Tips & Tricks

Bit Manipulation

Here is my solution for an enhanced bit manipulation class (see BITMANIP.PAS on the disk). There are some public domain solutions, which allow you to set or clear bits, but none I've seen allows you to insert or remove a bit position.

In CLASSES.PAS we find the undocumented `TBits` class, which can do most of the job and handle the necessary bit array size completely dynamically. It contains a lot of assembler stuff for good performance. So, I decided to take this AS my ancestor class.

Now we find the well-known problem that we need to inherit something that is defined in a `private` section of a class; in this case we need access to the pointer to the array, where all bits are maintained (`FBits`). The idea was to construct a fake class which makes something `public` that was `private` before:

```
type
  TFakeBits = class
  public
    FSize : Integer;
    FBits : Pointer;
  end;
```

In CLASSES.PAS the `TBits` class starts with the same declarations, the `public` is `private`. Now my enhanced bit manipulation class can be defined:

```
type
  tEnhancedBits = class(tBits)
    ffk : tFakeBits;
    constructor Create;
    procedure RemoveBit(idx:integer);
    procedure InsertBit(idx:integer; b:boolean);
  end;
```

The constructor is:

```
constructor tEnhancedBits.Create;
begin
 inherited Create;
 ffk:=tFakeBits(self);   { <<—  }
end;
```

So every time I need access to the bit array I can write `ffk.FBits`.

There is no additional secret in the `BitManip` unit in Listing 1. Perhaps someone has better algorithms in the `RemoveBit` and `InsertBit` methods (with a lot of `SHR`s and

```
unit BitManip;
{ unit for bit manipulation }
interface
uses
  Classes;
type
  TFakeBits = class
  public
    FSize: Integer;
    FBits: Pointer;
  end;
  TEnhancedBits = class(tBits)
    ffk : tFakeBits;
    constructor Create;
    procedure RemoveBit(idx:integer);
    procedure InsertBit(idx:integer; b:boolean);
  end;
implementation
uses
  Consts;
const
  BitsPerInt = SizeOf(Integer) * 8;

constructor tEnhancedBits.Create;
begin
 inherited Create;
 ffk := tFakeBits(self);
end;

procedure tEnhancedBits.RemoveBit(idx:integer);
var
  p,j,i : integer;
  tmp : tEnhancedBits;
begin
  { check index }
  if (idx<0) or (idx>=size) then
    raise EBitsError.CreateRes(SBitsIndexError);
  { create temporary class }
  tmp := tEnhancedBits.create;
  { set size of temporary class }
  tmp.size := size-1;
  { load temporary class }
  j := 0;
  for i := 0 to size-1 do
    if (i<>idx) then begin
      tmp[j] := bits[i];
      inc(j);
    end;
  { deallocate bit array of old class }
  Size := 0;
  { set new size }
  Size := tmp.Size;
  { compute size of allocated bit array }
  p := ((tmp.Size + BitsPerInt - 1) div
       BitsPerInt)*sizeof(integer);
  { move new array }
  move(tmp.ffk.FBits^,ffk.FBits^,p);
  { free temporary class }
  tmp.free;
end;

procedure tEnhancedBits.InsertBit(idx:integer;b:boolean);
var
  p,j,i : integer;
  tmp : tEnhancedBits;
begin
  if (idx<0) or (idx>size) then
    raise EBitsError.CreateRes(SBitsIndexError);
  { create temporary class }
  tmp := tEnhancedBits.create;
  {set size of temporary class }
  tmp.size := size+1;
  { load temporary class }
  j:=0;
  for i := 0 to size-1 do begin
    if i=idx then begin
      tmp[j] := b;
      inc(j);
    end;
    tmp[j] := bits[i];
    inc(j);
  end;
  if idx=size then tmp[j] := b;
  { deallocate bit array of old class }
  Size := 0;
  { set new size }
  Size := tmp.Size;
  { compute size of allocated bit array }
   p := ((tmp.Size + BitsPerInt - 1) div
        BitsPerInt)*sizeof(integer);
  { move new array }
   move(tmp.ffk.FBits^,ffk.FBits^,p);
  { free temporary class }
  tmp.free;
end;
end.
```

➤ *Listing 1*

SHLs), but my version can be understood and is not too slow. Perhaps one of the Delphi gurus could write an article on fake classes and how to overcome the `private` constraint?

Contributed by Reinhard Greeven,
Email: rgreeven@cincom.com

Version Info

File version information is essential to a better life. Version info is operating system supported and works under all versions of Windows. It's used by the OS and installation programs (or you) to ensure that your files and their dependents are as you think: the latest version!

Unfortunately, Delphi doesn't add version info to your EXE or DLL, thus making it very hard on the OS, you and your installation program to ensure that what you want to run with is actually what you get! To add a `VERSIONINFO` resource to your Delphi app simply do the following.

