## Copyright

* Assembly Language for Intel-Based Computers, 5th Edition , Kip R. Irvine
* © Pearson Education, 2006. All rights reserved.
* Modified extensively by Kent Jones

## Learning Goals for This Study Guide

* Background Information

Win32 Console Programs

API and SDK

Windows Data Types

Standard Console Handles

* Console Input
* Console Output
* Reading and Writing Files
* Console Window Manipulation
* Controlling the Cursor
* Controlling the Text Color
* Time and Date Functions
* Writing graphical windows programs
* IA 32 Memory Management

# Win32 Windows Programming

## Background Information

### API and SDK

* Microsoft Win32 Application Programming Interface

**API**: a collection of types, constants, and functions that provide a way to directly manipulate objects through programming

* Microsoft Platform Software Development Kit

**SDK**: a collection of tools, libraries, sample code, and documentation that helps programmers create applications

**Platform**: an operating system or a group of closely related operating systems

### Translating Windows Data Types

|  |  |
| --- | --- |
| **Windows Type(s)** | **MASM Type** |
| BOOL, BYTE | BYTE |
| LONG | SDWORD |
| COLORREF, HANDLE, LPARAM,  LPCTSTR, LPTSTR, LPVOID,  LRESULT, UINT, WNDPROC, WPARAM | DWORD |
| BSTR, LPCSTR, LPSTR | PTR BYTE |
| WORD | WORD |
| LPCRECT | PTR RECT |

# Writing a Graphical Windows Application

## Required Files

* WinApp.asm - Program source code
* GraphWin.inc - Include file containing structures, constants, and function prototypes used by the program
* kernel32.lib - Same MS-Windows API library used earlier in this chapter
* user32.lib - Additional MS-Windows API functions

## POINT and RECT Structures

* POINT - X, Y screen coordinates
* RECT - Holds the graphical coordinates of two opposing corners of a rectangle

POINT STRUCT

ptX DWORD ?

ptY DWORD ?

POINT ENDS

RECT STRUCT

left DWORD ?

top DWORD ?

right DWORD ?

bottom DWORD ?

RECT ENDS

## MSGStruct Structure

* MSGStruct - holds data for MS-Windows messages (usually passed by the system and received by your application):

MSGStruct STRUCT

msgWnd DWORD ?

msgMessage DWORD ?

msgWparam DWORD ?

msgLparam DWORD ?

msgTime DWORD ?

msgPt POINT <>

MSGStruct ENDS

### WNDCLASS Structure

* Each window in a program belongs to a class, and each program defines a window class for its main window:

WNDCLASS STRUC

style DWORD ? ; window style options

lpfnWndProc DWORD ? ; WinProc function pointer

cbClsExtra DWORD ? ; shared memory

cbWndExtra DWORD ? ; number of extra bytes

hInstance DWORD ? ; handle to current program

hIcon DWORD ? ; handle to icon

hCursor DWORD ? ; handle to cursor

hbrBackground DWORD ? ; handle to background brush

lpszMenuName DWORD ? ; pointer to menu name

lpszClassName DWORD ? ; pointer to WinClass name

WNDCLASS ENDS

* style is a conglomerate of different style options, such as WS\_CAPTION and WS\_BORDER, that control the window’s appearance and behavior.
* lpfnWndProc is a pointer to a function (in our program) that receives and processes event messages triggered by the user.
* cbClsExtra refers to shared memory used by all windows belonging to the class. Can be null.
* cbWndExtra specifies the number of extra bytes to allocate following the window instance.
* hInstance holds a handle to the current program instance.
* hIcon and hCursor hold handles to icon and cursor resources for the current program.
* hbrBackground holds a background (color) brush.
* lpszMenuName points to a menu string.
* lpszClassName points to a null-terminated string containing the window’s class name.

## MessageBox Function

* Displays text in a box that pops up and waits for the user to click on a button:

MessageBox PROTO,

hWnd:DWORD,

pText:PTR BYTE,

pCaption:PTR BYTE,

style:DWORD

* hWnd is a handle to the current window. pText points to a null-terminated string that will appear inside the box. pCaption points to a null-terminated string that will appear in the box’s caption bar. style is an integer that describes both the dialog box’s icon (optional) and the buttons (required).

