Beating the System: Manipulating 16-bit Resources

by Dave Jewell

Although Microsoft would no doubt like to persuade us that the days of 16-bit applications are well and truly over, all the evidence suggests that 16-bit software development is still very much alive and kicking. Just this last month, I've received no less than five requests for help with understanding the 16-bit Windows executable file format, and particularly how to extract resource information from existing files. OK, I can take a hint!

Why would you want to access the resource information in an existing file? There are many reasons for doing this. Suppose you're building an icon editor and you want to give users the ability to find and extract icons contained within Windows DLLs and other executable files. Or maybe you want to build some utility which allows you to examine the various forms and controls used by a particular Delphi executable? This is easy once you know how to track down the resource information inside a file.

If you're familiar with the Windows API, you'll know that it contains a number of resource related routines such as `ExtractIcon`. This API call can be used to get an icon handle to a particular icon within an executable file. Well, that's terrific if it's icons that you're after, but what if you want a dialog, a bitmap, or a Delphi form? Rather than needlessly duplicate the functionality of existing API routines, the code in this article will show you how to extract any type of resource from an existing NE format file.

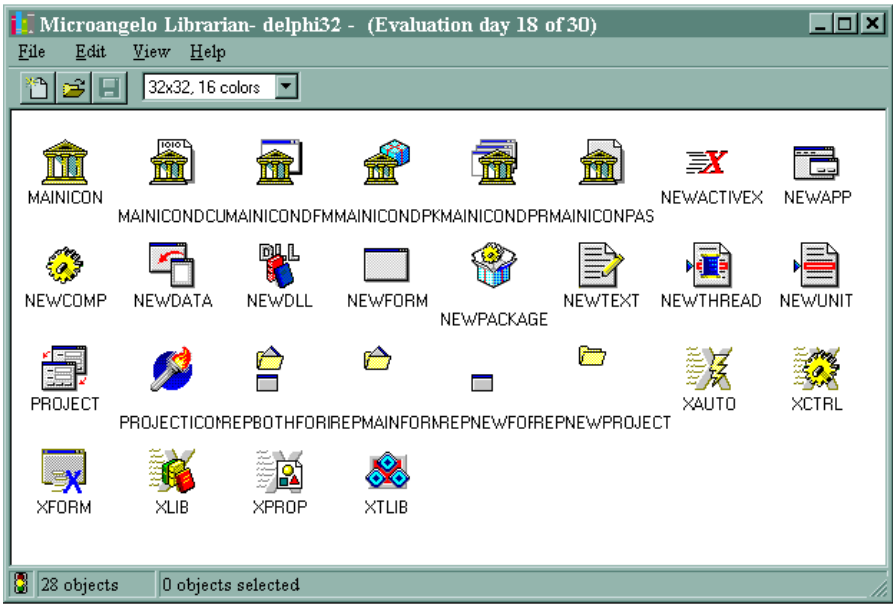
The Windows NE Executable File Format

NE (New-Executable) files use the 16-bit file format which Microsoft first created for real mode Windows. I won't waste time covering irrelevant details of this file format, we'll just concentrate on the specific information which you need to track down resource information within the file. The first word of any executable file (whether it be a real-mode DOS application, a 16-bit application or a 32-bit application) will always be set to the value `$5A4D` (ascii `MZ`). This is the magic signature which DOS uses to identify an executable file. If the signature isn't there, the file is rejected as invalid.

The magic signature is followed by a real-mode DOS header containing a large number of additional fields, none of which are relevant to us as they relate to the real-mode stub associated with the file. You need to realise that both 16-bit and 32-bit executables have a small real-mode stub containing a DOS program that's entirely separate from the code in the Windows part of the file. Microsoft did things this way because in earlier versions of DOS, running a Windows program from the DOS command line resulted in the error message *This program needs Microsoft Windows* or similar. The error message wasn't generated by DOS (which knew nothing about Windows when Windows wasn't running!) but by the real-mode stub code itself. Nowadays Windows 95 is smart enough to detect the fact that you're running a Windows application and will automatically launch it just as if you'd double-clicked it from within the Explorer. Nevertheless, the DOS stub code is still there.