First, build the file TEST.RES and add it to any unit in your project. The file is built by compiling the file TEST.RC, a Windows resource script – an example is shown in Listing 2. To turn the .RC file into a .RES file you have to compile it using a resource compiler. Borland's 32-bit one is called BRC32.EXE (note that BRCC32.EXE is something else!). A typical command line looks like this:

```
C:\PROGRAM FILES\DELPHI\BIN\BRC32 -R TEST.RC
```

This creates TEST.RES. The `-R` tells the compiler that you're going to be adding the .RES manually to your project: a good idea! To add the file to an existing unit add the directive `{$R TEST.RES}`. You could add this after the similar directive that Delphi adds:

```
{$R *.DFM}
{$R TEST.RES}
```

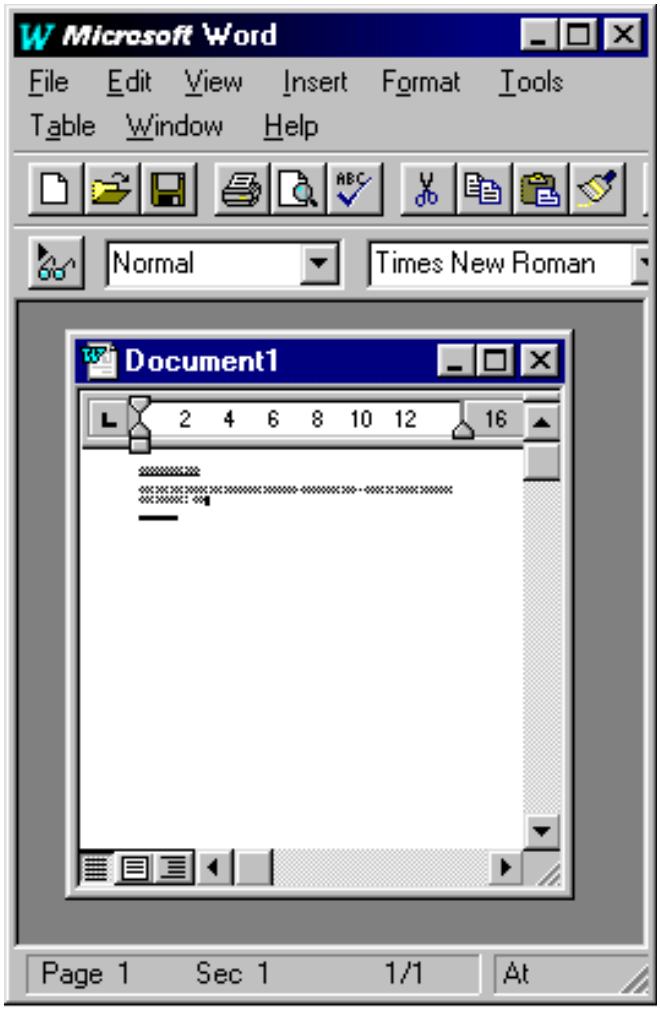
Note that Delphi concatenates your resources to those added already.

Now press `F9` to rebuild the project. That's it! To test whether you added it OK open either the File Manager or Explorer and locate your built .EXE file. Highlight the file and press `Alt Enter`. A dialog will appear in which you will find your version info. In Explorer it's on a separate page (in the tabbed dialog) called `Version`.

Contributed by Peter J Morris, TMS
Email: CompuServe 100016,2751

3D Window Borders

Have you ever noticed how Microsoft's MDI-based applications have a professional-looking, 3D border around the inner part of the main window? (see the example in Figure 1) If you try creating an MDI program with Delphi, you'll find that your main window has a much less interesting, 'flat' look to it. How do Microsoft get this effect?



➤ *Figure 1*

```
VS_VERSION_INFO     VERSIONINFO
  FILEVERSION             1,0,0,1
  PRODUCTVERSION          1,0,0,1
  FILEFLAGSMASK           0x3fL
  FILEFLAGS               0x0L
  FILEOS                  0x4L
  FILETYPE                0x1L
  FILESUBTYPE             0x0L
BEGIN
  BLOCK "StringFileInfo"
  BEGIN
    BLOCK "040904B0"
      BEGIN
        VALUE "CompanyName",
          "The Mandelbrot Set (Int'l) Limited\0"
        VALUE "FileDescription",  "Test Application\0"
        VALUE "FileVersion",      "1, 0, 0, 1\0"
        VALUE "InternalName",     "TEST\0"
        VALUE "LegalCopyright",
          "Copyright \251 1996\0"
        VALUE "LegalTrademarks",  "\0"
        VALUE "OriginalFilename", "Test\0"
        VALUE "ProductName",      "Test Application\0"
        VALUE "ProductVersion",   "1, 0, 0, 1\0"
        VALUE "FileVersion",      "1, 0, 0, 1\0"
      END
    END
    BLOCK "VarFileInfo"
      BEGIN
        VALUE "Translation", 0x409, 1200
      END
END
```

➤ *Listing 2*

An MDI application actually uses three different types of windows. Firstly, there's the outermost frame window which contains all the other windows: it's the frame window which hold the menus for your application. Secondly, there's the MDI 'client' window which corresponds to the inner part of the frame window. Finally of course, there are the individual document windows themselves.

Getting a 3D border around the edge of the client window is just a matter of setting a new, Windows 95 extended style bit when creating the client window. The VCL framework allows you to get at the client part of an MDI form by using the `ClientHandle` field.