### MessageBox Example

.data

hMainWnd DWORD ?

QuestionText BYTE "Register this program now?"

QuestionTitle BYTE "Trial Period Has Expired"

.code

INVOKE MessageBox,

hMainWnd,

ADDR QuestionText,

ADDR QuestionTitle,

MB\_OK + MB\_ICONQUESTION

## WinMain Procedure

* Every Windows application needs a startup procedure, usually named WinMain, which is responsible for the following tasks:
* Get a handle to the current program
* Load the program’s icon and mouse cursor
* Register the program’s main window class and identify the procedure that will process event messages for the window
* Create the main window
* Show and update the main window
* Begin a loop that receives and dispatches messages

### WinProc Procedure

* WinProc receives and processes all event messages relating to a window

Some events are initiated by clicking and dragging the mouse, pressing keyboard keys, and so on

* WinProc decodes each message, carries out application-oriented tasks related to the message

WinProc PROC,

hWnd:DWORD, ; handle to the window

localMsg:DWORD, ; message ID

wParam:DWORD, ; parameter 1 (varies)

lParam:DWORD ; parameter 2 (varies)

* (Contents of wParam and lParam vary, depending on the message.)

### Sample WinProc Messages

* In the example program from this chapter, the WinProc procedure handles three specific messages:

WM\_LBUTTONDOWN, generated when the user presses the left mouse button

WM\_CREATE, indicates that the main window was just created

WM\_CLOSE, indicates that the application’s main window is about to close

* (many other messages are possible)

### ErrorHandler Procedure

* The ErrorHandler procedure has several important tasks to perform:

Call GetLastError to retrieve the system error number

Call FormatMessage to retrieve the appropriate system-formatted error message string

Call MessageBox to display a popup message box containing the error message string

Call LocalFree to free the memory used by the error message string

### ErrorHandler Sample

INVOKE GetLastError ; Returns message ID in EAX

mov messageID,eax

; Get the corresponding message string.

INVOKE FormatMessage, FORMAT\_MESSAGE\_ALLOCATE\_BUFFER + \

FORMAT\_MESSAGE\_FROM\_SYSTEM, NULL, messageID, NULL,

ADDR pErrorMsg, NULL, NULL

; Display the error message.

INVOKE MessageBox, NULL, pErrorMsg, ADDR ErrorTitle,

MB\_ICONERROR + MB\_OK

; Free the error message string.

INVOKE LocalFree, pErrorMsg

### Message Loop

* In WinMain, the message loop receives and dispatches (relays) messages:

Message\_Loop:

; Get next message from the queue.

INVOKE GetMessage, ADDR msg, NULL,NULL,NULL

; Quit if no more messages.

.IF eax == 0

jmp Exit\_Program

.ENDIF

; Relay the message to the program's WinProc.

INVOKE DispatchMessage, ADDR msg

jmp Message\_Loop

### Processing Messages

* WinProc receives each message and decides what to do with it:

WinProc PROC, hWnd:DWORD, localMsg:DWORD,

wParam:DWORD, lParam:DWORD

mov eax, localMsg

.IF eax == WM\_LBUTTONDOWN ; mouse button?

INVOKE MessageBox, hWnd, ADDR PopupText,

ADDR PopupTitle, MB\_OK

jmp WinProcExit

.ELSEIF eax == WM\_CREATE ; create window?

INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,

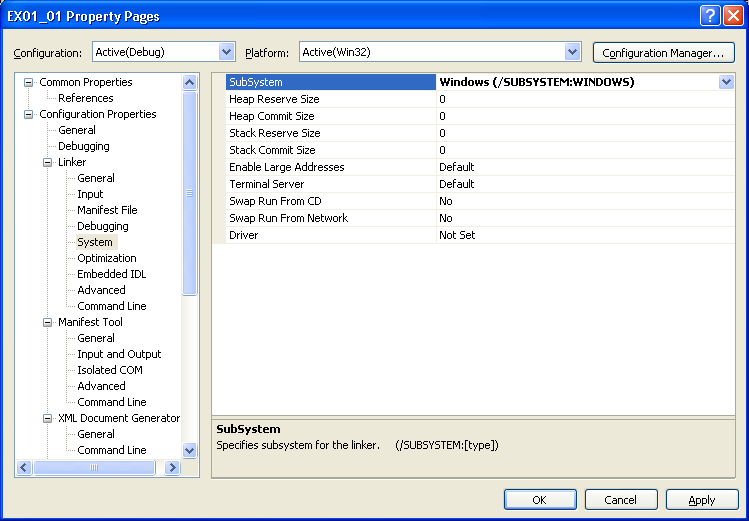
ADDR AppLoadMsgTitle, MB\_OK

jmp WinProcExit

(etc.)

