Using Resource Files In Delphi

by Dave Bolt

For those of us who have previously used C and C++ to write our Windows programs, Delphi's use of resource files seems a little strange. The only real problem is that when a project is created in Delphi, a resource file is also created with the same name as the project. ***Delphi*** controls this file. Using the Image Editor to add bitmaps, cursors or icons to the Delphi-controlled resource file is doomed to failure under normal circumstances.

However, all is not lost. In this article I'll work through some examples of the use of bitmaps, icons and cursors from resources, then finish off by outlining a method by which the Delphi-controlled resource file can be manipulated without raising any objections from Delphi.

The Image Editor

A word of caution should be given about the Image Editor. Apart from my own programming errors, this is the only part of Delphi that has caused problems to date.

I would advise anyone using this tool to make sure that everything else which is open has been saved first. I would also advise that any work done in the Image Editor be saved regularly and that each open image and file be closed before closing the editor. This will not prevent problems totally, but will reduce the likelihood of work being lost.

Loading From .BMP Files

This first project loads a bitmap from a file which is not part of a resource file. In the project itself, the name of the bitmap is given explicitly. Unless a path is defined, Delphi assumes that the bitmap is in the same directory as the executable file. You can of course include the full path and file name for the bitmap in the code, but this would tie your application to a specific directory structure.

If we create a new project, rename UNIT1.PAS to BIT1.PAS and PROJECT1.DPR to BITPRJ1.DPR, then add a `TImage` component to the form, we have the basic framework to display an image. We can initialise the image component by adding code to the FormCreate handler as in Listing 1.

Compiling and running this project results in the bitmap being displayed. If the bitmap is not found when the program runs, an `EFOpenError` exception is generated. This can easily be detected and handled if required.

The object `MyBitmap` is our responsibility, and must be created before use then destroyed afterwards by our code. A separate copy of the bitmap is stored in the `TImage` component and is automatically allocated and de-allocated by the program.

Loading From Resource Files

To load a resource from a resource file, the file must have a different name to the project in which it is used. If we create a new project exactly as above, but call the files BIT2.PAS and BITPRJ2.DPR, we find that after saving the project there is a resource file called BITPRJ2.RES in the directory where the project was saved. In order to access our own resource file, it must be referenced explicitly in the project file (.DPR) using, eg:

```
{$R BITS.RES}
```

for a file called BITS.RES (which is on this issue's disk along with the other files from this article, and contains a bitmap resource called `FIRST`). You might think that the `{$R *.RES}` statement included by default in the .DPR file will pull in your own .RES files, but this is strangely not the case! The best place to insert your `{$R ...}` statement is *after* the existing one, after the `uses` statement.

If we add the same code to the `FormCreate` method as in Listing 1, but change the `LoadFromFile` command to:

```
MyBitmap.Handle :=
  LoadBitmap(hInstance,
  'FIRST');
```

then tidy up the display by adding the following two lines:

```
Image1.Width :=
  MyBitmap.Width;
Image1.Height :=
  MyBitmap.Height;
```

then run the program, we should get the same image displayed as before. The extra lines of code adjust the size of the `TImage` component to fit the bitmap at run time.

Caution

If the bitmap which is to be loaded is not in the resource file, the program will still run, it just won't display the bitmap. This situation can be avoided by testing to see if the handle is zero after a call to `LoadBitmap` and taking the appropriate action. Failure can also occur if there is insufficient memory to load the bitmap.

➤ *Listing 1*

```
procedure TForm1.FormCreate(Sender : TObject);
var
  MyBitmap : TBitmap;
begin
  MyBitmap := TBitmap.Create;
  MyBitmap.LoadFromFile('FIRST.BMP');
  Image1.Canvas.Draw(0, 0, MyBitmap);
  MyBitmap.Destroy;
end;
```