➤ *Listing 3*

```
constructor TMainForm.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  SetWindowLong(ClientHandle, GWL_EXSTYLE,
  GetWindowLong(ClientHandle,
    GWL_EXSTYLE) or WS_EX_CLIENTEDGE);
  SetWindowPos (ClientHandle, 0, 0, 0, 0, 0,
    swp_DrawFrame or swp_NoMove or swp_NoSize
    or swp_NoZOrder);
end;
```

The code in Listing 3 shows how to add the 3D effect to your own applications: it's just a matter of setting the `WS_EX_CLIENTEDGE` style bit for the client window. The `SetWindowPos` call is just a sneaky way of ensuring that the window redraws itself immediately. Without this call in there, you'll find that the 3D effect doesn't appear until you resize the form.

Contributed by Dave Jewell with thanks to hgibson@cix.compulink.co.uk

Where Is It?

I was looking for some code that searched all the drives available to a user for a particular file or set of files. I cam across some freeware components none of which did quite what I wanted. The example shown in Listing 4 fills a listbox with a list of all the DLLs and their paths. *[Note the use in this example of several layers of nested procedures and functions, plus recursion. Editor].*

Contributed by Tom Corcoran, tomc@unitime.com

➤ *Listing 4*

```
procedure FindFile(
  initialPath : string;      { initial path }
  fileMask : string;         { mask to look for }
  recursive: boolean;        { search subdirectories? }
  stopOnFirstMatch: boolean; { one match? }
  files: TStringList);       { add match(es) to list  }
{ Starting at <initialPath>, FindFile will look for a
  match <fileMask>, if <Recursive> is True, all
  subdirectories beneath <initialPath> will be visited
  as well. If an initialPath is not given FindFile
  searches all non-removeable drives. Adds the paths
  where <fileMask> found to <files> }
type
  TAryDrive = array[0..25] of char;
const
  aryDrive: TAryDrive = ('a', 'b', 'c', 'd' ,'e', 'f',
     'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
     'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z');
var
  currentPath, currentDrive: string;
  i: byte;
  function IsDriveValid(drive: integer): boolean;
  { returns true if a valid drive }
  begin { IsValidDrive }
    { not searching removable drives }
    Result := not (GetDriveType(drive) =
      DRIVE_REMOVABLE);
    if Result then begin
      ChDir(Format('%s:', [aryDrive[drive]]));
      Result := (IOResult = 0);
    end;
  end;
  procedure SearchDirectory(fileMask: string;
    path: TFileName);
    function MakePath(path, fileName: TFileName) :
      TFileName;
      function AddSlashIfNeeded(path: string): string;
      begin { AddShlashIfNeeded }
        Result := Path;
        if (not (path = '') and
          not (path[Length(path)] = '\')) then
            Result := Format('%s\', [Result]);
      end;
    begin { MakePath }
      Result := Concat(AddSlashIfNeeded(path), fileName);
      writeln(Format('Path = %s', [Result]));
    end;
  var
    searchRec: TSearchRec;
    stopped: boolean;
  begin  { SearchDirectory }
    { search the current directory for fileMask }
    stopped := False;
    try
      if FindFirst(MakePath(path, fileMask), faAnyFile,
        searchRec) = 0 then
        repeat
          if searchRec.Attr <> faDirectory then begin
            files.Add(ExtractFilePath(MakePath(Path,
              searchRec.Name)));
            if stopOnFirstMatch then
              stopped := True
          end
        until (FindNext(searchRec) <> 0) or stopped;
    finally
      FindClose(searchRec);
    end;
    if recursive then
      {search the subdirectories for fileMask }
      try
        { Search current directory for subdirectories }
        if FindFirst(MakePath(path, '*.*'), faDirectory,
          searchRec) = 0 then
          repeat
            with searchRec do
              if (Name <> '.') and (Name <> '..') and
                (Attr = faDirectory) then
                SearchDirectory(fileMask,
                MakePath(path, Name));
              { we have to be gentle to the others apps }
              Application.ProcessMessages;
          until (FindNext(searchRec) <> 0) or stopped;
      finally
        FindClose(searchRec);
      end;
  end;
  begin { FindFile }
    if initialPath <> '' then
      SearchDirectory(fileMask, initialPath)
    else begin
      { bookmark current drive and directory }
      GetDir(0, currentPath);
      currentDrive := Copy(currentPath, 1, 1);
      for i := 0 to High(aryDrive) do
         if IsDriveValid(i) then
           SearchDirectory(fileMask, Format('%s:\',
             [aryDrive[i]]));
      { reset to previous path }
      ChDir(Format('%s:', [currentDrive]));
      ChDir(currentPath);
    end;
end;

procedure TForm1.FormCreate(Sender: TObject);
const
  fileMask = '*.dll';
var
  files: TStringList;
begin
   try
     { fill combo box with all paths
       that contain <filemask> }
     files := TStringList.Create;
     FindFile('', fileMask, True, False, files);
     comnboBox1.Items.Assign(files);
   finally
     files.Free;
   end;
end;
```