Using Windows Resources

by Phil Brown

Resources are such an inescapable part of Windows programming that all modern development environments provide ways of hiding the nitty gritty detail of resource files from the application developer. While in general this allows for faster and more interactive development, many application developers are unfamiliar with resources and the techniques required for their effective use.

Before we consider employing resources in our applications, it is necessary to understand their nature. A resource file is simply a binary file containing arbitrary data which can be accessed using predefined identifiers. Resource files exist in two states: the .RC source text file and the .RES compiled binary file. Although some tools (including Delphi) manipulate resource files in their binary format directly for the visual design of forms, as developers it is only the .RC format that concerns us. Resource files are edited using any text editor in this format, and then a resource compiler (one is provided with Delphi) is used to produce the binary .RES file which can then be included in our applications. The resource compiler provided with Delphi 1 is brcc.exe and that for Delphi 2 and 3 is brcc32.exe. These are DOS based command line compilers, but at least you can create a new association for .RC files with the resource compiler in Windows Explorer to allow you to right click on a file and compile it. To enable this, select `View|Options` from the `Explorer` menu, click on the `File Types` tab that appears, then choose `New Type`. Type in `RC` into `Associated extension`, then click the `New` button. Enter `Compile` into the `Action` edit box in the new dialog, then use the `Browse` button to select the resource compiler from the bin subdirectory where Delphi was installed and click OK. If you like you can change the associated icon to a more distinctive one, and change the description of the file type that appears in the Windows Explorer panel.

The resource file format is very simple: all entries map an identifier onto some data. In order to complete the entry, an indication is required as to the type of the data: it could be a string, some icon information or simply some unknown data (a user defined data structure). The syntactical format of the declaration depends upon the type of data: for example, the declaration of an icon resource in the file is:

```
nameID ICON [[load-mem]] filename
```

The `[load-mem]` optional parameter is a set of flags indicating how the OS may handle the memory allocated to the resource when the application is run: they are generally used for Windows 3.1 compatibility and typically default values are acceptable. `nameID` is the general identifier by which the resource will be accessed, and can be either a textual string name or an integer. The `filename` parameter points to a standard Windows icon file, and this file must exist when the resource file is compiled (the file name can include a fully qualified path).

There are many different kinds of resources (or data) that can be defined in an .RC file, but most of these relate to Windows controls for C/C++ programs. It is possible to use these in Delphi, but the Delphi environment handles interface details much more cleanly, so there is little point. For custom resource files, however, there are a few resource types that are very useful: the `CURSOR`, `ICON`, `BITMAP`, `STRINGTABLE` and `RCDATA` resources. Listing 1 shows a complete compilable resource file declaring all of these types. Do not be confused by the `BEGIN..END` definition in the resource file, it does not follow standard Delphi syntax!

The example demonstrates two main things: that the general format for the types we are interested in using follows the same syntax as the `ICON` example earlier, and the `STRINGTABLE` resource allows us to bind a number of strings, each with a unique identifier, into the one declaration.

If we compile the .RC file, the result is a .RES file, but how do we access the resources in it? Resources form part of a compiled program, either an .EXE or a .DLL, and it is the job of the linker to bind the resources into the executable. First, however, the linker must be told to include the .RES file, and this is achieved in Delphi with the `$R` directive. Including the line `{$R MyResources.RES}` in *any* file included in the program will direct the linker to bind those resources into the executable.

You should already be familiar with the `$R` directive: Delphi uses the statement `{$R *.DFM}` in each form file to ensure that the .DFM resource file is bound in to the application. The .DFM file is a large custom data structure that Delphi uses to define component properties when forms are created, and is simply one example of the use of custom data structures held in resource files.

➤ *Listing 1: An example of a complete .RC resource file.*

```
CDAudioIcon ICON "Audio.ICO"
SplahScreenLogoBitmap BITMAP "Logo.BMP"
STRINGTABLE
BEGIN
  1, "Hello, world"
  2, "My first resource file!"
END
READMEFILE RCDATA "ReadMe.TXT"
```

As noted before, the identifiers in a resource file can be either string names or integer values. It is generally better to use integer identifiers as they are accessed more quickly than string identifiers. Rather than restrict you to using hard coded values, the resource file format allows integer expressions. The `#define` statement introduces a named constant and the `#include` statement allows the resource compiler to process another .RC format file in the middle of the original file. This has particular benefits as one .RC file can declare the constants used as resource identifiers and can be included in both the main .RC file (using `#include`) and a Delphi program (using the `{$I}` directive). Using this approach it can be guaranteed that both the compiled resources and the Delphi program are using the same integer values to identify resources. This is very convenient as it avoids maintaining two sets of identifiers, one for the Delphi program and another for the resource script. Listing 2 contains an example of an include file which declares two integer values as expressions, and Listing 3 shows an example of the resource script which includes the file and uses the constants.

