Programming Windows

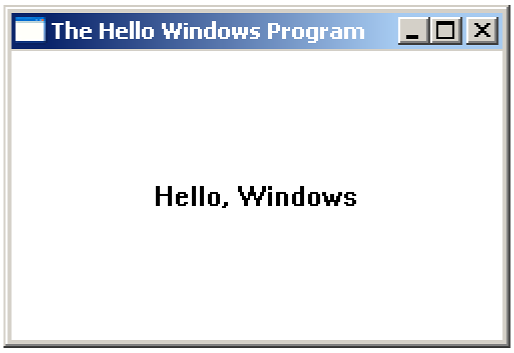
Terry Marris January 2013

2 Hello Windows

We display some text in the middle of a window, and see how the text remains there whenever the window is re-sized or moved.

2.1 Hello Windows

The output from the *hellowin* program is



A window class structure is initialised. A window is created based on this structure and a message loop is set up. A windows procedure manages the message queue and processes selected messages. Here is the entire source code:

```
/* hellowin.c - Displays "Hello, Windows" in the client area */

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                   PSTR cmdLine, int cmdShow)
{
   static char appName[] = "HelloWin";
   HWND hwnd;
   MSG msg;
   WNDCLASS wc;

   wc.style          = CS_HREDRAW | CS_VREDRAW;
   wc.lpfnWndProc    = WndProc;
   wc.cbClsExtra     = 0;
   wc.cbWndExtra     = 0;
   wc.hInstance      = hInst;
   wc.hIcon          = NULL;
   wc.hCursor        = LoadImage(NULL, IDC_ARROW, IMAGE_CURSOR,
                                 0, 0, LR_SHARED);
   wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
   wc.lpszMenuName  = NULL;
   wc.lpszClassName = appName;

   RegisterClass(&wc);

   hwnd = CreateWindow(appName,
                       "The Hello Windows Program",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL,
                       NULL,
                       hInst,
                       NULL);

   ShowWindow(hwnd, cmdShow);
   UpdateWindow(hwnd);

   while (GetMessage(&msg, NULL, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return msg.wParam ;
}


LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
                         WPARAM wParam, LPARAM lParam)
{
   HDC hdc;
   PAINTSTRUCT ps;
   RECT rect;

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

      GetClientRect(hwnd, &rect);

      DrawText(hdc, "Hello, Windows", -1, &rect,
               DT_SINGLELINE | DT_CENTER | DT_VCENTER);

      EndPaint(hwnd, &ps);
      return 0;

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

2.2 The Window Class

A window is always based on a *window class*. A window class identifies the procedure that processes messages to the window. That procedure is named *WndProc* in our example.

And the messages include *WM_PAINT* and *WM_DESTROY*. *WndProc* and the messages are discussed below.

```
static char appName[] = "HelloWin";
   ...
WNDCLASS wc;

wc.style         = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc   = WndProc;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;
wc.hInstance     = hInst;
wc.hIcon         = NULL;

wc.hCursor       = LoadImage(NULL, IDC_ARROW, IMAGE_CURSOR,
                             0, 0, LR_SHARED);
wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.lpszMenuName  = NULL;
wc.lpszClassName = appName;
```

A *WNDCLASS* structure has ten members.

```
wc.style = CS_HREDRAW | CS_VREDRAW;
```

says that the entire window is to be redrawn if it changes in size, either horizontally or vertically. *CS* stands for Class Style. | is the C bitwise or operator.

```
wc.lpfnWndProc = WndProc;
```

WindProc, shown above and discussed below, is the function that draws the text in the middle of the window. *lpfn* stands for long pointer to a function. And the type of *lpnfnWndProc* is *WNDPROC*.

```
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
```

cbClsExtra and *cbwndExtra* are both set to 0. These reserve extra space in both the class and windows structures if required by the program. Both are *int*s. They are not used here.

```
wc.hInstance = hInst;
```

The field *hInstance* is set to the parameter *hInst*. *hInstance* is a handle that uniquely identifies the program at run time.