What is of interest is the 32-bit long word at offset `$3C` in the file. This field is used only by Windows executables: it contains a file-relative offset that points to another header contained within the same file. This header is the one that is of relevance to the Windows portion of the file. It's quite a complex data structure (very complex in the case of 32-bit files) and again, most of it isn't relevant to us. As with the DOS header, the first word of the Windows file header is used to

➤ *MicroAngelo is one of the best shareware programs for extracting icons from executables (http://www.impactsoft.com/muangelo/muangelo.html)*



identify the type of header we're dealing with: it contains the value $454E (ascii NE) in the case of 16-bit executables and $4550 (ascii PE) for 32-bit Portable Executables. Having found the DOS executable signature at the beginning of the file and $454E at the location pointed to by the file offset at location $3C, we can be pretty sure that we're dealing with a 16-bit Windows executable.

The next step is to determine if the file contains any resource data. Once the Windows file header has been found, the resource table itself must be tracked down. The 16-bit offset of the resource table is located at byte position $24 within the Windows file header. This part of the Windows header file is arranged as a series of 16-bit offsets which point to other consecutive parts of the file. Thus, the size of the resource table can be determined by comparing the offset of the resource table with the offset of the next part of the file. In my code, I look to see if the resource table is greater than four bytes in length. If it isn't, then I know it's empty. This is because I've found by experimentation that if you strip all the resources out of an executable file using Borland's Resource Workshop program, it will leave a vestigial 4-byte resource table in the file. If we assumed that any non-zero value for the resource table size indicated the presence of resources, then we might end up in deep trouble!

Resource Table Format

At this point, we've found the resource table and we know how big it is. This would be a good point to review the format of the resource table itself. As you'll no doubt appreciate, a resource is identified by two things: the resource type, and the resource ID. Microsoft have defined a number of standard resource types such as rt_Cursor, rt_Bitmap and so forth (you can find a list of these declarations in WINTYPES.PAS, or in WINDOWS.PAS if you're using a 32-bit version of Delphi). Delphi applications make extensive use of the rt_Data resource type which corresponds to an application-specific chunk of arbitrary data. This resource type is used to store a binary representation of all the Delphi form files used by a program.

The resource table itself is organised as a series of resource type identifiers, followed by a count and then all the resources of that type. For example, if a file contains five bitmaps and twelve forms, you might expect the file to start with the resource type identifier for bitmaps, followed by a count of five, and then the information relating to the five individual bitmap resources. This would then be followed by the resource type identifier for rt_Data resources, a count of twelve, and then the information relating to each of the twelve form resources in the file.

Just to make things a little more interesting, resources can be named either with a string, or with a number and the same string/number system can even be used when defining resource types. For example, you can name a resource as WIDGET or you can give it the number 10. Inside your application code, you can refer to a resource by name or by number. How does the resource table format cater for both eventualities? A name or number is specified by a single 16-bit word. If the high bit (bit 15) of the word is set high, then the other bits of the word specify a number. However, if the high bit isn't set, then the number is a file offset (relative to the start of the resource table) which points to a Pascal-style string.

The ResFile Unit

I could go on and describe the format of the resource table entries (the stuff that follows the resource count for each type of resource), but let's lay the theory aside for a while and look at the actual code, shown in Listing 1. Although there was some mileage to be gained by writing my Shell Link (see Issue 19) as a component, this isn't really the case here, so I've manfully resisted the temptation (!) and left ResFile as an ordinary Pascal unit.

