Windows Programming/Resource Script Reference

< Windows Programming

This appendix page will attempt to list the different types of resources, and will attempt to show how to use those resources.

Contents

General construction [edit]

Resource script files are human-readable text files in either ANSI or Unicode (more strictly: UTF-16 with *byte order mark* (BOM)) format. To intermix different languages in ANSI format, a quirks `#pragma` exist to switch the code page in between. Unicode, `#pragma` switch, and the `LANGUAGE` statement are only supported for Win32.

A typical small file may look like

```
#include <windows.h>
#define IDC_STATIC -1

100 ICON "ProgIcon.ico"

10 MENU
{               // or BEGIN
 POPUP "&File"
 {
  MENUITEM "&Exit",IDCANCEL
 }
}               // or END
```

Using *curly braces* or BEGIN/END depends on your preference. The old, space-wasting style is the BEGIN/END pair, coming from the Pascal heritage of MacOS and Win16 API calls. C programmers typically prefer curly braces. The Visual Studio Resource Editor always generates BEGIN/END pairs, besides lots of housekeeping stuff.

Beginning with some header inclusion and `#define` statements, every resource is included as either

```
id_of_resource  resource_type  [memory management flags]  "filename"
```

or

```
id_of_resource  resource_type  [memory management flags]
BEGIN
 subsequent data
END
```

An exception from this rule is

- The `LANGUAGE` statement, can be placed almost everywhere (Win32 only)
- The `DIALOG` and the `VERSIONINFO` resource, there are additional statements between heading line and BEGIN
- The `STRINGTABLE` resource, where no resource ID is before the keyword. Instead, every string is prefixed with an ID

`id_of_resource` and `resource_type` can be either a string or a number. No quotes are there! A number is the preferred method for identification. All predefined resource types are numbers. But watch out! If you use an unknown ID for the resource compiler (let's say MANIFEST for Visual Studio 6), you will get neither error nor warning, and the resource is built with string resource type. Windows XP, using `ExecProcess()`, will not find that intended manifest and your program will show up in old visual style.

Conditional compilation with #if / #ifdef / #endif is also supported.

Expressions for IDs are limited to very simple math, no boolean operators are permitted.

Under the hood [edit]

Resources are compiled into a three-level directory structure:

1. Resource type (MENU, DIALOG etc.)
2. Resource ID (the number that is ahead the Resource Type – for STRINGTABLE, the ID of a group of upto 16 strings)
3. Resource language (the currently active language, given by command-line option or LANGUAGE statement; Win32 only)

The content of following data depends on actual resource type. Mostly, it's binary.

Read access to the binary data of an arbitrary resource is done with

```
FindResource()          // get a handle
LoadResource()          // get the binary size
LockResource()          // get a pointer; Win32: This is a simple macro, Win16: This is
a function call.
…                       // do something
```

```
UnlockResource()        // Win32: This is a do-nothing macro, Win16: This is a function
call.
FreeResource()          // release
```

Because bitmap, icon, cursor, dialog, string table, and menu resources are not officially documented and a bit hard to parse, programmers should use specialized resource load functions for these resource types instead. See description and examples for these types below.

Identifiers [edit]

Identifiers are generally named in a certain way, although the reader and all programmers are free to alter this naming scheme. It is simply a suggestion. Identifiers generally start with the prefix "ID", followed by a letter that denotes the type of identifier:

- IDS: A string resource
- IDM: A menu resource
- IDC: A command identifier
- IDD: A dialog box resource
- IDA: An Accelerator table resource
- IDI: An Icon or bitmap resource
- IDB: A Bitmap resource
- ID: A custom resource, or an uncommon resource type.

Sometimes, the command identifiers in a menu are given an "IDM_" prefix, to distinguish between commands from other sources.

There is no need to use symbolic identifiers. In some cases, identifiers complicate access to numerically adjanced controls in a dialog or menu. In any case, identifiers don't help non-English programmers to read a software source. Numbers never need translation. And both identifiers and numbers need explanation.

IDs are allowed in range 0..65535 and preferred in range 1..32767.

