# Game Programming Genesis
# Part II : Using Resources in Win32 Programs

by Joseph "Ironblayde" Farrell

## Introduction

Welcome back! As you may have guessed by the title, in this article I'm going to show you how to use resources in your Windows programs. Simply put, resources are binary data that's appended to your `.EXE` file after the actual program code. Using resources is easy to learn and has a lot of advantages. It allows the developer to consolidate a lot of data into one file, include custom icons and such things with their programs, and prevent users from altering that data. Windows supports a large number of resource types, so I'm just going to cover the ones I think are most convenient and easiest to learn: bitmaps, cursors, icons, menus, and string tables. After that, I'll show you how to create a custom resource type, so you can include anything you want.

Again, all you need to understand this article is a basic understanding of the C language. C++ always helps since Windows itself is object-oriented, but most of my code is straight C. Also, I will assume that you have read my previous article, "Beginning Windows Programming," or have the equivalent knowledge. I use and recommend the Microsoft Visual C++ compiler, but if you're using a different one, it's not a big deal. Ready? Here we go!

## Resource Scripts

Before we get into any of the specific resource types, we need to go over the method used to tell the compiler what resources to include, and how. This method is to use a special file called a resource script, which is simply a text file either written by the developer or automatically generated by Visual C++, or whatever IDE you happen to be using. Your script file should have the file extension `.rc`. Most of a script file is taken up by lines which define or specify the resources to include. The simplest of these lines is used by several resource types, and looks like this:

```
[identifier]  [resource type]  [filename]
```

The identifier can be one of two things: a string representing the resource, or a numeric constant that's `#define`d in a header file meant to accompany the resource script file. If you use numeric constants, which is usually a good idea, you can use the `#include` directive in your script file to include the header that corresponds to it. You can also use C-style comments to make things a little easier to understand. That said, here's what a very simple resource script file might look like:

```
#include "resource.h"

// icons
ICON_MAIN     ICON    myicon.ico

// bitmaps
IMG_TILESET1  BITMAP  tileset.bmp
IMG_TILESET2  BITMAP  tileset2.bmp
```

That's not too bad, right? There's one thing that can be confusing, though. Just by looking at my brief example, you can't tell if `ICON_MAIN` and `IMG_TILESET` are meant to be strings or numeric constants. The file would appear the same no matter which case were true. At compile time, your compiler will look at the identifiers you're using and search through your header files looking for their definitions. If no matching `#define` statements are found, it's assumed that you're using string identifiers.

Don't worry about the actual lines themselves just yet; I'll explain each type of entry when I get to that particular resource. If you don't want to bother with resource scripting at all, you can just insert the resources from your IDE (in Visual C++, go to "Resource..." under the Insert menu) and a resource script will be generated automatically. I prefer to do it myself with good old Notepad, but don't ask me why because I can't think of a good reason. :) Now that you know the basics of creating a resource script, let's get started on the specific resource types.

## Icons and Cursors

Most of the Windows programs you use every day have their own icons built in, and now you know how it works: they're simply resources included in the EXE file. Custom cursors that are used by those programs are also included as resources. You've already seen an example of the script line that includes an icon resource, and the line for cursors is very similar. Here they are:

```
[identifier]  CURSOR  [filename]
[identifier]  ICON    [filename]
```

After adding a line such as this to your script file -- make sure to include the script file in your project -- the icon or cursor specified by `[filename]` will be included as a resource in your EXE file. That's all there is to it! You can use any icon/cursor editor to generate the files you want to include. I use the one that's included in Visual C++.

Including the resources doesn't do a whole lot for your program, though, because you don't know how to use them yet! To get an idea for how icon and cursor resources are utilized in a program, let's revisit the window class we developed in the last article:

```
WNDCLASSEX sampleClass;                                  // declare structure variable

sampleClass.cbSize =        sizeof(WNDCLASSEX);          // always use this!
sampleClass.style =         CS_DBLCLKS | CS_OWNDC |
                            CS_HREDRAW | CS_VREDRAW;     // standard settings
sampleClass.lpfnWndProc =   MsgHandler;                  // message handler function
sampleClass.cbClsExtra =    0;                           // extra class info, not used
sampleClass.cbWndExtra =    0;                           // extra window info, not used
sampleClass.hInstance =     hinstance;                   // parameter passed to WinMain()
sampleClass.hIcon =         LoadIcon(NULL, IDI_WINLOGO); // Windows logo
sampleClass.hCursor =       LoadCursor(NULL, IDC_ARROW); // standard cursor
sampleClass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);  // a simple black brush
sampleClass.lpszMenuName =  NULL;                        // no menu
sampleClass.lpszClassName = "Sample Class"               // class name
sampleClass.hIconSm =       LoadIcon(NULL, IDI_WINLOGO); // Windows logo again
```

