Performance Optimisation Part 3: 32-bits

by Bob Swart

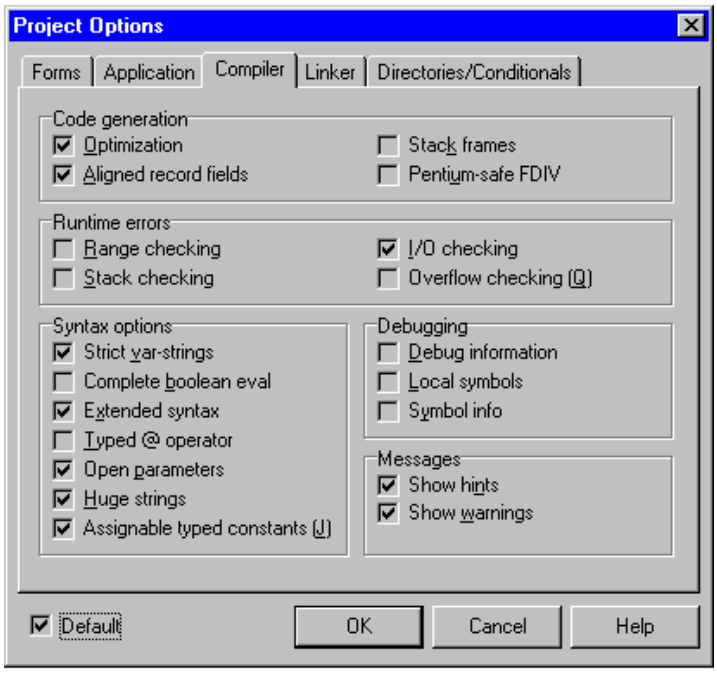
Back in the November 1995 and January 1996 issues I investigated tools and techniques to improve the speed of 16-bit Delphi 1.0 applications. Now, with the release of Delphi 2.0, it's time to take a look at 32-bit specific issues.

Delphi 2.0

Before we even start to think about 32-bit performance optimisation, we need to check out Delphi 2.0's compiler options. Although compiler options have less impact than a bad algorithm, choosing the wrong ones can seriously slow down your code and expand your total code size.

My personal project options for Delphi 2.0 are shown in Figure 1. Note the checked `Default`, which means that these options are now default for every new project in Delphi on my machine.

➤ *Figure 1*



Compared to Delphi 1.0 there are a few new things. There is now an `Optimization` option. Frankly, I don't know why you should want to have this one turned off. It only takes a fraction longer to compile and link than without it, so why not use it all the time? Another change is the `Aligned record fields` option, which will align fields within records to 32-bit boundaries. In Win32 land, this field alignment means faster execution compared to non-alignment, since `DWORD`-sized items on `DWORD` addresses are accessed much faster than on `DWORD`-odd addresses. Note that you must use this option with care if you're porting Delphi 1.0 code over to 2.0, since the record layout may change, which is especially harmful if you're reading or writing files that contain these (previously non-aligned) records.

The `Stack frames` option forces the compiler to generate stack frames on all procedures and functions. Using this option will slow down your performance, but on the other hand may enhance debugging capabilities.

Another new compiler directive is `Huge strings`, a local switch directive, which controls the use of long string types. This `$H` directive switches the meaning of the reserved word `string`, when used alone in a type declaration. With `$H+` a `string` refers to a new, longer counted string type called `AnsiString`. This is the default setting. With `$H-` a `string` refers to a 255-character string (or the `ShortString` type), as in Delphi 1.0.

In Delphi 1.0 we could define typed constants, of the form

```
const X: Integer = 42;
```

Delphi 2.0 now sees these definitions as *non-changeable*, and we must use

```
var X: Integer = 42;
```

to define pre-initialised variables. To make sure any old Delphi 1.0 code which uses typed constants that are changed in other places in the code still compiles, we need to set the `Assignable typed constants` or `$J+` option.

Finally, the `Show hints` and `Show warnings` options will only help you while writing code. They may identify that you've declared variables that are never used, or assigned a negative value to a `Word` and many more.