LANGUAGE [edit]

This keyword has different scope:

- Local (for one resource) if located below the resource line, like

```
21 MENU
LANGUAGE 7,1 // or, LANG_GERMAN, SUBLANG_GERMAN
{
 POPUP "&Datei"  // = "&File"
 …
```

This language applies to that menu

- Global (for all next resources) if located somewhere else

Language-neutral resources, like culture-free icons, VersionInfo, and Manifests, should be always set to LANGUAGE 0,0 (or, more verbously, LANG_NETUTRAL,SUBLANG_NEUTRAL).

Note to always check images against culture dependency! A typical mistake is the plug symbol for a mains-supplied notebook: It shows undoubtly an American plug, even in Europe. Obviously, images containing letters or text are culture-dependent.

Memory Management Flags [edit]

There are some Memory Management Flags from Win16 heritage, like MOVEABLE, FIXED, etc. See LocalAlloc() for some flags.

DISCARDABLE [edit]

Resources are loaded into memory when the program is run. However, if a resource is not in use, and if Windows does not need them immediately, resources can be optionally unloaded from memory until needed. To specify that it is okay to unload an unused resource from memory, you may list the **DISCARDABLE** keyword with the resource. DISCARDABLE resources allow more efficient memory usage, but can slow down your program if they need to be loaded from disk.

The DISCARDABLE keyword is ignored for 32-bit Windows, but remains for compatibility. [1] 32 bit resources are never loaded but mapped into memory.

Icons [edit]

Icons can be stored in a resource file using the `ICON` keyword. Here is a general example of using an icon in a resource script:

```
IDI_ICON<n> ICON [DISCARDABLE] "iconfile.ico"
```

Windows Explorer will display the binary executable with the first icon from the script. For instance, if we load two icons, as such:

```
IDI_ICON1 ICON DISCARDABLE "icon1.ico"
IDI_ICON2 ICON DISCARDABLE "icon2.ico"
```

And we define our macros as such in the corresponding `resource.h`:

```
#define IDI_ICON1 1
#define IDI_ICON2 2
```

The executable file will have *icon1.ico* as its icon.

To load an icon from an executable module, assuming we have an instance handle to the module (`hInst` in the following example), we can get a handle to the icon as such:

```
HICON hIcon;
hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_ICON1));
```

This will return a handle to the icon associated with the identifier "IDI_ICON1". Icon identifiers are generally prefixed with an "IDI_" which is short for "ID for an Icon".

The second parameter to the `LoadIcon()` function is a pointer to a string. String pointers are 32 bit values. However, if the most significant 16 bits are all zero, Windows will treat the value as a resource number, and not a string. To make the conversion between a string and a 16-bit integer, Microsoft provides the **MAKEINTRESOURCE** macro. Similarly, we could have used a string to define our Icon:

```
MYICON1 ICON DISCARDABLE "icon1.ico"
```

And we could load this string by name:

```
HICON hIcon;
hIcon = LoadIcon(hInst, "MYICON1");
```

String identifiers for resources are case insensitive.

`WNDCLASSEX` has handle values for two icons: a large icon and a small icon. The small icon is the icon used in the upper-left corner. Small icons are generally 16 pixels square. Larger icons are 32 pixels square. If no small icon handle is provided, the large icon will be shrunk down to fit.

If the `LoadIcon()` function is supplied with a NULL instance handle, Windows will supply a default icon for use.

Recently, the Win32 API provides the **LoadImage**function for loading icons, bitmaps, and mouse cursors from a single function. You can find more information about this function on MSDN.

Internally, Icons are stored under numeric resource type RT_ICON == 3, and are grouped under RT_GROUP_ICON == 14.