You remember this, don't you? The `hIcon` field specifies the icon to be used to represent the program, and the `hIconSm` field is the icon used on the Start Menu and the window's title bar. The `hCursor` field sets the cursor to be used when the mouse is within the boundaries of the window you create. I promised you we'd take a look at the functions used to fill these fields a little more closely, so here are their prototypes:

```
HICON LoadIcon(
    HINSTANCE hInstance,  // handle to application instance
    LPCTSTR lpIconName    // icon-name string or icon resource identifier
);

HCURSOR LoadCursor(
    HINSTANCE hInstance,   // handle to application instance
    LPCTSTR lpCursorName   // name string or cursor resource identifier
);
```

The return type is a handle to the cursor you're loading. The parameters are very straightforward:

**`HINSTANCE hInstance`**: This is a handle to the instance of your application. To load resources from your program, just pass the `HINSTANCE` that is passed to your `WinMain()` function when the program is executed. To use standard Windows resources like we did in the window class above, set this to NULL.

**`LPCTSTR lpIconName, lpCursorName`**: This is a string identifier that identifies the resource you want to load. If your script file refers to resources by string, simply pass the string. But if you're using numeric constants, the Windows header files include a macro that changes an integer to a form compatible with this parameter called `MAKEINTRESOURCE()`.

As an example, let's look at the line that sets the icon to represent the program. Suppose your resource script file looks like this:

```
#include "resource.h"

ICON_MAIN     ICON    myicon.ico
CURSOR_ARROW  CURSOR  arrow.cur
```

If the identifiers `ICON_MAIN` and `CURSOR_ARROW` do not have matching `#define` statements somewhere in resource.h, then you would pass the corresponding string to the appropriate resource-loading function, like this:

```
sampleClass.hIcon = LoadIcon(hinstance, "ICON_MAIN");
```

Now let's say that resource.h contains a few `#define` directives:

```
#define ICON_MAIN        1000
#define CURSOR_ARROW     2000
```

Now you have to use the `MAKEINTRESOURCE()` macro to turn the numerical identifier into something of type `LPCTSTR`. This gives you a little more ease of flexibility in loading resources. Any of the following calls would be correct:

```
sampleClass.hIcon = LoadIcon(hinstance, MAKEINTRESOURCE(ICON_MAIN));
  or...
sampleClass.hIcon = LoadIcon(hinstance, MAKEINTRESOURCE(1000));
  or...
int ident = 1000;
sampleClass.hIcon = LoadIcon(hinstance, MAKEINTRESOURCE(ident));
```

That's about all you need to know about including icons and cursors in your programs, but I'll mention one more thing while we're on the topic. If you want to set a cursor sometime other than at the beginning of the program, there's a simple Windows function you can use to accomplish this:

```
HCURSOR SetCursor(HCURSOR hCursor);
```

The one parameter is the handle you get by calling `LoadCursor()`, and the handle that is returned is a handle to the previous

cursor. If no previous cursor was set, the return value is NULL. Relatively painless, wouldn't you say? Let's move on to something a bit more interesting.

## Bitmaps

Including bitmap resources is probably the easiest way to add images to your program. Bitmaps are native to Windows and so there are functions included to deal with loading and manipulating them, but remember, if you include too many, you'll end up with an enormous .EXE file. In any case, you include bitmaps in your resource script file in basically the same way you handle icons and cursors:

```
[identifier]  BITMAP  [filename]
```

There is a function called LoadBitmap() that is analagous to LoadCursor() and LoadIcon(); it is used to retrieve a handle to a bitmap, but since I haven't talked about graphics yet, I won't describe this function here. You can probably guess exactly how it works, but once you have a handle to a bitmap, what would you do with it? More to come on that in the future, don't worry! For now, I just wanted to show you how to include a bitmap resource. Now let's look at something you can use right away.