```
wc.hIcon = NULL;
```

hIcon represents the little icon shown on the left side of the title bar. If its value is *NULL* then Windows provides a default icon. Its type is *HICON*, a handle to an icon. *WNDCLASSEX*, an extended version of *WNDCLASS*, would be used if you want to set an icon.

The cursor is set with

```
wc.hCursor = LoadImage(NULL, IDC_ARROW, IMAGE_CURSOR,
                       0, 0, LR_SHARED);
```

NULL indicates that the image is to be loaded from a standalone resource, in this case, a standard system cursor. *IDC_ARROW* is the standard arrow-shaped cursor.

IMAGE_CURSOR says that the image is a cursor. The *0, 0*, means that the actual width and height of the cursor is used. And *LR_SHARED* must be specified to load a system cursor. *LoadImage()* returns a handle to a cursor, or *NULL* if it fails.

The background colour is set to white with

```
   wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

GetStockObject() returns a handle to the requested object, or *NULL* if it fails for any reason.

```
   wc.lpszMenuName = NULL;
```

says there is no menu set.

```
   wc.lpszClassName = appName;
```

sets the class name to *HelloWin*. *appName* is also used in the call to *CreateWindow* - see below.

The class is registered with the Windows operating system by

```
   RegisterClass(&wc);
```

and made ready for a subsequent call to *CreateWindow*. You would use *RegisterClassEx* if you wanted to set the small window icon. *RegisterClass* returns zero on error.

2.3 The Window

A window is created by a call to *CreateWindow*, and displayed by a call to *ShowWindow*.

Windows are identified by a handle, an integer that uniquely identifies the window. *HWND* is the handle-to-a-window type. *CreateWindow* returns a handle to a window, or *NULL* if it fails.

```
   HWND hwnd;
      ...
   hwnd = CreateWindow(
      appName,                      /* the class name               */
      "The Hello Program",          /* title bar text               */
      WS_OVERLAPPEDWINDOW,          /* window style                 */
      CW_USEDEFAULT, CW_USEDEFAULT, /* position of top left corner  */
      CW_USEDEFAULT, CW_USEDEFAULT, /* window width and height      */
      NULL,                         /* parent window                */
      NULL,                         /* menu                         */
      hInstance,                    /* program instance handle      */
      NULL);                        /* additional creation parameters */
```

The class name defines the window type. It could be a programmer-defined type, the class name as used here, or a pre-defined type such as *BUTTON*, *COMBOBOX* or *EDIT* depending on the purpose.

The window style *WS_OVERLAPPED* gives the window a title bar, a system menu, a border, and minimise, maximise and close buttons.

The Create Window identifier *CW_USEDEFAULT* means a value chosen by Windows itself is used to position and size the window.

CreateWindow returns *NULL* on error.

```
ShowWindow(hwnd, cmdShow);
```

sets the given window's display state and displays the window. *hwnd* is the handle to the window previously created. *cmdShow* determines how the window is to be initially shown, for example, normally, or minimised. If the value of *cmdShow* is *SW_SHOWNORMAL* the client area of the window is erased with the background brush specified in the window class. The client area is then repainted with

```
UpdateWindow(hwnd);
```

by sending a *WM_PAINT* message to the Windows procedure *WndProc*.

The client area is the part of a window below the title bar. *UpdateWindow* returns zero on error.

2.4 The Message Loop

Events such as mouse clicks and keystrokes generate messages. Windows holds these messages in the *message queue*. *GetMessage* retrieves the next message from the queue. *TranslateMessage* translates keyboard keystrokes and *DispatchMessage* passes messages to the registered window procedure, *WndProc*.

The message structure, *MSG*, looks like this:

```
typedef struct tagMSG {
   HWND hwnd;
   UINT message;
   WPARAM wParam;
   LPARAM lparam;
   DWORD time;
   POINT pt;
} MSG;
```

where *POINT* is

```
typedef strict tagPPOINT {
   LONG x;
   LONG y;
} POINT;
```

DWORD stands for double size *WORD*, where a *WORD* is 16 bits, an unsigned integer. And *LONG* is a long integer. Messages in the form of an integer or a handle are usually passed in a *WPARAM* object, and messages in the form of a pointer are usually passed in a *LPARAM* object.

```
MSG msg;
   ...
while (GetMessage(&msg, NULL, 0, 0)) {
   TranslateMessage(&msg);
   DispatchMessage(&msg);
}
return msg.wParam ;
```

In *GetMessage*, *msg* is the message structure, *NULL* specifies any current window, and the *0, 0* specify that no message filters are used.