Bitmaps [edit]

Bitmaps can be loaded similarly to Icons in resource files:

```
(bitmap ID or name) BITMAP [DISCARDABLE] "bitmapfile.bmp"
```

Bitmaps can be accessed with the aptly named **LoadBitmap** function (again, new versions of the Win32 API prefer you use **LoadImage** to load a bitmap, icon, or cursor). LoadBitmap returns an HBITMAP handle type:

```
HBITMAP hBmp;
hBmp = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_BITMAP1));
```

Or, if we have named our bitmap resource:

```
hBmp = LoadBitmap(hInst, "MyBitmapRes");
```

Bitmaps are large resources, and if windows can't load the bitmap into memory (or if the ID or name value is invalid), the function will return a NULL value. Make sure you test this value before you use the handle.

Bitmaps must be unloaded from memory by passing the handle to the DeleteObject() function. You can find more information about this on MSDN

Bitmap identifiers generally use a "IDB_" prefix, to indicate that it is the ID of a bitmap.

Internally, Bitmaps are stored under numeric resource type RT_BITMAP == 2.

Mouse Cursors [edit]

Mouse cursors are specified similarly to icons and bitmaps, and are loaded with the **LoadCursor**function.

Internally, cursors are stored under numeric resource type RT_CURSOR == 1, and are grouped under RT_GROUP_CURSOR == 12.

As for any resource that implies binary data, the use of an external file with "filename" is recommended. However, most resource compilers allow to inline binary data to the resource file in this way:

```
42 ICON
{
 123,4567,0x89AB,0xCDEF
 '\x01','\x23',"ajx"
}
```

with this rule, coming from Win16 heritage:

- Numbers (decimal or hexadecimal) are stored contiguously as 16-bit little-endian quantities (unaligned)
- Characters are stored as 8-bit quantities

String Tables [edit]

A resource script can have many string tables, although this is unneccessary: the tables aren't differentiated (i.e. they get merged), and each string object, in any table, must have a unique identifier. Strings in a string table also may not use names, but instead must use numeric identifiers. After all, it doesn't make any sense to have to address a string with a string, does it?

Here is a general string table:

```
STRINGTABLE DISCARDABLE
BEGIN
   IDS_STRING1, "This is my first string"
   IDS_STRING2, "This is my second string"
   ...
END
```

It is important to note that in place of the BEGIN and END keywords, the programmer may also use the more C-like curly brackets, as such:

```
STRINGTABLE DISCARDABLE
{
   IDS_STRING1, "This is my first string"
   IDS_STRING2, "This is my second string"
   ...
}
```

Some people prefer one over the other, but they are all the same to the resource compiler.

Strings can be loaded using the **LoadString**function. LoadString is more involved then the LoadBitmap or LoadIcon functions:

```
int LoadString(HINSTANCE hInstance, UINT uID, LPTSTR lpBuffer, int nBufferMax);
```

The hInstance parameter, as we know, is the instance handle for the module that contains the string. The uID parameter contains the string number that we are trying to access. lpBuffer is the character array variable that will receive the string, and the nBufferMax number tells windows what the maximum number of characters that can be loaded is. This count is a security precaution, so make sure not to allow Windows to write character data beyond the end of the string. MSDN displays a large warning on the page for this function, and it is important that programmers heed this warning. msdn

Windows will automatically zero-terminate the string, once it is written to the buffer. LoadString will return the number of characters that were actually written into the string, in case the number of characters is less then the maximum number allowed. If this return value is 0, the string resource does not exist, or could not be loaded.

Strings can have "\0" in the middle. As strings are saved as counted strings, LoadString returns the number of characters saved, including the zeroes in between. But most Resource Editors fail with such strings.

Internally, stringtables are stored under numeric resource type RT_STRING == 6, in groups of upto 16 adjanced IDs.

Accelerators [edit]

Keyboard accelerators are a common part of nearly every windows application, and therefore it is a good idea to simplify the job of creating accelerators by putting them in a resource script. Here is how to create an accelerator table:

```
(Accelerator Table ID or name) ACCELERATORS [DISCARDABLE]
BEGIN
   (key combination), (Command ID)
   ...
END
```

Key combinations are specified in terms of either a string literal character ("A" for instance) or a virtual key code value. Here are some examples:

```
IDA_ACCEL_TABLE ACCELERATORS DISCARDABLE
BEGIN
   "A", IDA_ACTION_A //Shift+A
END
```

Now, when the key combination "Shift+A" is pressed, your window procedure will receive a WM_COMMAND message with the value IDA_ACTION_A in the WPARAM field of the message.

