
BOOL CALLBACK DlgProc (
    HWND    hDlg.       /★ handle of dialog box ★/
    UINT    Msg.        /★ message ID ★/
    WPARAM  wParam.
    LPARAM  lParam)
```

对 32 位的 Windows 应用程序而言，对话框函数写法如下：

```
BOOL APIENTRY DlgProc (
    HWND    hDlg.       /★ handle of window ★/
    UINT    Msg.        /★ message ID ★/
    WPARAM  wParam.
    LPARAM  lParam)
```

除了写法不同外，对话框函数的改变与窗口过程函数的状态完全相同，我们只需看一看