Custom Cursors

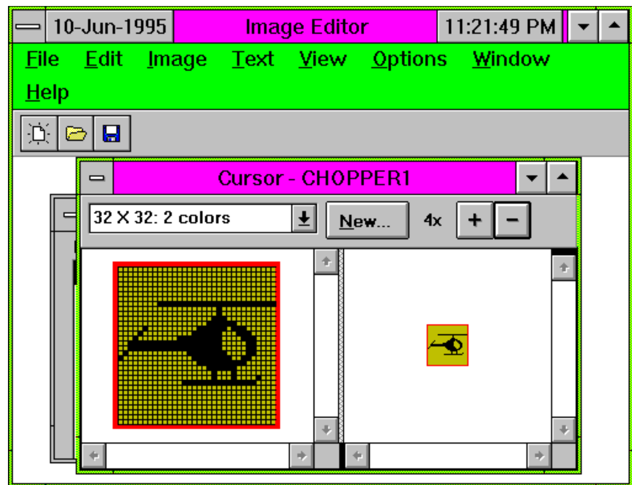
by Ken Otto

Creating custom cursors in Delphi can be a confusing venture. Not because it is hard, but because it is so poorly illustrated (and even wrong) in the Delphi documentation and help files.

A cursor actually contains two 32 x 32 monochrome bitmaps. One of these bitmaps is referred to as the 'XOR' bitmap, and the other is known as the 'AND' bitmap. When you create a cursor with Delphi's Image Editor, you won't need to worry about this. The cursor also has two fields defining the 'hot-spot': the point on the cursor representing its exact location.

There are 17 pre-defined cursor constants for your Delphi application. Some of the constant values in the Delphi Help file are incorrect. A corrected listing of cursors and their constants is shown opposite.

To create your own cursor, first create a resource (.RES) file. From the Delphi IDE select `Tools|Image Editor` then select `File|New`. By default, the 'Resource File (RES)' radio button will be selected. Choose `OK`. A window appears, captioned 'Untitled1.RES'. Click `New`, and a window will pop up asking for the type of resource you want to create. Choose `Cursor` and click `OK`. You may need to maximize the window at this point. Select a tool and begin drawing your cursor. If you make a mistake, select `Edit|Undo` to erase the last image written (this will continue to remove several layers each time it is selected). You can enlarge the drawing area by clicking the `Zoom` button. If you prefer a grid on the drawing area, select `Options|Show Grid On Zoom`. If you want the hot-spot to be the center of the cursor, select `Image|Hotspot...` and place 16 in the X and Y fields.



When you have finished drawing your cursor, you will probably want to rename it from the default name `CURSOR_1`: close the cursor editor window, ensure the Cursors tab on the untitled project window is selected, select `CURSOR_1` and click the `Rename` button. Make sure the new name is all capital letters. Now save the .RES file by selecting `File|Save As`, taking care not to give the file the same name as your project. A sample is included in the CHOPDEMO.LZH archive on the disk.

Ken Otto writes Pascal applications on the HP3000 in Sacramento, CA, USA; programming in Delphi is a hobby he enjoys. He can be reached on CompuServe at 73041,1336

Resources By Number

by Brian Long

When you name a resource in a resource file, Windows lets you choose numbers instead of names. In fact Microsoft recommends you use numbers instead of names for efficiency. However, the Image Editor will only store character strings as resource identifiers.

To generate a resource file for a cursor that marks resources by number, make a cursor resource file (.CUR file) with the Image Editor. One is supplied on the disk in the NUMCURS.LZH archive as file TARGET.CUR. A text file then acts as a resource script (a file with a .RC extension), and can look like this file, CURSOR2.RC (see my Typecasting Part 3 article in this issue for details of how to share constants between the resource script and the Delphi project):

```
2 CURSOR TARGET.CUR
```

This can be compiled with the command-line resource compiler BRCC.EXE (found in the directory DELPHI\BIN) with the command: `BRCC CURSOR2.RC`