If we want to use combinations of the "Alt" key, or the "Ctrl" key, we can use the ALT and CONTROL keywords, respectively:

```
IDA_ACCEL_TABLE ACCELERATORS DISCARDABLE
BEGIN
   "a", IDA_ACTION_A, ALT          //Alt+A
   "b", IDA_ACTION_B, CONTROL      //Ctrl+B
   "c", IDA_ACTION_C, ALT, CONTROL //Alt+Ctrl+A
END
```

Also, we can use the "^" symbol to denote a CONTROL key code:

```
IDA_ACCEL_TABLE ACCELERATORS DISCARDABLE
BEGIN
   "^a", IDA_ACTION_A //Control+A
END
```

Similarly, if we want to be super hackers, would could use the ASCII code directly:

```
IDA_ACCEL_TABLE ACCELERATORS DISCARDABLE
BEGIN
   65, IDA_ACTION_A, ASCII //65 = "A", Shift+A
END
```

Or, we could refer to keys (including non-alphanumeric keys) with their Virtual Key Code identifiers, by using the VIRTKEY identifier:

```
IDA_ACCEL_TABLE ACCELERATORS DISCARDABLE
BEGIN
   VK_F12, IDA_ACTION_F12, VIRTKEY             //press the "F12 Key"
   VK_DELETE, IDA_ACTION_DEL, VIRTKEY, CONTROL //Ctrl+Delete
END
```

Now, If we make an accelerator correspond to a menu command, the menu command will light up when we press the accelerator. That is, the menu will light up unless we specify the "NOINVERT" keyword:

```
IDA_ACCEL_TABLE ACCELERATORS DISCARDABLE
BEGIN
   "A", IDA_ACTION_A, NOINVERT //Shift+A (non inverted menu selection)
END
```

To Load an accelerator table, we need to use the **LoadAccelerators** function, as such:

```
HACCEL hAccel;
hAccel = LoadAccelerators(hInst, MAKEINTRESOURCE(IDA_ACCEL_TABLE));
```

Again, we could have given our resource a string name, and used that string to load the table.

When using accelerators, we need to alter our message loop to intercept the keypress messages, and translate them into command messages according to our accelerator table rules. We use the **TranslateAccelerator** function, to intercept the keypress messages, and translate them into command messages, as such:

```
while ( (Result = GetMessage(&msg, NULL, 0, 0)) != 0)
{
    if (Result == -1)
    {
        // error handling
    }
    else
    {
```

```
        if (!TranslateAccelerator(hwnd, haccel, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
}
```

Also, if we are writing an MDI application, we need to intercept Accelerator messages from the child windows, we use the **TranslateMDISysAccel**function also:

```
while ( (Result = GetMessage(&msg, NULL, 0, 0)) != 0)
{
    if (Result == -1)
    {
        // error handling
    }
    else
    {
        if (  !TranslateMDISysAccel(hwndClient, &msg)
           && !TranslateAccelerator(hwndFrame, haccel, &msg) )
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
}
```

Where "hwndFrame" is the handle to the frame window, and "hwndClient" is the handle to the MDI client window.

Internally, Accelerators are stored under numeric resource type RT_ACCELERATOR == 9.

Menus [edit]

Menus can be defined in a resource script using the MENU keyword. There are 2 types of items that appear in a menu, the top level "POPUP" menu items, and the secondary "MENUITEM" items. These are defined in a menu as such:

```
(ID or name) MENU [DISCARDABLE]
BEGIN
   POPUP "File"
   POPUP "Edit"
   BEGIN
      MENUITEM "Copy", IDM_EDIT_COPY
      MENUITEM "Paste", IDM_EDIT_PASTE
   END
   ...
 END
```

We have included a few examples here, so that you can see the difference between a POPUP and a MENUITEM. When we have a menu with the ID_MENU identifier, we can load it into our program as such:

```
HMENU hmenu;
hmenu = LoadMenu(hInst, MAKEINTRESOURCE(ID_MENU));
```

Once we have this handle, we can pass it to the CreateWindow function, and apply it to our window.