Note that the Delphi resource compiler is quite happy interpreting limited Delphi syntax, such as comments, line terminating semicolons and the const keyword. You may use either of the paired commenting standards `{ ... }` and `(* ... *)`, but not the double slash comment `//`. This acceptance of such syntax greatly facilitates the use of shared constants in both Delphi source and resource files. Note that the use of a base identifier value (such as `TXT_BASE` in our example) is very common, as integer identifiers have an allowable range of 0..65,535 and offsetting all identifiers from a base value eases any allocation issues that may arise later.

Now that we know how to define resources we can start to put them to use in our applications. Apart from storing form descriptions for languages such as C++, one of the main uses of resource files is to store all of the string constants for an application. These constants are referenced only by identifier throughout the program, and storing all of the actual string values in a single file allows this file to be translated to a different language (say Italian) and a different set of resources bound into the application. This provides for easy localisation of applications that must be aware of national boundaries. Starting with Delphi 3, Inprise actually provided a new declaration called `resourcestring` which facilitates this process greatly, hiding most of the complexity of resource files from the developer. However, for Delphi 1 and 2, or for access to datatypes other than strings, a custom resource file is still the only answer.

There are a number of API calls which are used with resources, but due to Delphi's ability to hide most of the detail from us, we need concern ourselves with just a few. These are `LoadIcon`, `LoadBitmap` and `LoadString` which do pretty much what you expect, they provide access to the data types held within a resource file given an identifier. They also require a handle of the module that contains the resources, and fortunately Delphi provides us with the value we need for resources stored in applications in the global `HInstance` variable. The `LoadString` API call also requires parameters for a buffer to store the string (we can use a Delphi 2 long string, typecast as a `PChar`) and the length of the buffer. Delphi does provide a function called `LoadStr` that performs the above operations using a much simpler interface (with just the resource identifier as an integer parameter), but this function is not able to handle resources stored in anything other than the application. The result of the `LoadIcon` and `LoadBitmap` is a Windows resource handle, which we can simply assign to the `Handle` property of `TIcon` and `TBitmap`, Delphi takes care of the details of releasing the previous graphic data and displaying the new one. Listing 4 shows examples of loading these three resource types.

```
(* ResourceConstants.inc
 *
 * Declares constants for resource identifiers
 *)
const
  TxtBase = 10000;
  TxtHelloWorld = TxtBase + 1;
  TxtMessage = TxtBase + 2;
```

➤ *Listing 2: A resource script that can be included in both a Delphi program and a resource file.*

```
#include "ResourceConstants.inc"
STRINGTABLE
BEGIN
  TxtHelloWorld, "Hello, world"
  TxtMessage, "This is a resource file."
END
```

➤ *Listing 3: A resource file that includes another.*

```
var
  Icon1: TIcon;
  Bitmap1: TBitmap;
  MyStr: String;
procedure LoadResources;
begin
  (* note Icon1 and Bitmap1 must already have been created *)
  Icon1.Handle := LoadIcon (HInstance, Pointer (ResourceID));
  Bitmap1.Handle := LoadBitmap (HInstance, Pointer (ResourceID));
  SetLength (MyStr, 512);
  SetLength (MyStr, LoadString (HInstance, ResourceID, PChar (MyStr),
    Length (MyStr)));
end;
```

➤ *Listing 4: Some code loading data from resources.*

Note that a complication exists in the case for strings, which is that the buffer must be large enough to store the result obtained from the resources. The `LoadString` API call returns an integer indicating the length of the string, and so once the string has been loaded we should reset the length of the string using `SetLength`. To ensure the buffer is large enough to store the resource, it is set to be a value larger than the 256 characters which is supposed to be the maximum length of a resource string. The Delphi resource compiler does not always enforce this however, so a value of 512 is used.