### Custom Properties for Assembling/Linking Windows Applications

* When linking windows programs, remember to replace /SUBSYSTEM:CONSOLE with: /SUBSYSTEM:WINDOWS for the current project as shown in the following window:



### WinApp.asm Source File Listing

TITLE Windows Application (WinApp.asm)

; This program displays a resizable application window and

; several popup message boxes.

; Thanks to Tom Joyce for creating a prototype from which this

; program was derived. Last update: 9/24/01

INCLUDE Irvine32.inc

INCLUDE GraphWin.inc

;==================== DATA =======================

.data

AppLoadMsgTitle BYTE "Application Loaded",0

AppLoadMsgText BYTE "This window displays when the WM\_CREATE "

BYTE "message is received",0

PopupTitle BYTE "Popup Window",0

PopupText BYTE "This window was activated by a "

BYTE "WM\_LBUTTONDOWN message",0

GreetTitle BYTE "Main Window Active",0

GreetText BYTE "This window is shown immediately after "

BYTE "CreateWindow and UpdateWindow are called.",0

CloseMsg BYTE "WM\_CLOSE message received",0

ErrorTitle BYTE "Error",0

WindowName BYTE "ASM Windows App",0

className BYTE "ASMWin",0

; Define the Application's Window class structure.

MainWin WNDCLASS <NULL,WinProc,NULL,NULL,NULL,NULL,NULL, \

COLOR\_WINDOW,NULL,className>

msg MSGStruct <>

winRect RECT <>

hMainWnd DWORD ?

hInstance DWORD ?

;=================== CODE =========================

.code

WinMain PROC

**; Get a handle to the current process.**

INVOKE GetModuleHandle, NULL

mov hInstance, eax

mov MainWin.hInstance, eax

**; Load the program's icon and cursor.**

INVOKE LoadIcon, NULL, IDI\_APPLICATION

mov MainWin.hIcon, eax

INVOKE LoadCursor, NULL, IDC\_ARROW

mov MainWin.hCursor, eax

**; Register the window class.**

INVOKE RegisterClass, ADDR MainWin

.IF eax == 0

call ErrorHandler

jmp Exit\_Program

.ENDIF

**; Create the application's main window.**

**; Returns a handle to the main window in EAX.**

INVOKE CreateWindowEx, 0, ADDR className,

ADDR WindowName,MAIN\_WINDOW\_STYLE,

CW\_USEDEFAULT,CW\_USEDEFAULT,CW\_USEDEFAULT,

CW\_USEDEFAULT,NULL,NULL,hInstance,NULL

mov hMainWnd,eax

**; If CreateWindowEx failed, display a message & exit.**

.IF eax == 0

call ErrorHandler

jmp Exit\_Program

.ENDIF

**; Show and draw the window.**

INVOKE ShowWindow, hMainWnd, SW\_SHOW

INVOKE UpdateWindow, hMainWnd

**; Display a greeting message.**

INVOKE MessageBox, hMainWnd, ADDR GreetText,

ADDR GreetTitle, MB\_OK

**; Begin the program's message-handling loop.**

Message\_Loop:

**; Get next message from the queue.**

INVOKE GetMessage, ADDR msg, NULL,NULL,NULL

**; Quit if no more messages.**

.IF eax == 0

jmp Exit\_Program

.ENDIF

**; Relay the message to the program's WinProc.**

INVOKE DispatchMessage, ADDR msg

jmp Message\_Loop

Exit\_Program:

INVOKE ExitProcess,0

WinMain ENDP

**;-----------------------------------------------------**

WinProc PROC,

hWnd:DWORD, localMsg:DWORD, wParam:DWORD, lParam:DWORD

**; The application's message handler, all non-**

**; application-specific messages are forwarded**

**; to the default Windows message handler.**

**;-----------------------------------------------------**

mov eax, localMsg

.IF eax == WM\_LBUTTONDOWN ; mouse button?

