Performance Optimisation Part 2

by Bob Swart

In the last issue, we started a small application which used two components: `TFileListBox` and `TDirectoryOutline`. We enhanced the `TDirectoryOutline` component's performance by modifying the VCL code in DIROUTLN.PAS (remember that any enhancements made in the 'local' version of the `TDirectoryOutline` code and .DCU file override the version in COMPLIB.DCL). I called this feature *bottom-up efficiency*, as it enables us to write more efficient versions of components, enhancing efficiency by simply recompiling the applications which use these components!

This time, we'll examine other possible performance enhancements and efficiency issues for Delphi, starting with the new Object Pascal features in Delphi.

Language Enhancements

The Object Pascal language used in Delphi contains many helpful new features for programmers who are concerned with efficiency.

Virtual vs Dynamic

For starters, we now have two ways to indicate whether or not a method can be overruled in a descendant class. We can declare methods as either `virtual` or `dynamic`. The difference is the algorithm that is used to find the correct (polymorphic) method to use at run-time.

If you declare a method as `virtual`, the compiler will find the correct method by walking the virtual method table (VMT) of the instance. If you declare a method as `dynamic`, the compiler will use another algorithm, which uses less storage space, but is slower. So, again we see here a trade-off between size and speed and Delphi offers us the chance to make our own decisions about it! If you want a small program, or need a faster one, check your `virtual` and `dynamic` routines.

Open Arrays

A very nice feature of Object Pascal is the ability to use open arrays when passing an array to a routine where the routine does not know beforehand exactly how many elements are in the array. The elements of the open array are numbered from 0 (just like C/C++ and the `PChar` type). Using the `High` function on an open array, we can get the maximum index number. This enables us to write one routine which is capable of accepting arrays of several different lengths, thereby reducing code size! Consider for example the function `Min` in Listing 1, which returns the minimum value of an open array of integers.

The function contains a possibly fatal error: the open array is supplied as *value* parameter to `Min`. This not only costs a lot of time in copying the open array to the stack, it might also blow the stack! A *value* open array is copied onto the stack before the routine is executed (including any stack checking code) and since an open array can be up to 64Kb in size (just like the stack), this offers lots of `RunError(202)` opportunities. So, I urge you to always use `Const` or `Var` when passing open arrays. They not only save time, they also prevent stack overflows!

Run Time Type Information

RTTI enables Delphi to identify each instance of a class. This is used, for example, by the Object Inspector to display the properties of a class instance. But we can use RTTI to our own advantage, too, using the two new keywords `is` and `as`. Using `is` we can check whether or not an instance is of a specific class type, while `as` can be used to do a RTTI-safe typecast (one that will raise an exception if RTTI identifies the cast as invalid):

```
if Sender is TButton then
  with (Sender as TButton) do
  ...
```

Note that we can omit the `as` and use an untyped cast since the `is` operator already identifies `Sender` as being of type `TButton`. So, a faster version of the the last line is:

```
with TButton(Sender) do ...
```

Exceptions

Delphi exceptions, including the new keywords `try`, `except`, `on ... do` and `raise`, offer a new way of detecting and handling errors. Many programmers, however, are concerned about the efficiency of exceptions compared to regular error handling.

My timings have shown that the use of exceptions has no significant effect on the performance of a certain piece of code, as long as the regular (non-exception) path is followed. Where an exception is raised, I noticed some overhead. But what the heck, I don't mind waiting just a little bit longer to get an error message anyway.

➤ *Listing 1*

```
program CrtApp;
uses WinCrt;
function Min(A: Array of Integer): Integer;
var i: Word;
begin
  Result := High(Integer);
  for i := Low(A) to High(A) do { note: Low(A) is always 0! }
    if A[i] < Result then Result := A[i]
end {Min};
begin
  writeln(Min([1]));
  writeln(Min([8,4,2,6,5,3,7,12]));
end.
```

This does mean, however, that exceptions should not be used in tight loops where the exception is just part of the execution flow (like reading a text file and raising an 'end-of-line' exception at the end of each line). This kind of programming will slow down your application, since for each exception which is raised an instance of `EException` is created, your stack is cleared and walked to the nearest exception handler. For normal error detection and handling exceptions are just fine and often make the code that much more readable!

Finally

Measurements of the `try ... finally` block which was also introduced with Delphi exceptions don't indicate any significant performance loss either. This, combined with the fact that a `try ... finally` block is a very helpful way to safeguard resources and prevent any leakages, makes this my preferred way of programming!

Assembly

As we saw with `TDirectoryOutline`, an efficient algorithm or data structure can make a difference of an order of magnitude. Furthermore, new Delphi language features are able to decrease code size or increase speed and safety, although the results are less dramatic compared to algorithmic improvements.

