Programming Windows

Terry Marris January 2013

3 Message Boxes

A message box is a pop-up window that shows a message, a set of buttons and, perhaps, an icon. The message box usually remains in control until the user responds with a mouse click on a message box button.

In this chapter we see how to use a message box to report run-time errors and diagnostics.

3.1 Windows Errors

Browse through the Windows documentation provided on *http://msdn.microsoft.com/en-us/library* and you will see what various Windows functions return if an error occurs. For example:

function	***returns on error***	***reason***
BeginPaint()	*NULL*	display device not available
CreateWindow()	*NULL*	invalid parameter value, windows procedure failure, control in dialogue template not registered
DrawText	*0*	
GetClientRect	*0*	
GetMessage()	*-1*	invalid window handle, invalid message
GetStockObject	*NULL*	
LoadImage	*NULL*	
MessageBox()	*0*	
RegisterClass	*0*	
TranslateMessage()	*0*	message not translated or not posted to the message queue
UpdateWindow()	*0*	

Serious programmers might take exhaustive pains to maintain control if a run-time error occurs, even if the code does become bloated. Authors of introductory texts might neglect error-handling to keep things simple. We take a look at how error scenarios may be handled.

3.2 Error Reporting

We may have used *printf* statements to report errors and diagnostics, and we may have used e*xit(EXIT_FAILURE)* to immediately halt program execution.

Now, we see how to use message boxes to report errors, and how to use *PostQuitMessage(0)* to terminate program execution.

A report could be fatal requiring immediate program termination, or just a warning, or a diagnostic enabling the values of variables to be examined.

```
#define REP_FATAL 1
#define REP_WARNING 2
#define REP_DIAGNOSTIC 3
```

The *report* function shown below implements variable length parameter lists.

```
int report(int repType, PSTR caption, PSTR format, ...)
{
   CHAR string[256];
   va_list args;

   va_start(args, format);
   vsprintf(string, format, args);
   va_end(args);

   MessageBox(NULL, string, caption, MB_OK);

   if (repType == REP_FATAL)
     PostQuitMessage(0);
   return 0;
}
```

Example calls to the function include

```
   char *fileName;
   ...
   report(REP_FATAL, "File Handler", "Unable to open file %s", "fileName);
```

and

```
   int age;
   double height;
   ...
   report(REP_DIAGNOSTIC, "Suspect" "Age: %, height: %0.2f", age, height);
```

repType is one of *REP_FATAL*, *REP_WARNING* or *REP_DIAGNOSTIC*. *caption* is the text passed to the message box title bar. *format* is the conversion specification, such as *%s*, *%d*, *%0.2f*, as used in a regular *printf* statement.

The ellipsis punctuator, the three dots at the end of the parameter list, indicates that the function can have a variable number of arguments. *va_list*, *va_start* and *va_end* manage the variable parameter list. *va_list* is the type used to handle lists of variable length. *va_start* initialises the object that holds the function arguments. *va_end* tidies up after an argument list has been traversed.

vsprintf constructs a formatted string. *format* contains the usual conversion specifications, the same as those you would use with *printf*. As with *printf*, the number, order and type of format specifications must match the number, order and type of the arguments they refer to. We assume *string* is large enough.

In

```
  MessageBox(NULL, string, caption, MB_OK);
```

NULL indicates there is no parent window, *string* is the text to be displayed, *caption* is the text that appears in the message box's title bar. And *MB_OK* is the OK button.

PostQuitMessage(0) is a request to exit the message loop and quit program execution.

An example of a call to *report* is shown below, where a deliberate error has been made in the first argument in the call to *CreateWindow*.

```
if (NULL == (hwnd = CreateWindow("ERROR", /* appName, */
                     "Robust Hello Windows",
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     NULL,
                     NULL,
                     hInst,
                     NULL)))
   report(REP_FATAL, "WinMain", "%s", "CreateWindow failed");
```

And here is the result.