When a menu item is selected, the host program receives a WM_COMMAND message, with the menu item identifier in the WPARAM parameter. If we have a basic window procedure switch-case statement, we can see this as follows:

```
case WM_COMMAND:
   switch(WPARAM)
   {
       case IDM_EDIT_COPY:
          //handle this action
          break;
       case IDM_EDIT_PASTE:
          //handle this action
          break;
   }
   break;
```

In a menu, if we want to associate a menu item with an accelerator, we can define it as such:

```
ID_MENU MENU DISCARDABLE
BEGIN
   POPUP "File"
   POPUP "Edit"
   BEGIN
      MENUITEM "&Copy", IDM_EDIT_COPY
      MENUITEM "&Paste", IDM_EDIT_PASTE
   END
   ...
 END
```

Notice how we put the ampersand (&) in front of the "C" in "Copy" and the "P" in "Paste". This means that those letters will be underlined, but more importantly, if an accelerator key combination is pressed, those items in the menu will be highlighted (unless the NOINVERT tag is specified in the accelerator table). If an ampersand is placed before a POPUP menu item, pressing ALT+ that letter will popup that menu. For instance, lets define our menu:

```
ID_MENU MENU DISCARDABLE
BEGIN
   POPUP "&File"
   POPUP "&Edit"
   BEGIN
      MENUITEM "Copy", IDM_EDIT_COPY
      MENUITEM "Paste", IDM_EDIT_PASTE
   END
   ...
 END
```

Now, if we press ALT+F, we will pop open the File menu, and if we press ALT+E it will open the Edit menu. That's pretty nice functionality for only a single extra character to type.

Internally, Menus are stored under numeric resource type RT_MENU == 4.

Version Information [edit]

A program can include certain information about its version, and its author in a resource script. This version information appears when you right-click the executable in Windows, and click "Properties". In the properties dialog box, this information appears on the "Version" tab.

```
BLOCK "StringFileInfo"
  BEGIN
      BLOCK "040904E4"
      BEGIN
          VALUE "CompanyName",      "My Company.\0"
          VALUE "FileDescription",  "A Win32 program."
          VALUE "FileVersion",      "1.0.0.0\0"
          VALUE "ProductName",      "The product name.\0"
          VALUE "ProductVersion",   "1.0\0"
          VALUE "LegalCopyright",   "My Company.\0"
      END
  END
  BLOCK "VarFileInfo"
  BEGIN
      VALUE "Translation", 0x409, 1252
  END
```

Internally, VersionInfo is stored under numeric resource type RT_VERSION == 16. It was introduced with Windows 3.

Dialog Boxes [edit]

Dialog box resources follow a general pattern:

```
(Dialog ID or name) DIALOG [DISCARDABLE] x, y, width, height
TITLE "(dialog box title)"
[CLASS "(class name)"]
FONT "(font name)"
BEGIN
   ...
END
```

if a dialog box is not being associated with a class, the CLASS field does not need to be filled in. All strings listed as being in quotes *must be in quotes in the resource script* or there will be an error. Individual items in a dialog box are then specified between the BEGIN and END tags.

Internally, Dialogs are stored under numeric resource type RT_DIALOG == 5.

Generic Controls [edit]

CONTROL classname,windowname,id,left,top,width,height,windowflags

Specific Buttons [edit]

Edit Boxes [edit]

Manifests [edit]

Manifest resources contain UTF-8 encoded XML description of operating system and DLL dependency. Mostly, this resource dictates to use the Windows XP Version 6.0 comctl32.dll, to have Luna style for the standard and common controls.

Internally, Manifests are stored under numeric resource type RT_MANIFEST == 24. It was introduced with Windows 4.1.

User-type resources [edit]

User-type resources should use some greater resource type identifiers, or RT_RCDATA == 10.