I've encapsulated the mechanics of 'resource sniffing' into a single

➤ *Facing page: Listing 1*

object, TResFile. In order to access the resources in a particular file, you call the constructor for TResFile, passing it the name of the file you're interested in. You can then determine how many different resource types are contained in the file by reading the value of the ResTypeCount property. For each resource type in the file, you can index into the ResTypes property to obtain the name of that particular type. For each type of resource, you can call GetResourceCount to obtain the number of resources of that specific type while calling GetResourceName will give you the name of an individual resource. Finally, the GetResourceInfo routine will provide important details relating to a specific resource such as its size in bytes, the byte offset of the resource within the file and any flags associated with the resource. (See the Windows SDK documentation for a description of the different flags that relate to a resource).

Bear in mind that Windows doesn't store the exact size of individual resources. Instead, it uses a special 'shift count' (stored as part of the resource table) to calculate the reported resource size and position. For example, suppose that this shift count has a value of four. In this case, the value for the resource's size and position (as found in the resource table) will be shifted left four places before being returned. If the actual size of a resource is $1234 bytes, the stored value will be $124 and a value of $1240 will be returned. Thus, resource sizes will always be rounded up to the next highest multiple of 16 (assuming a shift of 4) and resource file positions will always be aligned on 16-byte file boundaries. You might wonder why Microsoft implemented such a bizarre scheme. The reason is simply that it enables a lot of (potentially large) resources to be represented in a compact manner.

