Building Win32 GUI Applications with MinGW

Introduction

MinGW is a native Win32 port of the open source GNU Compiler Collection, and can be used to write applications targeting Windows in languages such a C and C++ (see the MinGW web site (http://mingw.org/) for further details of the supported programming languages). MinGW also supports cross compilation, for example allowing you to build Windows applications using a Linux based system. Whether you're running Windows, Linux, Mac OS, or some other OS, I will show you how use MinGW to create professional quality GUI (Graphical User Interface) applications targeting Microsoft Windows.

In this article, I will build a basic Windows GUI application in C using MinGW and the *mingw32-make* utility. It's not intended to be the worlds best example of user interface design, but rather an attempt to demonstrate some of the functionality which can be achieved using MinGW with the minimum amount of code. This is much like what the Visual Studio Application Wizard will generate for you, and you can use my application as a template for your own applications.

Get the Code

If you have Git installed, you can get the sample code by running "`git clone https://github.com/TransmissionZero/MinGW-Win32-Application.git`". Alternatively you can download a MinGW Win32 Application source release. You are also encouraged to visit that link to star and watch the repository if you find it useful.

Application Features

The following are the features which the Win32 application should demonstrate:

- Resizeable main window, with an empty client area.
- About dialog, with some basic text, an icon, and an "ok" button.
- Windows "visual styles" support, so that controls such as buttons are consistent with other Windows applications running with visual styles enabled.
- Version information resource, so that the version information and copyright information can be viewed using Windows Explorer.
- Main menu allowing exiting of the application, and the showing of the about dialog.
- Keyboard accelerator "Alt + A" to show the about dialog.
- System menu item allowing the about dialog to be shown.
- Unicode support.
- Target either 32 bit or 64 bit versions of Windows, without any source code changes.

As I said, it's not intended to be an example of great UI design, hence having a keyboard accelerator and system menu item which do nothing other than show the "about" dialog, but it does give you the basics you need in order to adapt it into a very professional Windows application.

Setting Up MinGW

It is assumed that you already have MinGW installed on your PC. If not, Windows users will find that it is covered very well in the MinGW "Getting Started" Wiki (http://www.mingw.org/wiki/Getting_Started). When setting up MinGW using this method, the C compiler will be installed by default. No other components are needed in order to build Windows applications in C, but you'll probably want to install the C++ compiler as well if you intend to mix your C with C++.

Installation under other operating systems will vary, for example under the Fedora Linux distribution, you can run "`yum install mingw32-gcc mingw32-gcc-c++ mingw32-w32api`" to install the necessary packages.

The Application’s WinMain Procedure

A good place to start would be with our WinMain procedure. Our application contains a fairly vanilla Window, typical of your usual “Hello Windows!” tutorial, but with a few additions which you won’t get by default (if you’re not familiar with the structure, I’d recommend checking out theForger’s Win32 tutorial (http://www.winprog.org/tutorial/)). The following is our *WinMain* function, which I will disect and explain in the following sections:

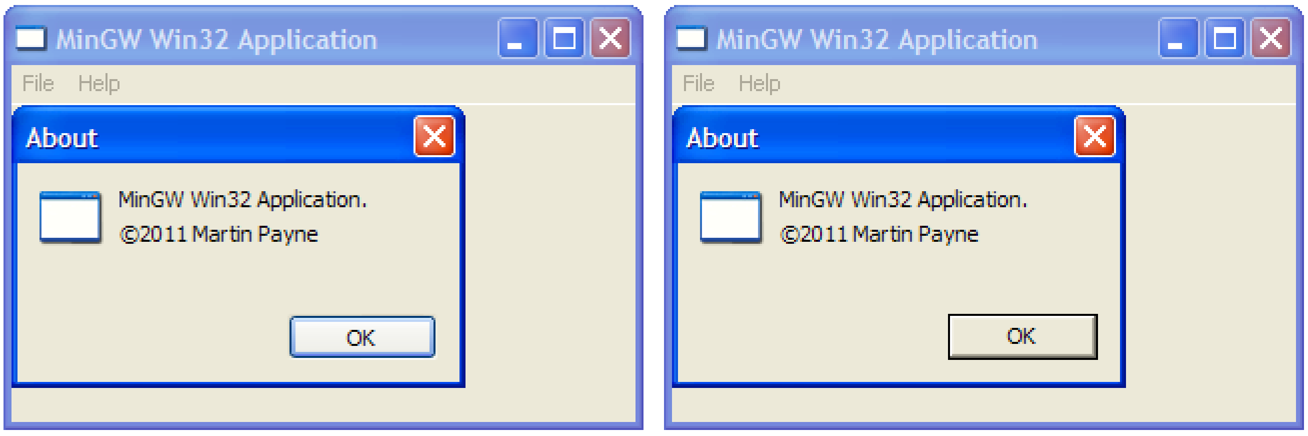
```
// Our application entry point.
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow)
{
  INITCOMMONCONTROLSEX icc;
  WNDCLASSEX wc;
  LPCTSTR MainWndClass = TEXT("Win32 Test application");
  HWND hWnd;
  HACCEL hAccelerators;
  HMENU hSysMenu;
  MSG msg;

  // Initialise common controls.
  icc.dwSize = sizeof(icc);
  icc.dwICC = ICC_WIN95_CLASSES;
  InitCommonControlsEx(&icc);

  // Class for our main window.
  wc.cbSize        = sizeof(wc);
  wc.style         = 0;
  wc.lpfnWndProc   = &MainWndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = (HICON) LoadImage(hInstance, MAKEINTRESOURCE(IDI_APPICON),
IMAGE_ICON, 0, 0,
                                       LR_DEFAULTSIZE | LR_DEFAULTCOLOR |
LR_SHARED);
  wc.hCursor       = (HCURSOR) LoadImage(NULL, IDC_ARROW, IMAGE_CURSOR, 0, 0,
LR_SHARED);
  wc.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1);
  wc.lpszMenuName  = MAKEINTRESOURCE(IDR_MAINMENU);
  wc.lpszClassName = MainWndClass;
  wc.hIconSm       = (HICON) LoadImage(hInstance, MAKEINTRESOURCE(IDI_APPICON),
IMAGE_ICON,
                                       GetSystemMetrics(SM_CXSMICON),
GetSystemMetrics(SM_CYSMICON),
                                       LR_DEFAULTCOLOR | LR_SHARED);

  // Register our window classes, or error.
  if (! RegisterClassEx(&wc))
  {
    MessageBox(NULL, TEXT("Error registering window class."), TEXT("Error"),
MB_ICONERROR | MB_OK);
    return 0;
  }

  // Create instance of main window.
  hWnd = CreateWindowEx(0, MainWndClass, MainWndClass, WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT, CW_USEDEFAULT,
                        320, 200, NULL, NULL, hInstance, NULL);

  // Error if window creation failed.
  if (! hWnd)
  {
    MessageBox(NULL, TEXT("Error creating main window."), TEXT("Error"),
MB_ICONERROR | MB_OK);
    return 0;
  }

  // Load accelerators.
  hAccelerators = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDR_ACCELERATOR));

  // Add "about" to the system menu.
  hSysMenu = GetSystemMenu(hWnd, FALSE);
  InsertMenu(hSysMenu, 5, MF_BYPOSITION | MF_SEPARATOR, 0, NULL);
  InsertMenu(hSysMenu, 6, MF_BYPOSITION, ID_HELP_ABOUT, TEXT("About"));

  // Show window and force a paint.
```

```
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    // Main message loop.
    while(GetMessage(&msg, NULL, 0, 0) > 0)
    {
      if (! TranslateAccelerator(msg.hwnd, hAccelerators, &msg))
      {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
      }
    }

    return (int) msg.wParam;
}
```

Also, don't worry too much about the missing `#includes` and missing functions—the functions will be covered as and when we come across them, and the full code including Makefile is available to download at the end of the article.

Enabling Visual Styles

The first part of our `WinMain` procedure involves enabling visual styles, which were introduced with Windows XP. This is what gives controls such as buttons their rounded 3D look under the Windows XP "Luna" Theme, along with the redesigned Window borders. In my opinion, supporting visual styles is one of the simplest things you can do to make an application fit in with the native Windows look and feel. You can see the difference on the buttons in the following screenshot of our "about" dialog, with and without visual styles enabled:



There are two things you must do to enable visual styles in versions of Windows from XP to Windows 7 (they are not supported in earlier versions of Windows, and are always enabled from Windows 8 onwards). Firstly, you must create an XML manifest specifying that version 6 of the Windows common controls library must be loaded by Windows, and embed it into your application as a resource with type "RT_MANIFEST" (actually, if you prefer you can name the manifest "[application name].exe.manifest" and include it in the same directory as the exe, rather than embedding it as a resource, but that does have the disadvantage that it can become accidentally deleted or corrupted). The manifest looks like the following:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>

<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
    <application>
      <!-- Supports Windows Vista / Server 2008 -->
      <supportedOS Id="{e2011457-1546-43c5-a5fe-008deee3d3f0}"/>
      <!-- Supports Windows 7 / Server 2008 R2 -->
      <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>
      <!-- Supports Windows 8 / Server 2012 -->
      <supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}"/>
      <!-- Supports Windows 8.1 / Server 2012 R2 -->
      <supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/>
      <!-- Supports Windows 10 -->
      <supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}"/>
    </application>
  </compatibility>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="asInvoker" uiAccess="false"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="Win32" name="Microsoft.Windows.Common-Controls"
version="6.0.0.0" processorArchitecture="*" publicKeyToken="6595b64144ccf1df"
language="*"/>
    </dependentAssembly>
  </dependency>
</assembly>
```

The important part is contained in the "`<dependency></dependency>`", and it's this which tells the Windows loader to load version 6 of the common controls library. You can embed it as a resource by putting the following in your resource script (replacing "*Application.manifest*" with the actual filename of your manifest):

```
CREATEPROCESS_MANIFEST_RESOURCE_ID RT_MANIFEST "Application.manifest"
```

The second step is to initialise the common controls. I've seen this step missed in some examples, and it can cause some odd behaviours under certain operating system versions and certain conditions, such as buttons being invisible, or dialogs failing to be created. Here is the snippet from the WinMain code above which does this:

```
INITCOMMONCONTROLSEX icc;

// Initialise common controls.
icc.dwSize = sizeof(icc);
icc.dwICC = ICC_WIN95_CLASSES;
InitCommonControlsEx(&icc);
```

Of course, if there are other common controls you need to initialise besides "`ICC_WIN95_CLASSES`", you should bitwise OR the `dwICC` member of the `INITCOMMONCONTROLSEX` structure accordingly. The INITCOMMONCONTROLSEX Structure (http://msdn.microsoft.com/en-us/library/bb775507.aspx) MSDN article has further details on this. Additionally, you will need to "`#include <commctrl.h>`" and specify linker flag "`-lcomctl32`" in order to not have compiler / linker errors.

Supporting User Account Control

In the XML manifest, you will notice the "`<requestedExecutionLevel/>`" section. You can find the full details in the Designing UAC Applications for Windows Vista (http://msdn.microsoft.com/en-us/library/bb756973.aspx) MSDN article, but here we have simply specified that the application doesn't require administrative privileges with the "`level="asInvoker"`" attribute. If your application requires administrator privileges then you can change this attribute accordingly. It's a good idea to always be explicit about this though, whether you require privilege escalation or not—it prevents Windows Vista and above from having to "guess" the privileges needed by your application.

Unicode Support

Windows 9x (Windows 95, Windows 98, and Windows ME) use an 8 bit character set internally, the specifics of which depend on your language, and the Windows API functions take character and string parameters which are based on the C "`char`" type. Windows NT (Windows NT 3.1 through to Windows XP, Windows 7, and beyond) support Unicode internally, in order to support true internationalisation, and the Windows API functions take character and string parameters which are based on the C "`wchar_t`" type. These encodings are referred to as *ANSI* and *Unicode* in most documentation (a bit of a misnomer, but I'll use this terminology to be consistent with the documentation).

Windows 9x generally supports only the ANSI versions of Windows API functions, whereas Windows NT generally supports both the ANSI and Unicode versions of functions. There are a few exceptions to this rule, but these are special cases. The recommendation is that you use the Unicode versions of functions in all of your applications, and use ANSI only if you have a need to support Windows 9x. It's actually quite easy to support both, because the Windows functions are prototyped like so:

```
/* Taken from "winuser.h" from the Microsoft Platform SDK. */

WINUSERAPI
int
WINAPI
MessageBoxA(
    __in_opt HWND hWnd,
    __in_opt LPCSTR lpText,
    __in_opt LPCSTR lpCaption,
    __in UINT uType);
WINUSERAPI
int
WINAPI
MessageBoxW(
    __in_opt HWND hWnd,
    __in_opt LPCWSTR lpText,
    __in_opt LPCWSTR lpCaption,
    __in UINT uType);
#ifdef UNICODE
#define MessageBox  MessageBoxW
#else
#define MessageBox  MessageBoxA
#endif // !UNICODE
```

If the preprocessor definition "`UNICODE`" is defined, then "`MessageBox`" is defined as "`MessageBoxW`" (Unicode version), otherwise it is defined as "`MessageBoxA`" (ANSI version). Notice that the ANSI version takes an "`LPCSTR`" (ANSI string), whereas the Unicode version takes an "`LPCWSTR`" (Unicode string). The Unicode (or *wide*) string would need to be prefixed with the letter 'L', for example `L"Hello world!"`, whereas the ANSI one should have no prefix. By using the `TEXT()` macro, the prefix will automatically be added where necessary, and you don't need to worry about your string literals. Putting it together looks like this:

```
MessageBox(NULL, TEXT("Error creating main window."), TEXT("Error"), MB_ICONERROR |
MB_OK);
```

To use Unicode functions, it's simply necessary to define "`UNICODE`" and "`_UNICODE`" when compiling your object code. The underscore prefixed definition I did not mention, but the only difference is that `UNICODE` is for Windows API functions, whereas `_UNICODE` is for C runtime functions. The object code compilation should look something like this:

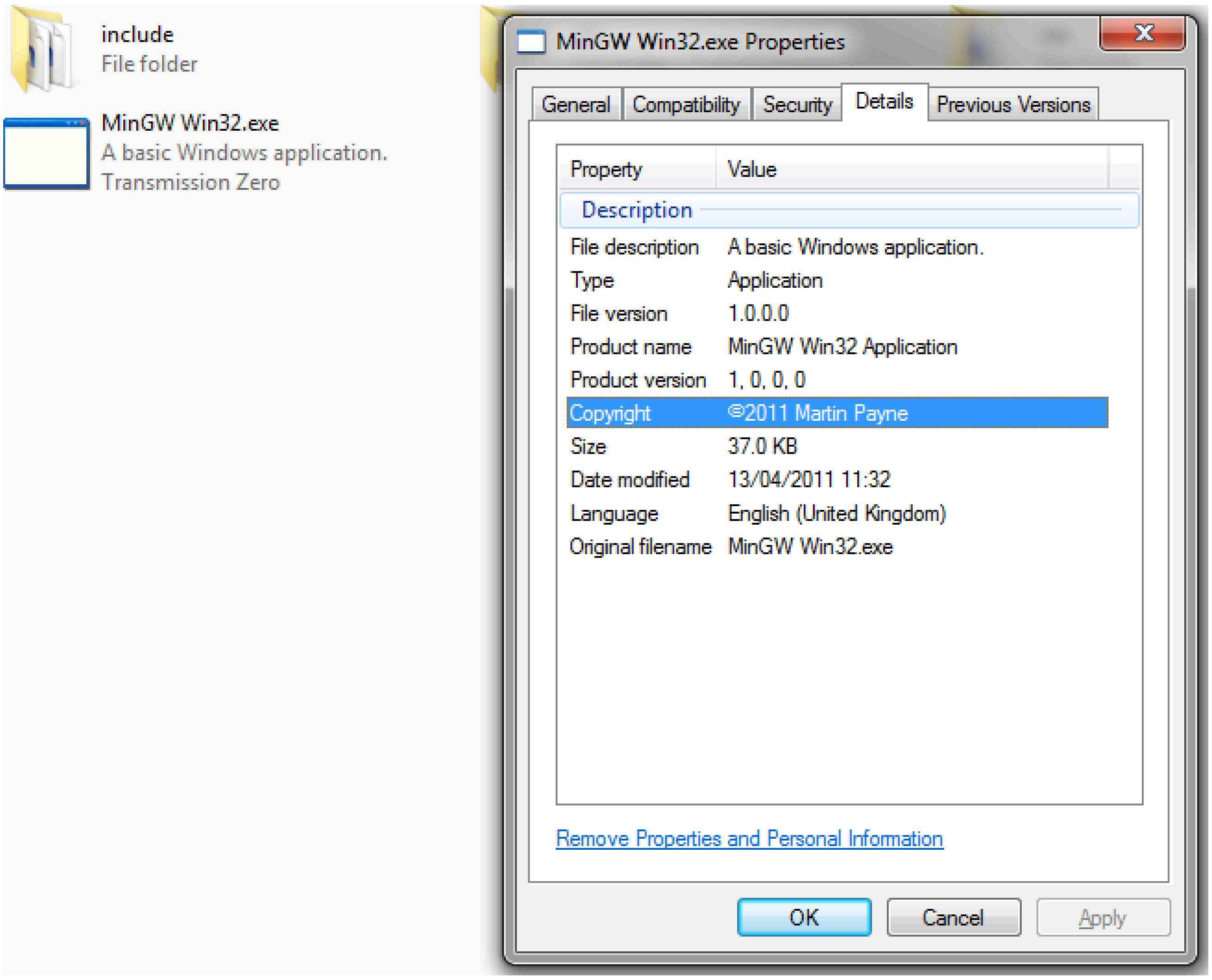
```
gcc -D UNICODE -D _UNICODE -c myfile.c -o myfile.o
```

You can switch between Unicode and ANSI builds simply by including or omitting these preprocessor definitions. As I mentioned though, it's recommended that you always use the Unicode functions for your applications. If you use ANSI functions on Windows NT, then Windows must allocate a buffer in memory, perform an ANSI to Unicode conversion, and then perform a function call to the Unicode version of the function. Likewise if the function you are calling is returning a string, Windows must convert it from Unicode to ANSI before handing it back to you. By using Unicode throughout your code, you avoid all of this overhead.

One thing to note is that Visual C++ supports a "`wWinMain`" entry point where the "`lpCmdLine`" parameter is a "`LPWSTR`". You would typically use the "`_tWinMain`" preprocessor definition for your entry point and declare "`LPTSTR lpCmdLine`" so that you can easily support both ANSI and Unicode builds. However, the MinGW CRT (C Runtime) startup library does not support wWinMain, so you'll have to stick with the standard "`WinMain`" and use "`GetCommandLine()`" if you need to access command line arguments.

The Version Information Resource

It's possible to add information into the executable, containing information such as the version, author, copyright, and description. This is done with a version information resource, and means Windows Explorer is able to display this when browsing folders or looking at the executable's properties page:



Property	Value
Description	
File description	A basic Windows application.
Type	Application
File version	1.0.0.0
Product name	MinGW Win32 Application
Product version	1, 0, 0, 0
Copyright	©2011 Martin Payne
Size	37.0 KB
Date modified	13/04/2011 11:32
Language	English (United Kingdom)
Original filename	MinGW Win32.exe

Adding a version information resource is simple a case of defining it in your resource script:

```
// Executable version information.
VS_VERSION_INFO    VERSIONINFO
FILEVERSION        1,0,0,0
PRODUCTVERSION     1,0,0,0
FILEFLAGSMASK      VS_FFI_FILEFLAGSMASK
#ifdef _DEBUG
  FILEFLAGS        VS_FF_DEBUG | VS_FF_PRERELEASE
#else
  FILEFLAGS        0
#endif
FILEOS             VOS_NT_WINDOWS32
FILETYPE           VFT_APP
FILESUBTYPE        VFT2_UNKNOWN
BEGIN
  BLOCK "StringFileInfo"
  BEGIN
    BLOCK "080904b0"
    BEGIN
      VALUE "CompanyName", "Transmission Zero"
      VALUE "FileDescription", "Win32 Test application"
      VALUE "FileVersion", "1.0.0.0"
      VALUE "InternalName", "Win32App"
      VALUE "LegalCopyright", "©2013 Transmission Zero"
      VALUE "OriginalFilename", "Win32App.exe"
      VALUE "ProductName", "Win32 Test application"
      VALUE "ProductVersion", "1.0.0.0"
    END
  END
  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x809, 1200
  END
END
```

I won’t go into detail about what each of these means, as it’s covered very well in the MSDN (http://msdn.microsoft.com/en-us/library/aa381058.aspx), but most of it is self-explanatory (I’d suggest reading the MSDN article anyway, particularly if the language is American English or anything other than British English).

Main Menu

The main menu is defined as a resource, rather than creating it programatically:

```
// Our main menu.
IDR_MAINMENU MENU
BEGIN
  POPUP "&File"
  BEGIN
    MENUITEM "E&xit",                  ID_FILE_EXIT
  END
  POPUP "&Help"
  BEGIN
    MENUITEM "&About",                 ID_HELP_ABOUT
  END
END
```

We associate the menu with our main window by setting the `lpszMenuName` property of the `WNDCLASSEX` structure, and handle the messages sent from the menu in the main window procedure:

```
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
  static HINSTANCE hInstance;

  switch (msg)
  {
    case WM_COMMAND:
    {
      switch (LOWORD(wParam))
      {
        case ID_HELP_ABOUT:
        {
          DialogBox(hInstance, MAKEINTRESOURCE(IDD_ABOUTDIALOG), hWnd,
&AboutDialogProc);
          return 0;
        }

        case ID_FILE_EXIT:
        {
          DestroyWindow(hWnd);
          return 0;
        }
      }
      break;
    }

    /* Other message handlers omited from code snippet. */
  }

  return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

About Dialog

The about dialog is also defined as a resource, rather than creating it progamatically:

```
// Our "about" dialog.
IDD_ABOUTDIALOG DIALOGEX 0, 0, 147, 67
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
BEGIN
  ICON             IDI_APPICON,IDC_STATIC,7,7,20,20
  LTEXT            "Win32 Test application.",IDC_STATIC,34,7,86,8
  LTEXT            "©2013 Transmission Zero",IDC_STATIC,34,17,84,8
  DEFPUSHBUTTON    "OK",IDOK,90,46,50,14,WS_GROUP
END
```

The dialog procedure does nothing more than handle the `WM_INITDIALOG` message and handle closing of the dialog, either via the “Ok” button, or via the close button in the Window frame:

```
INT_PTR CALLBACK AboutDialogProc(HWND hwndDlg, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
  switch (uMsg)
  {
    case WM_COMMAND:
    {
      switch (LOWORD(wParam))
      {
        case IDOK:
        case IDCANCEL:
        {
          EndDialog(hwndDlg, (INT_PTR) LOWORD(wParam));
          return (INT_PTR) TRUE;
        }
      }
      break;
    }

    case WM_INITDIALOG:
      return (INT_PTR) TRUE;
  }

  return (INT_PTR) FALSE;
}
```

The only thing to note is that this dialog procedure returns a pointer to an integer rather than the `BOOL` data type which you will see in many books and tutorials. It might seem strange that you are casting a boolean value to a pointer to an integer, but remember that some dialog messages require a pointer be returned. Whilst on 32 bit versions of Windows, a `BOOL` and `INT_PTR` are both 4 bytes in size, on 64 bit versions of Windows the `INT_PTR` is 8 bytes. If you use a `BOOL` as your return type and compile a 64 bit version of your application, any `INT_PTR`s returned from the dialog would be truncated from 8 bytes to 4 bytes, and your application would cause an access violation and crash when the pointer is dereferenced. Always use an `INT_PTR` return type for your dialog procedures, to allow 64 bit builds of your application without source code changes.

Keyboard Accelerators

Pressing "`Ctrl + A`" in the application will show the "about" dialog. Whilst it's not the greatest use of a keyboard accelerator, it does demonstrate their use and implementation. The keyboard accelerators are defined in the resource script:

```
// Our accelerators.
IDR_ACCELERATOR ACCELERATORS
BEGIN
  "A",            ID_HELP_ABOUT,      VIRTKEY, ALT, NOINVERT
END
```