A *WM_QUIT* Windows Message, generated, for example by clicking on a window's close icon, terminates the loop and the *wParam* member of the message structure is returned.

2.5 The Window Procedure

The windows procedure, named *WinProc* in our example, determines how the window responds to user input. It is referred to when the window class is created, and it is called automatically by *DispatchMessage*. You could have several windows based on the window class.

The function header is

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
                         WPARAM wParam, LPARAM lParam)
```

LRESULT is an integer returned to Windows and contains the response to a message.

CALLBACK is the calling convention for the function.

The four parameters are identical to the first four fields of the *MSG* structure.

hwnd is the handle to the window receiving the message. This is the same handle returned by a call to *CreateWindow*.

UINT is an *unsigned int*. *msg* identifies the message code. For example, the *WM_SIZE* message indicates the window was resized.

wParam and *lParam* both contain more information about the message. *wParam* usually contains a handle or an integer. *lParam* usually contains a pointer.

2.7 Window Messages

Window Messages, such as *WM_PAINT* and *WM_DESTROY*, are identified by a number contained within the *msg* parameter to *WndProc*.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
                         WPARAM wParam, LPARAM lParam)
{
   HDC hdc;
   PAINTSTRUCT ps;
   RECT rect;

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

      GetClientRect(hwnd, &rect);

      DrawText(hdc, "Hello, Windows", -1, &rect,
               DT_SINGLELINE | DT_CENTER | DT_VCENTER);
      EndPaint(hwnd, &ps);
      return 0;

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

HDC is a handle to a device context. A device context refers to an actual output device, such as a video display, and its driver.

PAINTSTRUCT contains information that is used to paint a client area.

RECT is a structure that defines the top left and bottom right corners of a rectangle.

Messages are usually processed by a switch-with-case statement.

```
   switch (msg) {
   case WM_PAINT:
      ...

   case WM_DESTROY:
      ...
   }
```

A *WM_PAINT* is a message to repaint the window. The process begins with a call to *BeginPaint* and ends with a call to *EndPaint*.

```
   hdc = BeginPaint(hwnd, &ps);
      ...
   EndPaint(hwnd, &ps);
```

BeginPaint fills the paint structure *ps* and prepares the window for painting. *BeginPaint* returns *NULL* on failure, indicating that no device context was available.

GetClientRect retrieves the coordinates of the client area and returns zero on failure.

Now we come to

```
   DrawText(hdc, "Hello, Windows", -1, &rect,
            DT_SINGLELINE | DT_CENTER | DT_VCENTER);
```

DrawText writes formatted text in the given rectangle. Here, *hdc* is the handle to a device context, *Hello, Windows* is the text to be displayed, *-1* means the text is terminated with the null character, *&rect* is a pointer to the rectangle, *DT_SINGLELINE* | *DT_CENTER* | *DT_VCENTER* is how the text is to be formatted, on a single line in the middle of the client area. DT stands for Draw Text. *DrawText* returns zero on error.

The *WM_DESTROY* message indicates that the user has clicked on the window close button or has selected close from the program's system menu. *PostQuit(0)* inserts a *WM_QUIT* message into the message queue. When *GetMessage* retrieves this *WM_QUIT* message, *GetMessage* returns zero and the message loop terminates.

The final statement in *WndProc*

```
   return DefWindowProc(hwnd, msg, wParam, lParam);
```

deals with all the messages not processed by *WndProc*. Its arguments are the same as *WndProc*'s parameters.

2.8 Size and Complexity

To manage size and complexity, we usually put the logic for handling each message into a separate function, something like this:

```
...

void CALLBACK printHello(HWND hwnd)
{
   HDC hdc;
   PAINTSTRUCT ps;
   RECT rect;

   hdc = BeginPaint(hwnd, &ps);

   GetClientRect(hwnd, &rect);

   DrawText(hdc, "Hello, Windows", -1, &rect,
               DT_SINGLELINE | DT_CENTER | DT_VCENTER);

   EndPaint(hwnd, &ps);
}


LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
                   WPARAM wParam, LPARAM lParam)
{
   switch (msg) {
   case WM_PAINT:
      printHello(hwnd);
      return 0;

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

CALLBACK is the calling convention that deals with arguments in a Win32-specific way. You could (should?) use it whenever your function makes Win32 function calls, or when your function uses a Win32 parameter type.