Every now and then, we'll meet a situation where this just isn't enough and we have to use assembly language in one of these three forms (check out *The Pascal Magazine* Issue 7 for more details):

- **BASM (Built-in ASseMbler)**. This kind of assembler offers almost anything you'd ever need, while a lot of the fuss is already taken care of. We can simply insert assembly statements between `asm` and `end` and don't have to worry about setting up our own stack frame for example.
- **InLine macros**. These macros consist of hexadecimal (and so almost unreadable) statements which are substituted for their 'call'. They offer a great opportunity for performance enhancements, but do increase executable file size.
- **External .ASM files**. Finally, if all else fails, you can step down to an external .ASM file which compiles to an .OBJ file, which can be linked in with the `{$L file.OBJ}` statement.

Code Size

Sometimes, we just need to minimise code size or stack/heap usage, especially if these resources are low. While Delphi executables are generally larger than comparable Borland Pascal applications, they are about the same size as corresponding C++ applications (of course, using Delphi the application itself is much easier to produce!).

Delphi does offer some special features to try to minimise code size, by supporting code re-use in a structured way. For example, event handlers...

Back To The Example

Let's add some more features to the example application we started last time. Drop `Rename` and `Copy` buttons onto the form. Now, if we double click on the `Rename` button, we can write instant code for the button click event. The same goes for the `Copy` button. It would be nice if we could re-use existing code and event handlers, and we can! We just assign the same event handler to the `OnClick` event in both buttons. Of course, we need to change the event handler to discriminate between both buttons, as shown in Listing 2.

Gauge CallBack

Note that the callback routine we used in the example above is

➤ *Listing 2*

```
procedure CallBack(Position, Size: LongInt); export;
var Progress: LongInt;
begin
  Progress := (Position * MainForm.Gauge1.MaxValue) div Size;
  MainForm.Gauge1.Progress := Progress { update Gauge }
end {CallBack};

procedure TMainForm.RenameButtonClick(Sender: TObject);
var
  CallBackProc: TCallBack; { procedure(Position, Size: LongInt); }
  f: File;
  FileName: String;
  Len: Byte absolute FileName;
begin
  if FileListBox1.FileName <> '' then begin
    FileNameDialog.OldFileName.Text := FileListBox1.FileName;
    FileNameDialog.NewFileName.Text := FileListBox1.FileName;
    if FileNameDialog.ShowModal <> idCancel then begin
      FileName := FileNameDialog.NewFileName.Text;
      try
        try
{ ==> }  if Sender = RenameButton then begin
            System.Assign(f,FileListBox1.FileName);
            System.Rename(f,FileName)
          end else begin
{ ==> }     { Sender = CopyButton }
            Gauge1.Progress := 0;
            CallBackProc := CallBack;
            FastFileCopy(FileListBox1.FileName,
              FileNameDialog.NewFileName.Text, CallBackProc)
              { see source code on disk for "callback" details }
          end
        except
          on E: Exception do MessageDlg(E.Message, mtError, [mbOk], 0)
        end;
      finally
        FileListBox1.Directory := '.';
        {Force Rescan}
        FileListBox1.Directory := DirectoryOutLine1.Directory;
        { Bug? TFileListBox chokes if the filename ends on a '.' }
        if FileName[len] = '.' then Dec(len);
        FileListBox1.FileName := FileName
      end
    end
  end
end;
```

connected to the TGauge. As with TDirectoryOutliner, the Delphi Gauge component is just a sample which Borland included. It doesn't do any fancy optimisation of screen refreshes. Indeed, we can see that without using the Gauge CallBack routine our application seems to run much faster. If we look at the code for the SetProgress routine in the VCL we see:

```
fCurValue := Value;
Refresh;
```

This means that whenever the progress value is changed the entire control will be redrawn. There are a number of ways to make this faster. One way is to check if the screen is actually likely to have changed. For example, instead of calling Refresh, do the following:

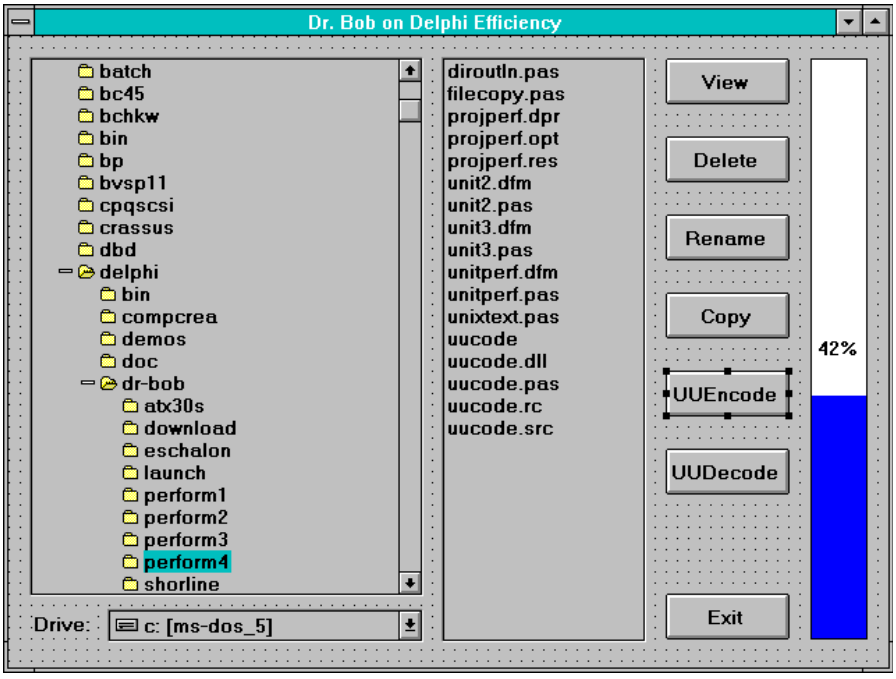
```
if abs(fCurValue-fLastDrawn) >=
  fDisplayDelta then begin
  Refresh;
  fLastDrawn := fCurValue;
end;
```

Another approach is to use a timer and to do a Refresh only, say, 20 times per second if SetProgress was called and a refresh needs to be done. Of course, we can combine the two strategies and come up with a really fast implementation of a TGauge, which is left as an exercise for the reader...

Adding Final Features

Time for the last additions: two new buttons to UUEncode or UUDecode the file selected using the FileListBox. The UUEncode and UUDecode routines are implemented in a 'foreign' DLL (see *Under Construction* in Issue 4) and this section will show how to write and use DLLs for Delphi efficiently (again, 'bottom-up efficiency').

UUEncode, by the way, is the process of encoding a file which may contain any characters (including high ASCII characters) into another file which uses a standard, limited, character set, so the encoded file can be reliably sent over many communication networks, such as the internet. A nice feature to include!



➤ *The example application, now with added features!*

```
library UUCode;
function UUEncode(FileName: PChar): Word; export;
begin
  ...
end {UUEncode};
function UUDecode(FileName: PChar): Word; export;
begin
  ...
end {UUDecode};
var  SaveExitProc: pointer;
procedure NewExitProc; far;
begin
  ExitProc := SaveExitProc
end {NewExitProc};
exports UUEncode index 1,
        UUDecode index 2;
begin
  SaveExitProc := ExitProc;
  ExitProc := @NewExitProc
end.
```

➤ *Listing 3*

UUCode DLL

First of all, a DLL is a library of routines and resources which is linked into your application at run time instead of compile time. The run time linking allows a DLL to be shared by multiple applications, thus saving on memory and resource usage and DLL load time.

Init And Exit

Each DLL has its own Init (loading) and Exit (unloading) code. Note that while a DLL can be used by many applications at the same time, the Init and Exit are only executed once, not for each application. In Object Pascal, we can express this as shown in Listing 3 for our UUCode DLL.

As a DLL is shared by a lot of applications, you can even enhance performance by placing shared code or resources in a DLL. Since the DLL won't have to be loaded by the second application, this saves on start-up time. Of course, loading each DLL, especially if we've got more than one DLL, takes more time than loading only one standalone EXE file would.

Loading And Unloading

Loading a DLL is usually achieved by writing an import unit and this can be done in two ways: either

automatically (with an *implicit* import unit) or by hand using `LoadLibrary` (with an *explicit* import unit). Implicit import units are very easy (see Listing 4), but can introduce a few problems when the DLL is not available, such as an error message from Windows telling us that the DLL wasn't found.

It's possible to check for a DLL's presence before attempting to use it. You'll have to change your interface unit by changing all of the procedure definitions to *procedural variable* declarations. By eliminating all of the `external 'UUCode' index IndexNum;` clauses you will eliminate the implicit reference to the DLL and the automatic attempt to load it. To make UUCode.DLL a DLL that can be conditionally loaded, the interface unit might look something like Listing 5.

This second import unit causes UUCode.DLL to be conditionally loaded. The variable `UUCodePresent` can be tested in the program which uses this unit, so it calls the DLL only when `UUCodePresent` is true. However, note that this is slightly slower than implicit loading of the DLL. Hence, we have a trade-off between safety and speed!

Unloading the DLL can also be done either automatically, or by hand with `FreeLibrary`.

Using a DLL

The data segment of your DLL is shared by all calling applications, and is limited to 64Kb minus the local heap. A DLL has no stack for its own, but uses the stack of the calling application. Therefore, your DLL should be conservative about stack space; allocating 8Kb or so of local variables on the caller's stack would be a bit 'rude', and can lead to a GPF real fast. Also, you might want to arrange the functions into logical groups and (using different units) mark the code segments as `loadoncall` and `discardable`. If they are marked as `preload`, Windows will try to drag them all into memory at start-up, which may take a long time.