The `{$R CURSOR2.RES}` compiler directive will bind in the resulting CURSOR2.RES and `LoadCursor` can also be used to load up a numbered, as opposed to named, resource. The last parameter to `LoadCursor` needs to be a `PChar` type, but we wish to specify a number. The parameter can be either `PChar(2)`, `'#2'` or `MakeIntResource(2)`. So, the `OnCreate` handler will look like:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crTarget] :=
    LoadCursor(HInstance, PChar(2)
  Form1.Cursor := crTarget;
end;
```

The complete example project in NUMCURS.LZH on the disk uses a 'named' cursor for the form and a different 'numbered' cursor for a button on the form.

Brian Long... well, by now surely he needs no introduction!

Cursor	Value
crDefault	0
crNone	-1
crArrow	-2
crCross	-3
crIBeam	-4
crSize	-5
crSizeNESW	-6
crSizeNS	-7
crSizeNWSE	-8
crSizeWE	-9
crUpArrow	-10
crHourGlass	-11
crDrag	-12
crNoDrop	-13
crHSplit	-14
crVSplit	-15
crMultiDrag	-16

➤ *Standard Cursors in Delphi*

A Simple Animation

This animation is so simple that it only has two frames. The program cycles through the images to hopefully give the impression of movement. This is the kind of thing which makes buttons appear to be pressed in or out in Windows programs and to animate logos.

If we take the project in the previous example and save it using the names BIT3.PAS and BITPRJ3.DPR, we have the basis for the next step. We need to add a timer to the form and the following variable declarations in the public part of the BIT3.PAS unit:

```
MyBitmap :
  array [0..1] of TBitmap;
BitMapNum : integer;
```

We also need `FormCreate`, `Timer1` and `FormDestroy` handlers as in Listing 2.

This project will now load two images from a resource file into an array of `TBitmap` objects and alternately display them in the image component. The image displayed is updated every fifth of a second in response to the timer. Note that the `TBitmap` array is declared as part of the `TForm1` object, instead of being local to the `FormCreate` handler, and must be initialised in `FormCreate` and destroyed in `FormDestroy`.

```
procedure TForm1.FormCreate(Sender : TObject);
begin
  MyBitmap[0] := TBitmap.Create;
  MyBitmap[1] := TBitmap.Create;
  MyBitmap[0].Handle := LoadBitmap(hInstance, 'FIRST');
  MyBitmap[1].Handle := LoadBitmap(hInstance, 'SECOND');
  Image1.Width := MyBitmap[0].Width;
  Image1.Height := MyBitmap[0].Height;
  { Show the image component copy of the bitmap }
  Image1.Canvas.Draw(0, 0, MyBitmap[0]);
  BitMapNum := 0;
  Timer1.Interval := 200;
end;
procedure TForm1.Timer1Timer(Sender : TObject);
begin
  BitMapNum := (BitMapNum+1) MOD 2;
  Image1.Canvas.Draw(0, 0, MyBitmap[BitMapNum]);
end;
procedure TForm1.FormDestroy(Sender : TObject);
begin
  MyBitmap[0].Destroy;
  MyBitmap[1].Destroy;
end;
```

➤ *Listing 2*

```
procedure TForm1.FormCreate(Sender : TObject);
begin
  Screen.Cursors[1]:=LoadCursor(HInstance,'ONE');
  Cursor:=1; { You could also reference your cursor as a constant:       }
end;        { "Cursor := crMyCursor;" just by including the statement     }
            { "const crMyCursor = 1;" before "implementation" in the unit }
```

➤ *Listing 3*

We have now loaded bitmaps from a bitmap file and from resource files. The loaded data has been copied to a `TImage` component either once only (at form creation), or repeatedly in response to an event. Similar things can be done with cursors.

Loading A Cursor

As they say in all the best recipes, first create your cursor (see opposite). Then, as for the bitmaps, create a new project, name the files CUR1.PAS and CURPRJ1.DPR, and add the `FormCreate` handler which is shown in Listing 3. Also, in the CUR1.PAS unit file, add a `{$R}` statement after the `interface` keyword: `{$R CURSORS.RES}`.

Although Delphi will permit you to insert this statement in a number of places in the unit, this seems to be the place to put it to get it to work. The `{$R}` statement can also go after the default `{$R}` statement in the .DPR file, as for the bitmap projects. If the unit is to be used in another project, this could lead to problems remembering to include the correct file so I prefer to add it to the relevant unit.

`Screen.Cursors[]` is an array of cursors supplied by Delphi. The default cursors use index numbers from 0 for the default cursor to -17 for `crSQLWait`. Unless we wish to replace any of the defaults, the best strategy is to use cursor numbers starting from 1 and working up.

Running the program should give a strange cursor consisting of an angle and a ring containing a black quadrant. If an attempt is made to load a cursor resource, but there is nothing matching that name in the resource files, the handle will be zero. If a resource other than a cursor is found, the handle will not be zero, so take care to only reference *cursors* in `LoadCursor`.

An Animated Cursor

We can create a new project by saving the previous one as CUR2.PAS and CURPRJ2.DPR. For the animation we need a `TTimer` component on the form and this time also a `TButton`. Caption the button 'Finish' so that there is some point to including it, and it helps if the size is a little larger than normal. Also set the `Cursor` property of the button to `crCross`.

In the `Public` part of the `type TForm1` declaration include:

```
CursorCount : Integer;
```

The handlers in Listing 4 are also required.

`CursorCount` is used to keep track of the next cursor to load. Having it zero-based simplifies the arithmetic. The first user-defined cursor is 1 and the last is 4, so we add 1 to `CursorCount` to give the cursor number to use. When this program is run the cursor will have a rotating quadrant as part of it. The effect can be improved considerably by drawing eight versions with the quadrant moved 45 degrees from the last position.

Compare the behaviour of the cursor with the previous version. As the cursor moves across the non-client area of the form, it starts flashing between the custom cursor and the relevant default cursor. This undesirable behaviour is brought on by the slightly simplistic method of changing the cursor in the timer handler. Note also that when the cursor is over the 'Finish' button it changes to the shape set at design time. Try this with the `Button1.Enabled` property set to `False`.

One method of modifying the cursor behaviour further would be to use `GetCursorPos` to find the position in terms of global co-ordinates, then convert to client area co-ordinates and check if the cursor is actually in a valid region or not.

Another method, which is demonstrated in CURPRJ2A.PRJ, uses one of the less known windows messages `WM_NCMOUSEMOVE`, which is generated in response to mouse movement in the non-client area of a window. This just happens to be perfect for controlling the flashing effect in CURPRJ2.PRJ.

We need to amend the `private` and `public` declarations of the `type` section in the unit file (now CUR2A.PAS) as in Listing 5.

The `WMNCMouseMove` procedure is a message handler for the required message. It overrides the default message handler, but instead of the `override` keyword it uses the `message` keyword. (For more information use Search All in the Help system, and look up 'message'). This method of overriding windows message handlers can be used for any message and also to handle additional messages generated by our own programs.

The message handling function is given in Listing 6, and goes in the implementation section of the CUR2A.PAS unit. The timer and mouse movement handlers also need to be amended.

Note that the `WMNCMouseMove` handler will only be called if Windows updates the mouse position while the mouse cursor happens to be on a non-client area of the form.

```
procedure TForm1.FormCreate(Sender : TObject);
begin
  Screen.Cursors[1] := LoadCursor(HInstance,'ONE');
  Screen.Cursors[2] := LoadCursor(HInstance,'TWO');
  Screen.Cursors[3] := LoadCursor(HInstance,'THREE');
  Screen.Cursors[4] := LoadCursor(HInstance,'FOUR');
  Cursor := 1;
  CursorCount := 0;
  Timer1.Interval := 100;
end;
procedure TForm1.Timer1Timer(Sender : TObject);
begin
  CursorCount := (CursorCount+1) MOD 4;
  Cursor := CursorCount+1;
end;
procedure TForm1.Button1Click(Sender : TObject);
begin
  Close;
end;
```

➤ *Listing 4*

```
private
  procedure WMNCMouseMove(var AMessage : TMessage); message WM_NCMouseMove;
public
  CursorCount : integer;
  DeadArea : BOOL;   { True if NonClient Area }
end;
```

➤ *Listing 5*