Linker Options

Apart from compiler options, we can also set linker options. A much simpler dialog, as shown in Figure 2 (over the page).

Delphi 2.0 uses a new 32 bit linking technology which also includes several optimisations. The new linker is 20% to 50% faster than previously because of a new unit caching scheme. In addition, EXEs are 20% to 25% smaller than before.

Finally, Delphi 2.0 supports the OBJ file format, so you can (try to) share code between Delphi 2.0 and C/C++, in addition to being able to create and share DLLs as before.

In Delphi 1.0 there was also an option `Optimize for size and load time` in the linker page. Note that on some machines in some situations this may sometimes lead to an erroneous *disk full* message when there is actually plenty disk space free. The fix in this case is to de-select this option and run the stand-alone program W8LOSS.EXE on the compiled program instead (the linker uses the W8LOSS.DLL).

In Delphi 2.0 this option has disappeared from the linker page. It's been fixed and is now *always* done (remember that Delphi 2.0 now shares the same optimising back-end with Borland C++).

If we develop from the IDE, we can easily obtain the current settings of the compiler directive by typing `Ctrl O O` in edit-mode, resulting in the current settings being pasted at the beginning of the source, for example:

```
{$A+,B-,D-,F-,G+,I+,K+,L-,P+,
  Q-,R-,S+,T+,U-,V-,W-,X+,Y-}
```

We can then change them to our liking and also make sure that a re-compile on another user's system (with perhaps other compiler options set in the IDE) still yields the same results. You can also view the options in the project's .OPT file for Delphi 1.0, or .DOF file for Delphi 2.0.

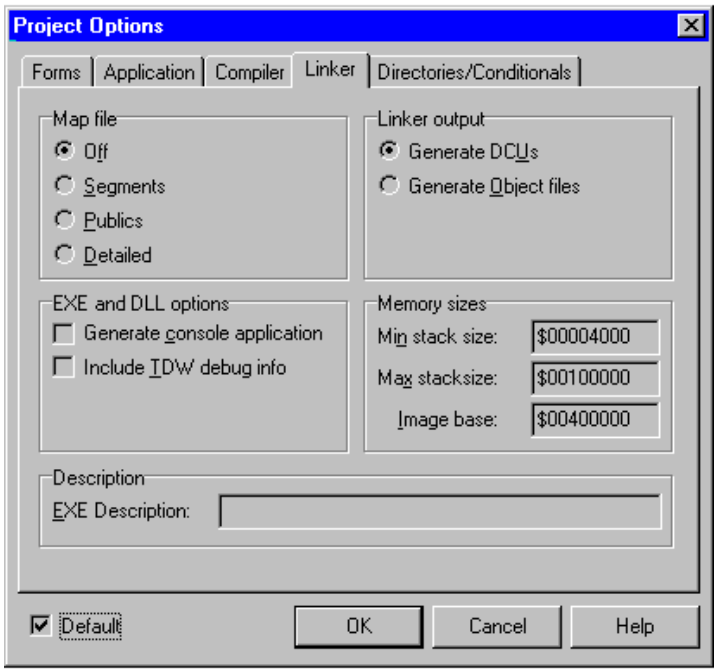
Delphi 2.0 Optimisations

The new 32-bit native code compiler achieves its performance increase by using a number of new code optimisation techniques. Unfortunately, we cannot select one or more of these techniques on an individual basis: it's all or nothing, depending on the `Optimization` switch setting. The techniques used include register optimisations, call stack overhead elimination, common sub-expression removal and loop induction variables, which result in faster performance for much code.

Register Optimisations

Heavily used variables and parameters will automatically be placed into registers, reducing the number of machine instructions

➤ *Figure 2*



required to load, access and store a variable. This results in much faster code since there is no need to load variables from memory into registers. This optimisation is done automatically by the compiler with no need to specify that certain variables or parameters should be placed in registers. The compiler also automatically performs variable *lifetime analysis* in order to be able to re-use registers. For example, if a variable `I` is used exclusively in one section of code and a variable `J` is used exclusively in a later section of code, the compiler will use a single register for both `I` and `J`.