## String Tables

The string table is one of my favorite resource types. It's exactly what you're thinking: a giant table full of strings. There are any number of purposes for using a string table. You can use it to store data filenames, character dialogue for a game, message-box text, text for menus that are generated by the program, anything you want. Creating a string table in your script file is easy. Here's what it looks like:

```
STRINGTABLE
{
// entries go here
}
```

An entry in a string table consists of a number to identify the string, followed by a comma, then the string itself, enclosed in double quotation marks. The strings in a string table can include escape sequences like \n or \t. Note that the string table itself does not have an identifier, so each program you write can include only one string table. A simple string table might look something like this:

```
// program information
STRINGTABLE
{
1, "3D Space Game v1.0"
2, "Written by The Masked Coder"
3, "(C) 2000 WienerDog Software"
}
```

To load a string from your program's string table, you use the -- you guessed it -- LoadString() function. Here is its prototype:

```
int LoadString(
    HINSTANCE hInstance,  // handle to module containing string resource
    UINT uID,             // resource identifier
    LPTSTR lpBuffer,      // pointer to buffer for resource
    int nBufferMax        // size of buffer
);
```

The integer returned by the function is the number of characters, excluding the terminating null character, that were successfully copied into the buffer. This corresponds to the length of the string. If you load a blank string, or if the function fails, the return value is 0. Take a look at the parameters:

**HINSTANCE hInstance**: Once again, this is the instance of your application.

**UINT uID**: This is the number that identifies the particular string you want to load.

**LPTSTR lpBuffer**: This is a pointer to the location you want the string copied to.

**int nBufferMax**: This is the size of the buffer in bytes. If the string to be loaded is longer than the buffer can hold, the string is truncated and null-terminated.

For example, to load WienerDog Software's copyright message, the following code would be used:

```
char buffer[80];
LoadString(hinstance, 3, buffer, sizeof(buffer));
```

Even though the declaration of a string table in your script file has to use numbers and not identifiers, I usually #define a number of string table constants in one of my header files when using a string table. For instance, to accompany the string table above, I might have a line like:

```
#define ST_WIENERDOGCOPYRIGHT  3
```

Your code will be much easier to read if you have `LoadString()` calls that use readable constants for the `uID` parameter, rather than just having the index numbers. This doesn't mean you should have a constant for every string table entry; that would take ages if you have a large string table. Usually I like to use one `#define` per "section" of the string table. For instance, `ST_FILENAMES` for the first index where filenames are stored, `ST_DIALOGUE` for the first index of the character dialog strings, etc.

## Menus

This is the last type of Windows resource I'll go over, and it's also one of the most useful. Menu resources are used to define the menu bar that would appear underneath the title bar of your application, and are loaded during the definition of the window class. Looking back, in the window class we developed during the last article, there was a line that looked like this:

```
sampleClass.lpszMenuName = NULL;
```

If you're creating a windowed application, chances are that you'll want to have a menu bar of some sort. This is done using the menu resource. The script file entry for this one can get a little complicated, but here is its most basic form:

```
[identifier] MENU
{
    POPUP [menu name]
    {
        MENUITEM [item name], [identifier]
    }
}
```

The identifier is what you're used to: either a string or a numeric constant that is used to refer to the menu. Within the `MENU` brackets, there can be one or more `POPUP` menus, each of which represent a pull-down menu, whose name is given by `[menu name]`. Within the `POPUP` brackets, there can be one or more `MENUITEM`s, each of which represents a final menu selection, with a name given by `[item name]` and an identifier that must be a numeric constant. Within the menu and item names, if you want that option to be accessible by a keyboard shortcut, you precede the letter of the shortcut with the ampersand (&). For instance, if you want to create a File menu accessible by pressing Alt+F, the menu name should be `&File`. Menu and item names should be enclosed in double quotation marks. With that, here is an example of a simple menu resource:

```
MAIN_MENU MENU
{
    POPUP "&File"
    {
        MENUITEM "&New",         MENUID_NEW
        MENUITEM "&Open...",     MENUID_OPEN
        MENUITEM "&Save",        MENUID_SAVE
        MENUITEM "Save &As...", MENUID_SAVEAS
        MENUITEM "E&xit",        MENUID_EXIT
    }
    POPUP "&Help"
    {
        MENUITEM "&Contents",    MENUID_CONTENTS
        MENUITEM "&Index...",    MENUID_INDEX
        MENUITEM "&About",       MENUID_ABOUT
    }
}
```