Notice that the ID of the accelerator is "`ID_HELP_ABOUT`", which is the same as the one found in our main menu. This is no coincidence, as keyboard accelerators generally duplicate menu functionality, and by using the same ID you don't need to configure any additional message handling in your window procedure—both menu and keyboard accelerator messages are handled by your "`WM_COMMAND`" handler. You need only load the accelerators before your message pump begins, and add some additional message preprocessing code:

```
// Load accelerators.
hAccelerators = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDR_ACCELERATOR));

/* Some non-accelerator intermediate code ommited. */

// Main message loop.
while(GetMessage(&msg, NULL, 0, 0) > 0)
{
  if (! TranslateAccelerator(msg.hwnd, hAccelerators, &msg))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

Further information on keyboard accelerators can be found in the MSDN Keyboard Accelerators (http://msdn.microsoft.com/en-us/library/ms645526.aspx) article.

System Menu

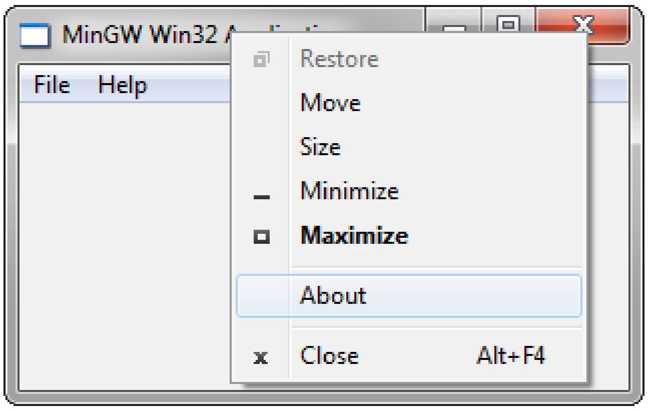
The system menu usually contains commands such as close, minimize, and maximize. It can be invoked by right clicking the title bar of a window, and on some versions of Windows by right clicking the taskbar icon. It’s possible to add additional items to this menu, and in our example we have added the ability to show the “about” dialog. Again, it’s not the greatest use of Windows functionality, but it does demonstrate how it works, and you are encouraged to find a much better use (see the Windows command prompt window, for example). The system menu additions are done entirely in code:

```
// Add "about" to the system menu.
hSysMenu = GetSystemMenu(hWnd, FALSE);
InsertMenu(hSysMenu, 5, MF_BYPOSITION | MF_SEPARATOR, 0, NULL);
InsertMenu(hSysMenu, 6, MF_BYPOSITION, ID_HELP_ABOUT, TEXT("About"));
```

The messages sent to your window procedure by the system menu are “WM_SYSCOMMAND” messages, rather than “WM_COMMAND”, so you need to create an additional message handler:

```
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
  static HINSTANCE hInstance;

  switch (msg)
  {
    /* Other message handlers ommited from code snippet. */

    case WM_SYSCOMMAND:
    {
      switch (LOWORD(wParam))
      {
        case ID_HELP_ABOUT:
        {
          DialogBox(hInstance, MAKEINTRESOURCE(IDD_ABOUTDIALOG), hWnd,
&AboutDialogProc);
          return 0;
        }
      }
      break;
    }

    /* Other message handlers omitted from code snippet. */
  }

  return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

The resulting menu, with a separator and "About" item, looks like the following on my Windows 7 machine:



Miscellaneous Functionality

The remaining functionality is fairly standard to Windows applications. I won't cover this in great detail, as you can find out more about it in the MSDN or your favourite Windows programming book.

Putting It All Together

Now we have all of the ingredients, we need to compile the code and resource files and link them all into an executable application. Assuming a very simple application which has "myfile.c" containing all of the code, "resource.rc" containing the resources, and an output of "myapplication.exe", the steps to build the application would be:

```
gcc -D UNICODE -D _UNICODE -c myfile.c -o myfile.o
windres -i resource.rc -o resource.o
gcc -o myapplication.exe myfile.o resource.o -s -lcomctl32 -Wl,--subsystem,windows
```

You may need to adjust the command names slightly if you aren't using Windows, for example on Fedora they are named "`i686-pc-mingw32-gcc`" and "`i686-pc-mingw32-windres`", but the operation of them should be identical.

The `-Wl,--subsystem,windows` linker switch ensures that the application is built as a Windows GUI application, and not a console application. Failing to do so would result in a console window being displayed whilst your application runs, which is probably not what you want. We must also link with *comctl32*, which we require due to using visual styles in our application, so we include `-lcomctl32` on the linker command line too.

Of course, the above build steps are the minimum needed to build an application, and you should include any other compiler and linker flags as needed, for example to enable warnings or perform optimisation of the generated machine code. A slightly more complete set of build steps can be performed using the Makefile contained in the file download.

Download the MinGW Win32 Application

The easiest way to get the application code is to run "`git clone https://github.com/TransmissionZero/MinGW-Win32-Application.git`". If you would prefer to download a source zip archive or tarball, you can download a MinGW Win32 Application release. You are free to use the code for whatever purpose you see fit (see the "License.txt" for the full terms of use). Using the Makefile, you can easily build the application from this article, and use it as a basis for your own Windows applications.

If you're looking to create an MDI (Multiple Document Interface) application, you can instead run "`git clone https://github.com/TransmissionZero/MinGW-Win32-MDI-Application.git`", or download a MinGW Win32 MDI Application source release.