One other point about the interface to TResFile: I've added a Boolean property called ResMapNames. This maps the standard

```
unit ResFile;
{ Implementation of TResFile. This version 16-bit (NE) only.
  Author: Dave Jewell, 1996-1997, ALL RIGHTS RESERVED. }
interface
uses WinTypes, WinProcs, Classes, SysUtils;
const
  { Magic numbers }
  DOS_Magic = $5A4D; { Magic word for old-style DOS EXEs }
  W16_Magic = $454E; { Magic word for new-style 16-bit EXEs }
  eFileNotFound    = 'File % not found';
  eFileNotExe      = 'File % is not an executable';
  eFileNotNE       =
    'File % is not a Windows 16-bit (NE) executable';
type
  EResFile = class (Exception);
  PResInfo = ^TResInfo;
  TResInfo =
    record
      ROffset: LongInt;    { Offset of resource data }
      RLength: Word;       { Length of resource data }
      RFlags: Word;        { Flags for this resource }
    end;
  TResFile = class (TObject)
  private
    fName: String;
    fMapNames: Boolean;
    fHeaderPos: LongInt;
    fTypesList: TStringList;
    procedure Panic(const Message: String);
    function MapResNumToString(const Name: String): String;
    function MapStringToResNum(const Name: String): String;
    function GetResList(const TypeName: String): TStringList;
    function GetTypeName (Index: Integer): String;
    function GetResourceTypeCount: Integer;
  public
    constructor Create (const FileName: String);
    destructor Destroy;
    property ResTypeCount: Integer read GetResourceTypeCount;
    property ResTypes[Index: Integer]: string
      read GetTypeName;
    property ResMapNames: Boolean
      read fMapNames write fMapNames;
    function GetResourceCount(const TypeName: String):
      Integer;
    function GetResourceName(const TypeName: String;
      Idx: Integer): String;
    procedure GetResourceInfo(const TypeName: String;
      Idx: Integer; var Info: TResInfo);
  end;
implementation
constructor TResFile.Create (const FileName: String);
var
  fs: TFileStream;
  ResShift, ResTablePos, ResTableSize: Word;
  function ReadByte: Byte;
  begin
    fs.Read (Result, sizeof (Result));
  end;
  function ReadWord: Word;
  begin
    fs.Read (Result, sizeof (Result));
  end;
  function ReadLong: LongInt;
  begin
    fs.Read (Result, sizeof (Result));
  end;
  function ReadString: String;
  var
    Idx, i: Word;
    OldPos: LongInt;
  begin
    Idx := ReadWord;
    if Idx = 0 then
      Result := ''
    else if (Idx and $8000) <> 0 then
      Result := Format ('#%d', [Idx and $7FFF])
    else begin
      OldPos := fs.Position;
      fs.Position := fHeaderPos+ResTablePos+Idx {-Ord(fType)};
      Result [0] := Char (ReadByte);
      for i := 1 to Ord (Result [0]) do
        Result [i] := Char (ReadByte);
      fs.Position := OldPos;
    end;
  end;
  function ReadResourceList: Boolean;
  var
    ResType: String;
    i, Count: Integer;
    Res: ^TResInfo;
    List: TStringList;
  begin
    Result := False;
    ResType := ReadString;
    if ResType <> '' then begin
      Result := True;
      List := TStringList.Create;
      { Count number of resources of this type }
      Count := ReadWord;  ReadLong;
      for i := 0 to Count - 1 do begin
        GetMem (Res, sizeof (TResInfo));
        Res^.ROffset := LongInt (ReadWord) shl ResShift;
        Res^.RLength := ReadWord shl ResShift;
        Res^.RFlags := ReadWord;
        List.AddObject (ReadString, TObject (Res));
        ReadLong;
      end;
      fTypesList.AddObject (ResType, List);
    end;
  end;
  procedure ReadResources;
  var ResType: Word;
  begin
    with fs do begin
      { Get the size and position of the resource table }
      Position := fHeaderPos + $24; ResTablePos := ReadWord;
      ResTableSize := ReadWord - ResTablePos;
      { Stripping all resources with RW leaves 4-byte table }
      if ResTableSize > 4 then begin
        Position := fHeaderPos + ResTablePos;
        ResShift := ReadWord;
        while ReadResourceList do ;
      end;
    end;
  end;
begin
  fName := FileName;
  fMapNames := False;
  fTypesList := TStringList.Create;
  if not FileExists (FileName) then
    Panic (eFileNotFound);
  fs := TFileStream.Create (FileName, fmOpenRead);
  with fs do
    try
      if ReadWord <> DOS_Magic then
        Panic (eFileNotExe);
      Position := $3C;
      Position := ReadLong;
      fHeaderPos := Position;
      if ReadWord <> W16_Magic then
        Panic (eFileNotNE);
      { We know it's NE executable, load what we're after }
      ReadResources;
    finally
      fs.Free;
    end;
end;
destructor TResFile.Destroy;
var
  j: Integer;
  TypeList: TStringList;
begin
  while fTypesList.Count > 0 do begin
    TypeList := TStringList (fTypesList.Objects [0]);
    for j := 0 to TypeList.Count - 1 do
      FreeMem (TypeList.Objects [j], sizeof (TResInfo));
    TypeList.Free;
    fTypesList.Delete (0);
  end;
  fTypesList.Free;
end;
procedure TResFile.Panic (const Message: String);
var
  p: Integer;
  Str: String;
begin
  p := Pos ('%', Message);
  if p = 0 then
    Str := Message
  else
    Str := Copy(Message, 1, p - 1) + '"' + fName + '"' +
      Copy(Message, p + 1, 255);
  raise EResFile.Create (Str);
end;
function TResFile.GetResourceTypeCount: Integer;
begin
  Result := fTypesList.Count;
end;
function TResFile.MapResNumToString(
  const Name: String): String;
begin
  Result := Name;
  if (Result [1] = '#') and fMapNames then
    case StrToInt (Copy (Result, 2, 255)) of
      1:   Result := 'CURSOR';
      2:   Result := 'BITMAP';
      3:   Result := 'ICON';
      4:   Result := 'MENU';
      5:   Result := 'DIALOG';
      6:   Result := 'STRINGTABLE';
      7:   Result := 'FONTDIR';
      8:   Result := 'FONT';
      9:   Result := 'ACCELERATOR';
      10:  Result := 'RCDATA';
      12:  Result := 'GROUPCURSOR';
      14:  Result := 'GROUPICON';
      16:  Result := 'VERSIONINFO';
    end;
end;
```