Call Stack Overhead Elimination

When possible, parameters passed to functions or procedures will also be placed in CPU registers. Not only does this eliminate the memory access similar to the register optimisation I described earlier, but it also means that there is no need to set up a stack frame in which to store the values temporarily. This eliminates additional instructions to create and destroy the stack frame so that function calls are as efficient as possible. In practice I have found it important to minimise the number of arguments to functions and procedures to enable the compiler to pass them all in registers. If you have more than, say, four or five parameters, the chances are high that the compiler won't find enough free registers, so it needs to set up a stack frame after all.

Eliminating Common Sub-Expressions

As the compiler translates complex mathematical expressions, it will ensure that any common sub-expressions, that is computations which would be performed more than once, will be eliminated. This allows us to write code in a manner that is clear and easy to read, knowing that the compiler will automatically reduce it to its most compact and efficient form. Very helpful!

Loop Induction Variables

The compiler automatically uses loop induction variables as a way to speed up access to arrays or strings within loops. If a variable is used only to index into an array, for example in a `for` loop, the compiler will *induce* the variable, eliminating multiplication operations and replacing them with a pointer which is incremented to access items in the array. In addition, if the variable size is a 1, 4 or 8, Intel scale indexing is used to provide additional performance benefits.

Code Elimination

Hey! Will the compiler remove my code? Well, yes, if you've written

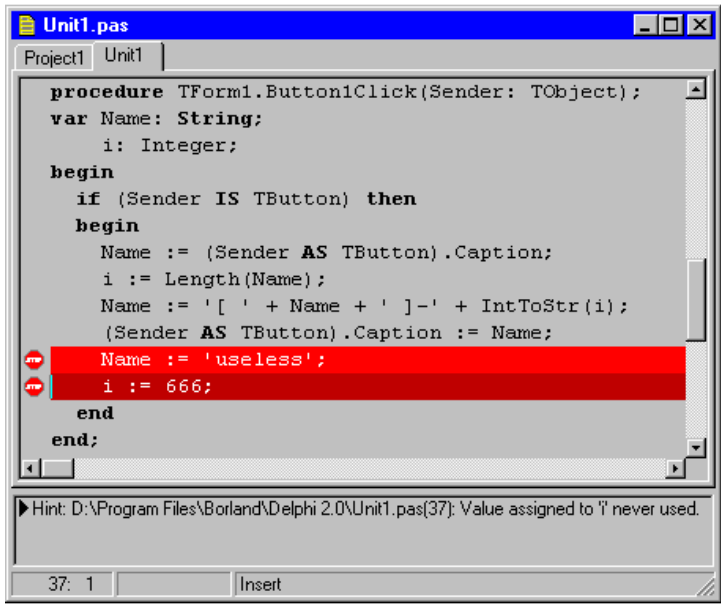
code that is not used the optimising compiler can decide to remove lines that have no effect.

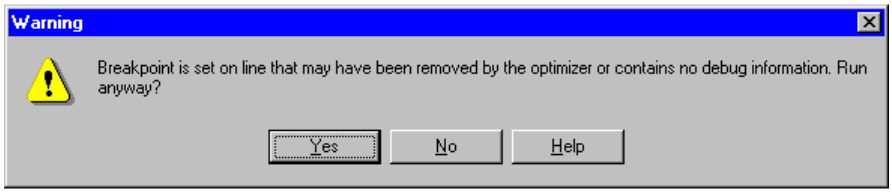
Note that this is different from removing dead code, as this code is not dead (it can be reached) it just doesn't have any effect!

For example, in the program shown in Figure 3, the last line before the `end` is removed from the executable when we have optimisation enabled. This means we still can set a breakpoint to it, but it won't get triggered! Fortunately, the Delphi IDE will give us a warning when we try to run this program as shown in Figure 4.

Note that we *will* get to the breakpoint on the line with the string assignment. And we didn't get a warning about it either. So it seems that there's a lot going on behind the scenes of string assignments that we may need to explore on another day...