You can also create submenus by including one `POPUP` inside of another, specify menu items as being initially grayed or checked, or do several other more advanced things, but I'm not going to go into that here. To obtain a handle to a menu resource, use the `LoadMenu()` function whose prototype is shown here:

```
HMENU LoadMenu(
    HINSTANCE hInstance,  // handle to application instance
    LPCTSTR lpMenuName    // menu name string or menu-resource identifier
);
```

You should be used to these parameters by now. The first one is the instance of your application, and the second is the identifier you assigned to the menu. Remember to use `MAKEINTRESOURCE()` if you used a numerical constant. Now, to attach a menu to a window, you have two options. The first is to set the menu as the default for your window class, like this:

```
sampleClass.lpszMenuName = LoadMenu(hinstance, MAKEINTRESOURCE(MAIN_MENU));
```

The second option is to leave `lpszMenuName` equal to `NULL`, and attach a menu yourself later. This can be useful if you want to create two windows with different menus, but don't want to define two separate window classes. To attach a menu, use the `SetMenu()` function:

```
BOOL SetMenu(
    HWND hWnd,   // handle to window
    HMENU hMenu, // handle to menu
);
```

The return value is TRUE if the function succeeds, or FALSE if it fails. The parameters are pretty easy to figure out:

**HWND hWnd**: This is the handle to the window to which you want to attach the menu. Pass the handle that was returned when you called CreateWindowEx().

**HMENU hMenu**: To identify the menu, pass the handle returned by LoadMenu(). If you pass NULL, the specified window's menu is removed.

This resource is particularly nice because all the functionality of the menu is defined by that simple scripting. But what happens when the user selects a menu option? The answer is that Windows sends a WM_COMMAND message informing the program that it must take action. Let's pay a visit to our message-handling function and see if we can't figure out how to handle this.

## Handling Menu Events

As you probably remember, Windows messages are handled by a special callback function usually called WindowProc() or something similar. The simple one we wrote last time was called MsgHandler(), and its prototype looked like this:

```
LRESULT CALLBACK MsgHandler(
    HWND hwnd,     // window handle
    UINT msg,      // the message identifier
    WPARAM wparam, // message parameters
    LPARAM lparam, // more message parameters
};
```

When a menu message is sent, msg will be WM_COMMAND, and the menu item that was selected will be contained in wparam. This is why menu item identifiers can't be strings; they need to fit into the wparam parameter. More specifically, the menu item identifier is the low word of wparam. To extract the low or high word of a 32-bit variable type like WPARAM, LPARAM, int, etc. Windows provides macros called LOWORD() and HIWORD() that do the job. They are shown here:

```
#define LOWORD(l)   ((WORD) (l)) #define HIWORD(l)   ((WORD) (((DWORD) (l) >> 16) & 0xFFFF))
```

In the case of LOWORD(), the typecast to WORD simply truncates the value to the lower 16 bits. HIWORD() shifts the upper 16 bits to the right, then performs a logical AND with 0xFFFF just to be sure any bits above the lower 16 are all set to zero. If you're not familiar with the >> and << operators, they are bit shifts. The << operator shifts all the bits of a variable a number of positions to the left, and the >> operator shifts to the right. For example, suppose you had a 16-bit variable x whose value was 244. In binary this is 0000 0000 1111 0100. The following example shows a bit shift, and the effect on x:

```
short int x = 244, y;
y = x << 4;
```

Contents of x: **0000 0000 1111 0100**
Contents of y: **0000 1111 0100 0000**

Anyway, use LOWORD() to extract the low word of wparam, and you have the ID of the menu item that was selected. So, somewhere in your MsgHandler() function, you should have something like this:

```
// handle menu selections
if (msg == WM_COMMAND)
{
    switch (LOWORD(wparam))
    {
    case MENUID_NEW:
        // code to handle File->New goes here
        break;
        case MENUID_OPEN:
        // code to handle File->Open goes here
        break;

    // the rest of the option handlers go here

    }

    // tell Windows you took care of it
    return(0);
}
```

Make sense? Good. That about wraps it up for the specific resource types I'm going to cover. There are others, such as accelerator tables (tables full of keyboard shortcuts), HTML pages, WAV files etc. but I think these are the most useful. Before I wrap this up, though, there's one more very powerful feature of Windows programs I'm going to show you, and that's defining a custom resource type.