{ ** CONTINUED ON NEXT PAGE —>> }

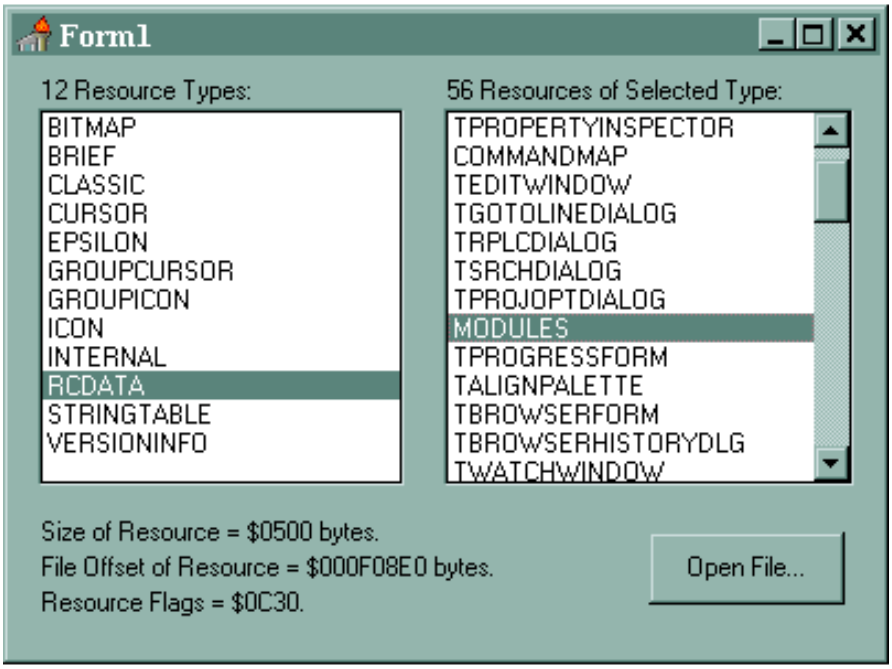
resource type names (and only the standard names) into a human readable form. Thus, if you have `ResMapNames` set to `True`, the `BITMAP` type will be returned as the string `BITMAP`. If the name mapping property is `False` then you'll just see it identified as `#2`.

Most of the work is done in the constructor for `TResFile`. Having verified that we're dealing with a 16-bit Windows executable (see previous explanation) and that the file contains resources, the `ReadResourceList` routine is called once for each encountered resource type. When it reaches the end of the resource list, this function returns `False`, otherwise it returns `True`. Assuming another set of resources are found, the code creates a new `TStringList` object and loops for as many times as are indicated by the resource count word, reading successive resource descriptions into memory, storing them in a small data structure of type `TResInfo`, and adding them to the string list.

You'll see from the code that each resource description consists of a word describing the file offset which has to be shifted left as described earlier. This is followed by a word giving the length of the resource which has to be shifted left in the same manner. Next comes a word containing resource flags and finally, a 16-bit word which is used to specify the number or name of the resource. You'll also see from the code that I then read and discard a 32-bit long value. This isn't used inside the file but is reserved for operating system usage (the Windows kernel loads the 12-byte data structure relating to each resource and uses this last field as a memory handle pointing to the in-memory resource data).