➤ *Figure 3*





➤ *Figure 4*

Finding Bottle-Necks

As with 16-bit programs, you must never *think* you know where performance bottle-necks are in your code. Always *measure* with some kind of tool. Unfortunately, there is no Turbo Profiler we can use to profile our 32-bit Delphi applications (for Delphi 1.0 we could use the Turbo Profiler from Borland C++ 4.5x, but there is no 32-bit profiler yet, even with BC++ 5.0).

For 16-bit Windows applications the `GetTickCount` API gives us the number of milliseconds since Windows was started. The accuracy of this API, however, is only 55ms, as it is updated 18.2 times each second (just like the real mode system clock). Hence, we should use `GetTickCount` only for very rough measurements.

From the 16-bit MMSYSTEM.DLL we can use the `timeGetTime` function (at index 607), which returns a `LongInt`. This API is accurate to one millisecond, so can be used for fine measurements.

These two APIs only show the total amount of time that has elapsed during a certain operation, without taking into account the time that other applications may have consumed during this time period). For truly insightful measuring, we can use the function `TimerCount` from TOOLHELP.DLL, which tells us the time spent only in our virtual machine.

The `GetTickCount` function is also available in the Win32 API and is identical to the `GetCurrentTime` function. Note that the `GetTickCount` and `GetMessageTime` functions actually return different times. `GetMessageTime` gives the Windows time when the given message was created, not the *current* Windows time.

The 32-bit version of `timeGetTime` retrieves the system time, in milliseconds. The system time is again the time elapsed since Windows was started. There is also another 32-bit API to do roughly the same thing: `timeGetSystemTime`. The only difference is that the latter uses the `MMTIME` structure to return the system time. The `timeGetTime` function has less overhead. Note that the value returned by the `timeGetTime` function is a `DWORD` value (we should use a `Longint`).

When using `timeGetTime` on Windows NT the default precision can be five milliseconds or more, depending on the machine. You can use the `timeBeginPeriod` and `timeEndPeriod` functions to increase the precision of `timeGetTime`. If you do so, the minimum difference between successive values returned by `timeGetTime` is only as large as the minimum period value set using `timeBeginPeriod` and `timeEndPeriod`. You can use the functions `QueryPerformanceCounter` and `QueryPerformanceFrequency` to measure short time intervals at a high resolution.

For `timeGetTime` on Windows 95 the default precision is 1 millisecond. In other words, it can return successive values that differ by just 1 millisecond. This is true no matter what calls have been made to the `timeBeginPeriod` and `timeEndPeriod` functions.

Algorithms And Data Structures

With every application that needs a performance boost, we first need to find the bottle-neck. You just don't simply start to optimise a program, you need to focus your attention to a specific target. Once we've found the bottle-necks in our application, the next thing we must

do is check the algorithm or data structures used. The chances are that an inefficient one is used and we can speed up the application by an order of magnitude by implementing a more efficient one.

Remember the example I used with Delphi 1.0? The component `TDirectoryOutline` (on the Samples page of the Component Palette) uses a linear search algorithm, with efficiency O(N*N), in procedure `BuildOneLevel`, to place new nodes in alphabetic order under their parent (see Figure 5 for the Turbo Profiler report showing the massive time usage in this section). Searching a directory with 100 subdirectories took more than 10 seconds, which is not acceptable, of course.

Although this sample component was updated to reflect the 32-bitness of Delphi 2.0 the search algorithm was not changed. We can solve the problem again by replacing the offending lines with a call to a binary searching algorithm, which results in a O(N * log N) performance instead of O(N*N). Now, searching a directory with 100 subdirectories takes under a second. Full source code for my revised DIROUTLN.PAS is on the subscribers' disk with this issue.

Language Enhancements

The 32-bit Delphi ObjectPascal language contains many new powerful (and sometimes even dangerous!) features for programmers who are concerned with efficiency. I'll focus on exceptions again, showing in an example that they do take a significant overhead when triggered, and also on variants. In future articles I'll look at some other Delphi 2.0 enhancements, such as long strings, and their performance impacts.