INVOKE MessageBox, hWnd, ADDR PopupText,

ADDR PopupTitle, MB\_OK

jmp WinProcExit

.ELSEIF eax == WM\_CREATE ; create window?

INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,

ADDR AppLoadMsgTitle, MB\_OK

jmp WinProcExit

.ELSEIF eax == WM\_CLOSE ; close window?

INVOKE MessageBox, hWnd, ADDR CloseMsg,

ADDR WindowName, MB\_OK

INVOKE PostQuitMessage,0

jmp WinProcExit

.ELSE ; other message?

INVOKE DefWindowProc, hWnd, localMsg, wParam, lParam

jmp WinProcExit

.ENDIF

WinProcExit:

ret

WinProc ENDP

**;---------------------------------------------------**

ErrorHandler PROC

**; Display the appropriate system error message.**

**;---------------------------------------------------**

.data

pErrorMsg DWORD ? ; ptr to error message

messageID DWORD ?

.code

INVOKE GetLastError ; Returns message ID in EAX

mov messageID,eax

; Get the corresponding message string.

INVOKE FormatMessage, FORMAT\_MESSAGE\_ALLOCATE\_BUFFER + \

FORMAT\_MESSAGE\_FROM\_SYSTEM,NULL,messageID,NULL,

ADDR pErrorMsg,NULL,NULL

; Display the error message.

INVOKE MessageBox,NULL, pErrorMsg, ADDR ErrorTitle,

MB\_ICONERROR+MB\_OK

; Free the error message string.

INVOKE LocalFree, pErrorMsg

ret

ErrorHandler ENDP

END WinMain

## Hints for Tracking Down Link Errors

Whenever you attempt to “mix” languages (i.e. call functions from one language in another language), you have to make sure that you are calling the correct functions.

This is especially true for Microsoft Window’s API functions. Why? Because Microsoft has “upgraded” the API functions over the years and the names have changed. Microsoft did “header file tricks” (i.e. typedefs) so that API users could use the old function names to call the “new updated functions” that actually live in the libraries.

**Example 1:**

Consider the function **DrawText**. The **DrawText** function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines, and so forth). To specify additional formatting options, use the [**DrawTextEx**](ms-help://MS.VSCC/MS.MSDNVS/gdi/fontext_4pbs.htm) function:

int DrawText(

HDC *hDC*, // handle to DC

LPCTSTR *lpString*, // text to draw

int *nCount*, // text length

LPRECT *lpRect*, // formatting dimensions

UINT *uFormat* // text-drawing options

);

Now, you would think that you can simply “translate” this windows API function call into the following prototype in assembler:

DrawText PROTO,

hDC:DWORD,

lpString:DWORD,

nCount:DWORD,

lpRect:DWORD,

uFormat:DWORD

However, if you use this prototype you will get the following error message when you attempt to compile:

WinApp.obj : error LNK2001: unresolved external symbol \_DrawText@20

WinApp.exe : fatal error LNK1120: 1 unresolved externals

Why? Because the actual library function is different from the function that you are attempting to call.

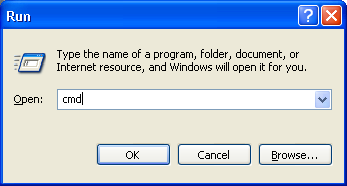
We now need to find out what the function in the library is actually called! Fortunately, Microsoft has provided a nice utility with Visual Studio called **dumpbin**. This utility will “dump” the symbols that are “exported” from a binary file or library.

IMPORTANT: See **Microsoft’s Knowledge Base Article 177429**  or see the **dumpbin** options help file in the help included with Microsoft Visual Studio.