```
procedure TForm1.WMNCMouseMove(var AMessage : TMessage);
begin
  DeadArea := TRUE; {Cursor is on the form but in an invalid area}
  AMessage.Result := 0; { Win API Help says return 0 if message was handled}
  inherited;
end;
procedure TForm1.Timer1Timer(Sender : TObject);
begin
  { Calculate the next cursor }
  CursorCount:= (CursorCount+1) MOD 4;
  { If in a valid region, Update cursor }
  if not DeadArea then Cursor := CursorCount+1;
end;
procedure TForm1.FormMouseMove(Sender : TObject; Shift : TShiftState;
  X,Y : Integer);
begin
  DeadArea := FALSE; { If this handler is called, DeadArea must be false }
  Cursor := CursorCount+1; { Update Cursor }
end;
```

➤ *Listing 6*

Icons

Looking at Delphi's `Options|Project|Application` settings we find that Delphi supplies a default icon. This can be replaced quite simply by clicking the 'Load Icon' button and browsing around until we find a suitable icon. After compiling the project and running it, it will quite nicely change itself into the new icon as requested. Further, if we use Image Editor to open the resource file that Delphi maintains for the project and edit `MAINICON`,

we find that the resource file actually contains the requested icon. This is the only situation where I have found Delphi giving the user any control over the default resource file for a project.

If we create a new project, we can note that the form properties in the Object Inspector list an Icon property, which is blank by default. Double-clicking on the '...' for that field brings up Picture Editor, which can be used to load another icon, in addition to the default one already discussed. This means that in a multi-form project we can have a different icon for each form.

As with the bitmaps, we can load icons from files using the method `LoadFromFile` and from resource files using the `LoadIcon` function from the Windows API. Listing 7 shows a simple example of loading from a resource file, the project is ICOPRJ1.DPR. Once again, I have included an animation. After the previous examples, this project should be clear enough. The animation is of course only visible when the project is minimised. The 'Shrink Me' button in the middle of the form minimizes the program.

If an icon name is supplied in `LoadIcon` but the icon is not found, the handle returned will be `NUL`.

```
unit Ico1;
interface
{$R ICONS.RES}
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls;
type
  TForm1 = class(TForm)
  Timer1 : TTimer;
  Button1 : TButton;
  procedure FormCreate(Sender : TObject);
  procedure Timer1Timer(Sender : TObject);
  procedure FormDestroy(Sender : TObject);
  procedure Button1Click(Sender : TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    IconNumber:integer;
    Ico : array [0..1] of TIcon;
  end;
var  Form1 : TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender : TObject);
begin
  Ico[0] := TIcon.Create;
  Ico[1] := TIcon.Create;
  Ico[0].Handle := LoadIcon(HInstance, 'Icon_1');
  Ico[1].Handle := LoadIcon(HInstance, 'Icon_2');
  Icon := Ico[0];
  IconNumber := 0;
  Timer1.Interval := 200;
end;
procedure TForm1.FormDestroy(Sender : TObject);
begin
  Ico[0].Destroy;
  Ico[1].Destroy;
end;
procedure TForm1.Timer1Timer(Sender : TObject);
begin
  IconNumber := (IconNumber+1) MOD 2;
  Icon := Ico[IconNumber];
end;
procedure TForm1.Button1Click(Sender : TObject);
begin
  Application.Minimize;
end;
end.
```

➤ *Listing 7*

Side-Stepping Delphi

At the start of this article I mentioned that Delphi maintains a resource file with the same name as each project and that attempting to manipulate the contents is generally a waste of time.

However, if you wish to alter the resources available in the default file, this can be done as long as the associated project is not active within Delphi, ie if you want to alter CURPRJ1.RES, either close the CURPRJ1 project, or open another project. Delphi will then be quite happy for you to alter the contents of the default .RES file.

A direct result of this behaviour is that it is possible to alter a project resource file without intending to, simply by forgetting that a particular resource file is the default for a project that has not been worked on for some time and adding or removing resources. If a project suddenly starts failing to compile because of duplicate resource identifiers, check all the resources included in case this has happened. The project must be re-compiled in order to make use of revised resources.

Personally, I feel that there is very little advantage in altering the default resource file. Firstly, there is too much to do, compared with just adding another file reference into either the project (.DPR) or a unit (.PAS) file. Secondly, it makes life difficult if you wish to copy a unit into another project, since you then have to manipulate the default resource file for that project as well. Thirdly, the work-around is not something that Borland have recommended. If you contact them, they specifically tell you not to try to modify the default resource file.

My real reason for including the technique here is that if like me you have accidentally replaced a default resource file in a project, you are now in a position to sort it out without deleting the whole project and starting again.

One final comment. The names of resources in the .RES file can in theory be in either upper or lower case. The feedback I have had from various people is that upper case throughout is needed to ensure that everything works correctly.

Dave Bolt hails from Barnsley in Yorkshire and can be contacted on CompuServe as 100112,522
©Copyright 1995 D M Bolt