The entire program is shown below.

```
/* report.c - Displays warnings and errors */

#include <windows.h>
#include <stdarg.h>
#include <stdio.h>

#define REP_FATAL 1
#define REP_WARNING 2
#define REP_DIAGNOSTIC 3
```

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

int report(int repType, PSTR caption, PSTR format, ...)
{
   CHAR string[256];
   va_list args;

   va_start(args, format);
   vsprintf(string, format, args);
   va_end(args);

   MessageBox(NULL, string, caption, MB_OK);

   if (repType == REP_FATAL)
      PostQuitMessage(0);
   return 0;
}


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

   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;

   if (0 == (RegisterClass(&wc)))
      report(REP_FATAL, "WinMain", "%s", "RegisterClass failed");

   if (NULL == (hwnd = CreateWindow("ERROR", /* appName, */
                       "Robust Hello Windows",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL,
                       NULL,
                       hInst,
                       NULL)))
      report(REP_FATAL, "WinMain", "%s", "CreateWindow failed");

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

```
   while ((ret = GetMessage(&msg, NULL, 0, 0)) != 0) {
      if (ret < 0)
         report(REP_FATAL, "WinMain", "%s", "GetMessage failed");
      else {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }
   }
   return msg.wParam ;
}


LRESULT CALLBACK WndProc(HWND hwnd, UINT msg,
                   WPARAM wParam, LPARAM lParam)
{
   return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

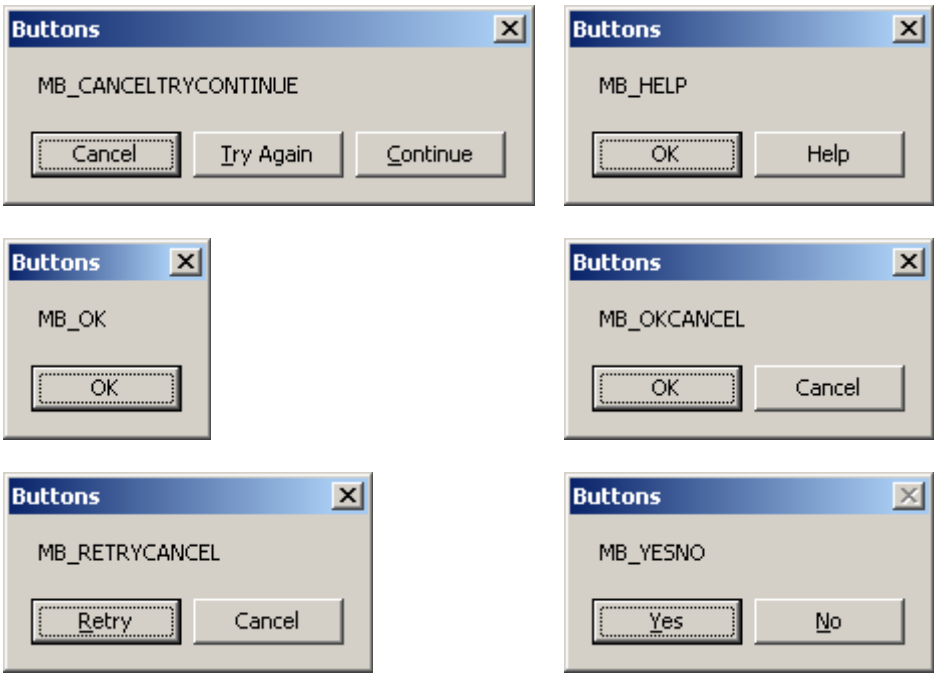
3.3 Message Boxes

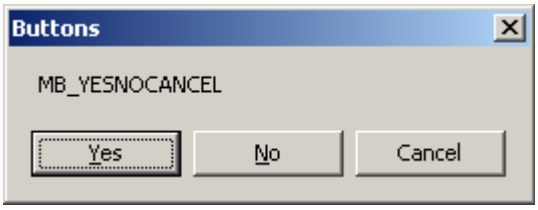
MessageBox has the format

int MessageBox(hwnd, text, title, type)

hwnd is the parent window. *text* is the message to be displayed. New lines, *\n*, can be imbedded in the text if required. *title* is the text that appears in the message box title bar. *type* is a combination of buttons and icons that are connected with the bitwise or operator. The value returned by *MessageBox* represents the button pressed by the user, or zero if an error has occured.

Buttons include





Icons include

Icon	Flag Values
	MB_ICONHAND, MB_ICONSTOP, or MB_ICONERROR
	MB_ICONQUESTION
	MB_ICONEXCLAMATION or MB_ICONWARNING
	MB_ICONASTERISK or MB_ICONINFORMATION

Return values include

Return Value	Button Pressed
IDABORT	Abort
IDCANCEL	Cancel
IDCONTINUE	Continue
IDIGNORE	Ignore
IDNO	No
IDOK	OK
IDTRYAGAIN	Try Again
IDYES	Yes

In the next chapter we start looking at buttons.