Now, locate the library in which **DrawText** is “supposed” to live. You can find this again from the online help. In this example the library is **USER32.LIB**

**Step 1:**

Start an MSDOS command line prompt. To do this, select start | run then type cmd as shown:



**Step 2:**

To use dumpbin, you must set the Visual Studio Environment Variables for your command prompt by first running **vcvars32.bat**. To do this, run the following command from the command line prompt:

**"C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat"**

**Step 2:**

Now that the environment variables are set, you can execute the dumpbin.exe utility. The syntax for this command is: ***{path}*\Dumpbin.exe /LINKERMEMBER:1 *{path}*\USER32.LIB > tempfile**

Here is the complete actual command I ran from the command line:

**"C:\Program Files\Microsoft Visual Studio 8\VC\bin\dumpbin.exe" /LINKERMEMBER:1 "C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Lib\User32.Lib" > C:\tempfile**

This generated a list in **tempfile** of all the functions in the **USER32.LIB**. I then did a “search” on the **tempfile** to find all functions that had “Draw” in the function name.

**: :**

**19784 \_\_imp\_\_GrayStringA@36**

**197F0 \_GrayStringW@36**

**197F0 \_\_imp\_\_GrayStringW@36**

**14CA6 \_DrawTextA@20**

**14CA6 \_\_imp\_\_DrawTextA@20**

**14DE8 \_DrawTextW@20**

**: :**

After locating the function, I noticed that the actual (undecorated name) was **DrawTextA**. Thus, I went back to the assembler prototype for the function and changed it to **DrawTextA** as follows:

DrawTextA PROTO,

hDC:DWORD,

lpString:DWORD,

nCount:DWORD,

lpRect:DWORD,

uFormat:DWORD

Now, when I built the program again, the linker was able to find the function successfully!

Thus, the moral of the story: “Search either the header files carefully or the library files before assuming what the name of a function is!”

## Debugging Windows Programs in Assembler

When debugging windows programs, the **GetLastError( )** API function is the best friend you could ever want!

I used it to debug code I was running in response to a **WM\_PAINT**. I simply placed a call to this function after each of my API function calls (one at a time), like this:

INVOKE DrawText, hdc, ADDR str1, -1, ADDR rc, 37 ;25h

call ErrorHandler

This immediately told me that the handle **hdc** was invalid. I looked back over the code and figured out that I was not initializing this correctly (the return from **BeginPaint** is in **eax** not in **esi**! )

I made the change and now the windows program works perfectly!

# IA-32 Memory Management

## Reviewing Some Terms

* Multitasking permits multiple programs (or tasks) to run at the same time. The processor divides up its time between all of the running programs.

## New Terms

* A segment selector is a 16-bit value stored in a segment register (CS, DS, SS, ES, FS, or GS).

provides an indirect reference to a memory segment

## Translating Addresses

* The IA-32 processor uses a one- or two-step process to convert a variable's logical address into a unique memory location.
* The first step combines a segment value with a variable’s offset to create a linear address.
* The second optional step, called page translation, converts a linear address to a physical address.

## Converting Logical to Linear Address

* The segment selector points to a segment descriptor, which contains the base address of a memory segment. The 32-bit offset from the logical address is added to the segment’s base address, generating a 32-bit linear address.

Selector

Offset

Logical address

Segment Descriptor

Descriptor table

+

GDTR/LDTR

(contains base address of

descriptor table)

Linear address

## Indexing into a Descriptor Table

* Each segment descriptor indexes into the program's local descriptor table (LDT). Each table entry is mapped to a linear address:

Logical addresses

0018

0000003A

(unused)

DRAM

SS

ESP

001A0000

0002A000

0001A000

00003000

Local Descriptor Table

0010

000001B6

0008

00002CD3

LDTR register

DS

18

10

08

00

(index)

Linear address space

IP

offset

### Paging

* Paging makes it possible for a computer to run a combination of programs that would not otherwise fit into memory.
* Only part of a program must be kept in memory, while the remaining parts are kept on disk.
* The memory used by the program is divided into small units called pages.
* As the program runs, the processor selectively unloads inactive pages from memory and loads other pages that are immediately required.
* OS maintains page directory and page tables
* Page translation: CPU converts the linear address into a physical address
* Page fault: occurs when a needed page is not in memory, and the CPU interrupts the program
* OS copies the page into memory, program resumes execution

## Page Translation

* A linear address is divided into a page directory field, page table field, and page frame offset. The CPU uses all three to calculate the physical address.

Directory

Table

Offset

Directory Entry

CR3

Page Directory

Page-Table Entry

Page Table

Physical Address

Page Frame

Linear Address

10

10

12

32