Right, let's see if it works as advertised. The screenshot shows a small program I developed that uses the `ResFile` unit. The source code is given in Listing 2. It allows

➤ *Here's my demo program peeking at the resources contained in the Delphi 1 IDE: as a general rule, if a Delphi application has an RCDATA resource whose name begins with a T then it's almost certainly a form resource*



➤ *Listing 1 (continued)*

```
{ ** CONTINUED FROM PREVIOUS PAGE }
function TResFile.MapStringToResNum(
  const Name: String): String;
var Num: Integer;
begin
  Num := -1;
  if (Name [1] <> '#') and fMapNames then begin
    if Name = 'CURSOR' then Num := 1;
    if Name = 'BITMAP' then Num := 2;
    if Name = 'ICON' then Num := 3;
    if Name = 'MENU' then Num := 4;
    if Name = 'DIALOG' then Num := 5;
    if Name = 'STRINGTABLE' then Num := 6;
    if Name = 'FONTDIR' then Num := 7;
    if Name = 'FONT' then Num := 8;
    if Name = 'ACCELERATOR' then Num := 9;
    if Name = 'RCDATA' then Num := 10;
    if Name = 'GROUPCURSOR' then Num := 12;
    if Name = 'GROUPICON' then Num := 14;
    if Name = 'VERSIONINFO' then Num := 16;
  end;
  if Num = -1 then
    Result := Name
  else
    Result := '#' + IntToStr (Num);
end;
function TResFile.GetTypeName (Index: Integer): String;
begin
  Result := '';
  if (Index >= 0) and (Index < fTypesList.Count) then
    Result := MapResNumToString (fTypesList.Strings [Index]);
end;
function TResFile.GetResList(
  const TypeName: String): TStringList;
var Idx: Integer;
begin
  Idx := fTypesList.IndexOf (MapStringToResNum (TypeName));
  if Idx = -1 then
    Result := Nil
  else
    Result := fTypesList.Objects [Idx] as TStringList;
end;
function TResFile.GetResourceCount(
  const TypeName: String): Integer;
var List: TStringList;
begin
  List := GetResList (TypeName);
  if List = Nil then
    Result := 0
  else
    Result := List.Count;
end;
function TResFile.GetResourceName(
  const TypeName: String; Idx: Integer): String;
var List: TStringList;
begin
  Result := '';
  List := GetResList (TypeName);
  if (List <> Nil) and (Idx >= 0) and (Idx < List.Count) then
    Result := List.Strings [Idx]
end;
procedure TResFile.GetResourceInfo(
  const TypeName: String; Idx: Integer; var Info: TResInfo);
var
  pInfo: PResInfo;
  List: TStringList;
begin
  List := GetResList (TypeName);
  if (List <> Nil) and (Idx >= 0) and (Idx < List.Count) then
    Info := PResInfo (List.Objects [Idx])^;
end;
end.
```

you to select a 16-bit Windows executable and then displays a list of all the available resource types and names in the two list boxes.

In the screenshot you can see some of the RCDATA resources inside the Delphi 1 executable. No prizes for guessing that these are the internal form names used by the Delphi IDE! Once you've selected a resource of interest, you can double click the resource name in the right-hand list box and the program will ask if you want to extract the selected resource. If you answer yes, then it will create a new binary file containing the data for that resource.

If, like me, you enjoy poking around in Delphi's innards, then you might be especially interested in the MODULES resource inside Delphi 1. Although this is an RCDATA resource, it's not a form. If you extract it, rename it as a text file and then open it, you'll find that it's actually a set of templates used by Delphi when creating units and DPR files. Ever wondered where all those skeletal class definitions come from? Look no further!

Here's one thing that might disappoint you: if you extract an icon, cursor or bitmap resource and then rename it to have the appropriate extension, you'll find that it isn't recognised by ordinary resource editors. That's because .ICO, .CUR and .BMP files have an additional 'file wrapper' that allows them to be recognised by utilities such as MicroAngelo. Next month, I'll explain how to extract resources into these industry standard file formats and extend our ResFile unit for 32-bit files too.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of *Instant Delphi Programming* published by Wrox Press. You can contact Dave as DaveJewell@msn.com, DSJewell@aol.com or as DaveJewell@compuserve.com