Timing a DLL

We can profile a DLL with Turbo Profiler (you need the one from Borland C++ 4.5x) . When you load a program that uses DLLs into Profiler, it checks for symbol tables (as with an EXE file) and automatically loads the symbol table and source of every linked DLL.

If your DLL is loaded automatically, you can set profile areas right from the start by picking the DLL as the Module to view. If the DLL is loaded by hand with `LoadLibrary`, you must set a Stop area marker right after the `LoadLibrary` call, in order to get to the DLL Module to set the profile areas. This hack is actually another reason why I generally prefer import libraries to automatically load DLLs instead of using `LoadLibrary`. Note that you'll get a separate .TFS and .TFA file for each DLL which is loaded by your program.

Load Time Efficiency

Splitting your Windows application into several DLLs to support your main application may have introduced another problem: load time.

Each DLL will have to be loaded, whether it's dynamically or automatically, and this takes time. You cannot expect each DLL to be already loaded by another application. The load time of a DLL depends on two factors: the number of functions inside the DLL which have to be imported and assigned to your import library functions, and the total code size of the DLL.

Optimising the individual functions for speed and size is something we've done before. Optimising the total DLL code size seems like something we cannot do much about, or can we?

➤ *Listing 4*

```
unit UUCode;
{ implicit import unit for UUCODE.DLL }
interface
function UUEncode(FileName: PChar): Word;
function UUDecode(FileName; PChar): Word;
implementation
function UUEncode; external 'UUCODE' index 1;
function UUDecode; external 'UUCODE' index 2;
{ supplying the index number increases performance slightly,
  since otherwise the function that is to be dynamically linked
  has to be searched for by name }
end.
```

➤ *Listing 5*

```
unit UUCode;
{ explicit import unit for UUCODE.DLL }
interface
uses Wintypes, Winprocs, Win31;
Const UUCodePresent: Boolean = False;
var
  UUEncode: function(FileName: PChar): Word;
  UUDecode: function(FileName: PChar): Word;
implementation
var
  SaveExitProc: Pointer;
  DLLHandle: Word;
procedure NewExitProc; far;
begin
  if DLLHandle >= 32 then FreeLibrary(DLLHandle);
  ExitProc := SaveExitProc
end {NewExitProc};
begin
  SetErrorMode(SEM_NOOPENFILEERRORBOX);
  DLLHandle := LoadLibrary('UUCODE.DLL');
  if DLLHandle >= 32 then begin
    SaveExitProc := ExitProc;
    ExitProc := @NewExitProc;
    @UUEncode := GetProcAddress(DLLHandle,'UUENCODE');
    @UUDecode := GetProcAddress(DLLHandle,'UUDECODE');
    UUCodePresent := True; { we can use UUEncode & UUDecode now }
  end
end.
```

W8LOSS

If you've installed the command-line tools with your copy of Delphi you will have a program called W8LOSS in your DELPHI\BIN directory. This program is able to shrink the file size of Windows 3.x New Executable programs (such as EXE, DLL, SCR, and others), in order to speed program loading and startup.

W8LOSS uses the Windows' loader support for chaining relocation records together by utilising the target address for the head record as a link to another offset requiring the identical fixup value. The chain is terminated by the occurrence of 0xFFFF in the segment data at the fixup's target offset. This results in executables and DLLs that are typically 5% to 20% smaller after running W8LOSS, even files that are not written in Delphi (such as the foreign UUCode.DLL, which was actually written in Borland Pascal).

Conclusions

In these first two articles, I've described various tools and techniques which are available for Delphi performance optimisation. We've seen a structured performance optimisation process, where we (top-down) break down the application and optimise step by step. We've also seen special bottom-up optimisation, a feature introduced by the re-usable component nature of Delphi!

All the source code for the examples and units (with the exception of the source for the UUCode.DLL) can be found on the free disk with this issue of The Delphi Magazine.

Next time

Once Delphi 2.0 is released, we'll return to the topic of performance optimisation and focus on the 32-bit programming world, exploring some Delphi 2.0 optimisation techniques.

Bob Swart (you can email him at 100434.2072@compuserve.com) is a professional 16- and 32-bit software developer using Borland Pascal, C++ and Delphi. In his spare time, he likes to watch video tapes of Star Trek Voyager with his 1.75 year old son Erik Mark Pascal.

Acknowledgements

The first two parts of this series are based on my talk in session DL390 (Delphi Performance Optimisation) from the 6th Annual Borland Developers Conference, August 6-9 1995, in San Diego, USA.