Conserving Memory

As noted before, the first parameter to each of these API calls is a module handle, and so far we have used `HInstance` exclusively, for resources stored in the application itself. This is perfectly valid, but has the disadvantage that as the resources form part of the .EXE, they will be consuming system memory all of the time that the application is loaded. Although for string resources this is fine, if you have large resources such as bitmaps permanently loaded, this can be wasteful of memory. To this end, it can be very useful to store the resources in a DLL, which is dynamically loaded as the resources are required. In these cases the parameter passed as the module handle to the resource API calls is simply the DLL handle, ie that which is returned by the `LoadLibrary` call to dynamically load the DLL. Note that we cannot use statically linked DLLs, as these would consume a similar amount of memory as if we had bound the resources into the application.

On this issue's disk is the source code for a `TResourceLibrary` class which encapsulates access to icon, bitmap and string resources for both an application and a DLL. Listing 5 shows the code for the class interface. It can be seen that there are two constructors, the standard `Create` that provides access to the resources bound in to the current application, and a `CreateForDLL` constructor to which the name of a DLL is passed, and the class then provides access to the resources in the DLL. Two public methods are available, to load an icon and bitmap images, and a public property, providing indexed access to strings. To fully gain the benefit of this class you should use resource file constants in your application as previously discussed.

An example application is included on the disk that demonstrates access to all three types of resources. To compile the application, you must first compile the ResourceFile.RC file into a .RES file, and then compile the ResourceDLL project that binds the resources into a DLL. The ResourceDemo application can then be run which loads icons, bitmaps and strings from the DLL into the main application. Although not demonstrated here, the `TResourceLibrary` class releases the DLL handle when it is destroyed, and therefore freeing a `TResourceLibrary` instance releases the memory for the resources.

There is one further Delphi class that helps in manipulating resources, the `TResourceStream` class. This allows access to data stored in a resource in a stream-based manner. Streams are a far more powerful and flexible way of dealing with files (and data generally) than the traditional Pascal file handles, and are to be strongly recommended when compatibility with old Pascal code is not a consideration. Delphi uses `ResourceStreams` extensively when loading form data, but there is a particularly convenient use for it with respect to application deployment.

It may be the case that you wish to ship an application with a number of related data files which will be used as required at runtime, for example say a number of word processor files stored in RTF format, which will be printed using the `TRichEdit` control for nice formatting. Normally, these files would be stored in a known directory (typically with the application), and would be accessed as required. The disadvantages of this approach are that the installation procedure is more complicated (as more files are involved), the application must make extra checks to ensure that the files exist when they are required, and such files have a knack of somehow being 'uninstalled' by clients!

Instead, such files can be bound into the application or into a DLL, and accessed directly using the `TResourceStream` class. Of course, this does have the disadvantage that the application will increase in size (unless you use a DLL), but it does mean that you need only to distribute a single .EXE which you know will have all of the required data in it. This can considerably ease deployment issues, especially for distribution over the web.

A `TResourceStream` class is created using a constructor with typical resource related parameters with which we should be familiar: the Windows module handle (`HInstance` or the DLL handle), an identifier for the resource and a string that indicates the type of data held in the resource. For these instances we will use a data type called `RT_RCDATA`, which pretty much means anything we want it to, the decoding of the data stream is entirely left down to us. The `TResourceLibrary` component again provides a method that provides access to a resource stream in a convenient form, and the demo application uses this to load a text file containing the source code for the class into listbox. Note that the stream is recreated whenever the

➤ *Listing 5: The public interface of the TResourceLibrary class.*

```
TResourceLibrary = class
public
  constructor Create;
  constructor CreateForDLL (DLLFileName: String);
  destructor Destroy; override;
  // access to known datatypes
  procedure LoadIcon (ResourceID: Cardinal; Icon: TIcon);
  procedure LoadBitmap (ResourceID: Cardinal; Bitmap: TBitmap);
  // access to strings
  property Strings[ResourceID: Cardinal]: String; default;
end;
```

property is accessed, and so to use the property it should either be assigned to a variable, used in a `with` clause, or of course it can be used as a one-off for statements such as `TStrings.LoadFromStream`.

Conclusion

In this article we have explored some of the ways in which resource files can be used to enhance your applications for localisation, reduced memory consumption and easier deployment. Resources are such a vital part of Windows programming that being able to use them appropriately adds another technique to the application developer's arsenal, and is one that can provide very great benefits. If you are interested in examining the resources used by an application, the demo project ResXplor provided with Delphi allows you to browse both EXE and DLL files, and can provide insights into how they are composed.

Philip Brown is a senior consultant with Informatica Consultancy & Development, specialising in OO systems design and training. When not orienting objects he enjoys sampling fine wine. He can be contacted via email at phil@informatica.uk.com