➤ *Listing 2*

```
unit Mainform;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, ResFile, StdCtrls;
type
  TForm1 = class(TForm)
    Types: TListBox;
    TypeCount: TLabel;
    ResList: TListBox;
    ResCount: TLabel;
    ResSize: TLabel;
    ResOffset: TLabel;
    ResFlags: TLabel;
    Button1: TButton;
    OpenDialog: TOpenDialog;
    procedure FormClose(
      Sender: TObject; var Action: TCloseAction);
    procedure TypesClick(Sender: TObject);
    procedure ResListClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure ResListDblClick(Sender: TObject);
  private
    rf: TResFile;
  public
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormClose(
  Sender: TObject; var Action: TCloseAction);
begin
  rf.Free;
end;
procedure TForm1.TypesClick(Sender: TObject);
var
  Count, Idx: Integer;
  ResType: String;
begin
  if (rf <> Nil) and (Types.Items.Count > 0) then begin
    ResList.Clear;
    ResType := Types.Items [Types.ItemIndex];
    Count := rf.GetResourceCount (ResType);
    ResCount.Caption :=
      IntToStr(Count) + ' Resources of Selected Type:';
    for Idx := 0 to Count - 1 do
      ResList.Items.Add (rf.GetResourceName (ResType, Idx));
    ResList.ItemIndex := 0;
    ResListClick (Self);
  end;
end;
procedure TForm1.ResListClick(Sender: TObject);
var Info: TResInfo;
begin
  if rf <> Nil then begin
    rf.GetResourceInfo(Types.Items[Types.ItemIndex],
      ResList.ItemIndex, Info);
    ResSize.Caption := 'Size of Resource = $' +
      IntToHex(Info.RLength, 4) + ' bytes.';
    ResOffset.Caption := 'File Offset of Resource = $' +
      IntToHex(Info.ROffset, 8) + ' bytes.';
    ResFlags.Caption := 'Resource Flags = $' +
      IntToHex(Info.RFlags, 4) + '.';
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
var Count, Idx: Integer;
begin
  ResList.Clear;
  Types.Clear;
  if rf <> Nil then rf.Free;
  if OpenDialog.Execute then begin
    try
      rf := TResFile.Create (OpenDialog.FileName);
      rf.ResMapNames := True;
      Count := rf.ResTypeCount;
      TypeCount.Caption := IntToStr(Count) +
        ' Resource Types:';
      for Idx := 0 to Count - 1 do
        Types.Items.Add (rf.ResTypes [Idx]);
      Types.ItemIndex := 0;
      TypesClick (Self);
    except
      Application.HandleException (Self);
    end;
  end;
end;
procedure TForm1.ResListDblClick(Sender: TObject);
var
  Info: TResInfo;
  ResData: Pointer;
  ResIndex: Integer;
  fs: TFileStream;
  TypName, ResName, fName: String;
begin
  with rf do begin
    ResIndex := ResList.ItemIndex;
    TypName := Types.Items [Types.ItemIndex];
    ResName := ResList.Items [ResIndex];
    if MessageDlg(Format(
      'Extract resource ''%s'' (type = ''%s'') ?', [ResName,
      TypName]), mtConfirmation, [mbYes, mbNo], 0) = idYes
      then begin
      GetResourceInfo (TypName, ResIndex, Info);
      GetMem (ResData, Info.rLength);
      try
        fs := TFileStream.Create(OpenDialog.FileName,
          fmOpenRead);
        try
          fs.Position := Info.rOffset;
          fs.Read (ResData^, Info.rLength);
        finally
          fs.Free;
        end;
        fName := ExtractFilePath(Application.ExeName) +
          ResName + '.BIN';
        fs := TFileStream.Create (fName, fmCreate);
        try
          fs.Write (ResData^, Info.rLength);
          MessageDlg('Resource has been written to ' +
            fName, mtInformation, [mbok], 0);
        finally
          fs.Free;
        end;
      finally
        FreeMem(ResData, Info.rLength);
      end;
    end;
  end;
end;
end.
```