Exceptions

Delphi exceptions offer an excellent way of detecting and handling errors. 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*. When an exception is raised, I noticed a slight overhead.

```
0.0002 151    if RootNode.HasItems then {if has children, must alphabetise}
              begin
0.1655 146      TempChild := RootNode.GetFirstChild;
                { Dr. Bottle-neck: Linear Search applied: }
11.684 5326     while (TempChild <> InvalidIndex) and
                      (Items[TempChild].Text < SearchRec.Name) do
12.076 5180       TempChild := RootNode.GetNextChild(TempChild);
0.1105 146      if TempChild <> InvalidIndex then
0.6892 132        NewChild := Insert(TempChild, SearchRec.Name)
0.0115 14       else NewChild := Add(RootNode.GetLastChild, SearchRec.Name);
0.0001 146    end
```

➤ *Figure 5*

```
for R:=1 to 12 do begin
  try
    Value := StrToFloat(Cells[Column,R]);
  except { Warning: will also get fired when the cell contains an empty '' }
    Value := 0
  end;
  Sum := Sum + Value
end;
Cells[Column,13] := FloatToStr(Sum)
```

➤ *Listing 1*

But what the heck, I don't mind waiting just a little bit longer to get an error message anyway.

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 can really slow down your application, since for each exception raised, an instance of `EException` is created, your stack is cleared and walked to the nearest `on except` handler. For normal error detection and handling, exceptions are just fine, and often make the code that much more readable!

For example: create a string grid with a lot of rows and make the last row the total of the values in the other rows (in this column), by using the code shown in Listing 1.

The mechanism of exception handling takes time to execute, which is the reason why, for an almost empty grid, it takes a few seconds before we can enter another cell. It's not the calculation of the column totals, but the raising and especially the handling of the exceptions that takes a relatively enormous amount of time. So, I recommend you use exceptions to indicate errors that must be responded to, but never use them in the normal flow of control.

Variants

Delphi 2.0 introduces `variant` types to give you the flexibility to dynamically change the type of a variable. This is useful when implementing OLE automation or certain kinds of database operations where the parameter types on the server are unknown to your Delphi-built client application.

A `variant` type is a 16 byte structure that has type information embedded in it along with its value, which can represent a string, integer, or floating-point value. In most cases you can use a `variant` just as you would any other type of variable. When performing OLE automation, `variant` types can respond to method calls from the OLE server.

Typecasting Variants

You can typecast a standard-type expression into a `variant` or a `variant` expression into a standard type. If you have range-checking turned on, casting a `variant` into a value outside the range of the type being cast raises an exception. Note that using a `variant` inflicts a lot of processing overhead. Even if not converting, it must still access the type information to check whether or not it needs to convert. Hence, only use `variant` types if you need to, like for OLE stuff, and never use them in common code.

32-Bit Assembly

As we've seen with the `TDirectoryOutline` example, an efficient algorithm or data structure can make a difference of an order of magnitude. Furthermore, the new Delphi language features are able to decrease code size or increase speed and safety, although less dramatically compared to algorithmic improvements. With the new optimising back-end compiler and linker we seldom need to step down to the assembly level: the compiler will outperform all but the most ingenious assembler hacker.

Furthermore, the `inline` statement has now been removed in Delphi 2.0, so it is now no longer possible to write `inline` macros. I guess they felt that the compiler and linker already produce efficient enough code, or they were perhaps afraid that it would use registers in a way that was undetectable by the compiler so that normal register optimisations (values passed in registers) would break because of it. Anyway, `inline` is gone forever now, so we'd better start writing good solid algorithms in plain ObjectPascal again

Acknowledgements

A large part of this 32-bits efficiency article was based on information from chapter 18 (on Optimisation) of the book *The Revolutionary Guide to Delphi 2*, published by WROX Press, ISBN 1-874416-67-2, available now from your favourite bookseller (eg UK Delphi Developer's Group, email 100016.355@compuserve.com)!

Bob Swart (aka Dr.Bob on http://www.pi.net/~drbob/) is a professional software developer using Delphi, Borland Pascal, C++, HTML and Java. In his spare time he likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 2-year old son Erik Mark Pascal.