## Custom Resources

The standard Windows resources are those which have special functions for loading and handling them, but they are not the only types you can use. Resources can be any data you want them to be! Working with custom resources requires a little more work

since you must locate and read the resource data manually, but it's not too bad. The script file entry for a custom type follows the basic format you're already used to:

```
[identifier]  [resource type name]  [filename]
```

The resource type name is a string that defines your custom resource, and can be whatever you want. For the purposes of this example, let's say you want to include a data file called `p1config.dat` that contains information necessary to initialize a character in a game program. We'll call the custom resource type `CHARCONFIG`. With that in mind, here's an example of what the script file entry might look like for your data file:

```
DATA_PLAYERINIT  CHARCONFIG  p1config.dat
```

Pretty simple, hey? Now that you've included your file, there are three steps you must take in order to retrieve a pointer to the resource data. Each involves calling a function we haven't talked about yet, so let's go through them one at a time. The first thing you must do is to find the resource with a call to `FindResource()`. Here's the prototype:

```
HRSRC FindResource(
    HMODULE hModule, // module handle
    LPCTSTR lpName,  // pointer to resource name
    LPCTSTR lpType   // pointer to resource type
);
```

The return value is a handle to the resource's information block, or `NULL` if the function fails. The parameters are as follows:

**`HMODULE hModule`**: The `HMODULE` data type is simply an `HINSTANCE`. Don't ask me why they felt they needed another name for it, but you should simply pass the instance of your application. You don't even need a typecast because the data types are exactly the same.

**`LPCTSTR lpName`**: This is the resource identifier. Remember to use `MAKEINTRESOURCE()` on this one if you're using numeric constants to define your resources.

**`LPCTSTR lpType`**: This is the resource type, so pass the string you used to define your resource type. In our case, this is `CHARCONFIG`.

A sample function call looks like this:

```
HRSRC hRsrc = FindResource(hinstance, MAKEINTRESOURCE(DATA_PLAYERINIT), "CHARCONFIG");
```

This is a handle to the info block the resource resides in. The next step to getting a pointer to the data is to take this handle and pass it to `LoadResource()` to actually load the data. This yields a handle to the resource itself. Here is the function prototype:

```
HGLOBAL LoadResource(
    HMODULE hModule, // resource-module handle
    HRSRC hResInfo   // resource handle
);
```

The return type, `HGLOBAL`, is a pretty general handle type, as opposed to the other load functions we've covered, which returned specific handle types like `HBITMAP` or `HICON`. If the function fails, this value will be `NULL`. The parameters are straightforward:

**`HMODULE hModule`**: Again, simply the application instance.

**`HRSRC hResInfo`**: Pass the handle that was returned by `FindResource()`.

Now that you have a handle to the resource, you can finally get a pointer to the data that was in the resource file you included. This is achieved with a call to `LockResource()`, shown here:

```
LPVOID LockResource(HGLOBAL hResData);
```

Simply pass the handle that was returned by `LoadResource()`. If the return value is `NULL`, the function call failed. If not, you've got your pointer! Now you're free to do whatever you like with the data. Note that the return type is `LPVOID` (Windows-speak for `void*`), so if you want to use array notation on the pointer, you need to cast it to something like a `BYTE*`. Now that we've gone through all the steps, I'll show you an example of a function you might write to return a pointer to a specified resource:

```
UCHAR* LoadCustomResource(int resID)
{
    HRSRC hResInfo;
    HGLOBAL hResource;

    // first find the resource info block
    if ((hResInfo = FindResource(hinstance, MAKEINTRESOURCE(resID), "CUSTOMRESOURCETYPE")) == NULL)
        return(NULL);

    // now get a handle to the resource
    if ((hResource = LoadResource(hinstance, hResInfo)) == NULL)
```

```
            return(NULL);

        // finally get and return a pointer to the resource
        return ((UCHAR*)LockResource(hResource));
    }
```

## Closing

Well, that about does it for resources! See, programming for Windows is fun. :) Even with all this knowledge of resources, you're still pretty limited in what you can actually get your programs to do, so next time I'll be going over some basic Windows GDI (Graphics Device Interface) functions, so you can start using all this stuff to put some demo programs together. As always, send me your comments, your ideas, your death threats:

E-mail: ironblayde@aeon-software.com
ICQ: UIN #53210499

Farewell everyone, until we meet again...

**Discuss this article in the forums**

Date this article was posted to GameDev.net: **11/18/2000**
(Note that this date does not necessarily correspond to the date the article was written)

**See Also:**
General