Delphi 6: A Cross-Platform Development Perspective

Reviewed by Dave Jewell

One of the key aspects of Delphi 6 is the ability to create new applications using either the good old VCL or the new-fangled Kylix-compatible CLX library. A decision you will have to make when creating new applications is whether to continue with the VCL, as before, or use CLX.

Of course, if you have clients banging on your door begging for Linux versions of your existing applications, or demanding cross-platform (Windows and Linux) software, the decision is easy: go for CLX, it's designed for cross-platform work. If you are not in that situation, you need to think carefully about where your future development path will be.

The demand for Linux applications does not seem to be as high as some folk were hoping, especially the marketing department at Borland. If there is going to be a significant demand for desktop Linux apps (as opposed to server apps), then my guess is that it won't be in the very near future. Also, on the Windows front, .NET is bearing down upon us rapidly. We understand that Borland is working very hard on a .NET-compatible version of Delphi. Although we have no ideas on timescales, clearly the sooner the better.

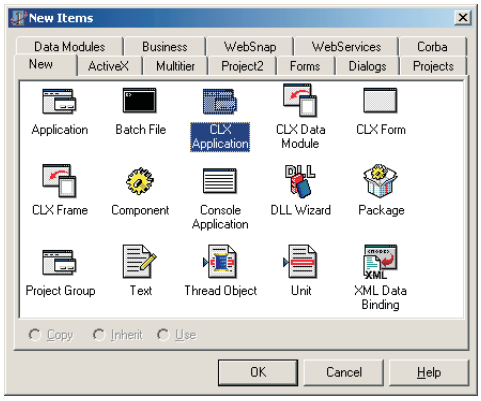
So, if you are going to learn a new framework, maybe .NET would be a more lucrative proposition. I don't want to sound like a party-pooper, and the last thing I want to do is dampen your enthusiasm for Kylix and Linux. That said, let's examine the implications of working with CLX rather than the VCL.

Deployment Issues With CLX

What about new applications? What sort of performance hit will you get by writing new code for CLX instead of the VCL? What are the deployment issues when it comes to releasing a Windows application, authored with Delphi 6, that's using CLX and the Qt classes? It's things like this that I want to address here, so let's roll up our sleeves and begin.

Creating a new CLX program is as simple as firing up the New Items dialog and selecting `CLX application`: see Figure 1. You'll notice that the CLX items are rather messily interspersed with the ordinary VCL stuff, and the same is true on the other tabs of the dialog: `CLX MDI Application` sitting side-by-side with `MDI Application`, etc. I would have preferred to see a two-level scheme where you first select CLX versus VCL (perhaps a `TTabSet` component at the bottom of the dialog?) and you then see all the items relevant to your choice of framework.

A nice touch is that the component palette is automatically reconfigured when working on a CLX application, showing only the CLX components which are installed. Similarly, you'll notice that the Property Inspector displays a somewhat different set of properties for `TForm` according to whether you're working on a VCL or CLX project. Such goodies as `DefaultMonitor` are obviously Windows-specific, and CLX does not have support for dockable windows, so properties like `DockSite` and events such as `OnDockDrop`, `OnEndDock`, will be conspicuous by their absence. Under CLX, all string properties use the `WideString` type.



➤ *Figure 1: Creating a new CLX app is as easy as selecting 'CLX Application' from the New Items dialog, although it would be nice if all the CLX items were more clearly distinguished from the VCL stuff.*

Yes, I know it's a drag (pun strictly intentional) but as I've said elsewhere, my hope is that once Borland tries to rebuild the Kylix IDE without the benefit of Wine/Winelib, it will buckle down and come up with a decent docking window implementation which can then be made available to all CLX clients.

A do-nothing CLX application built with Delphi 6 weighs in at a fairly reasonable 291Kb. Well, massive hard disks are ridiculously cheap now, so let's not quibble! Using the venerable Merlin EXE sniffer (Figure 2) we see that even this do-nothing executable pulls in a host of `Qxxx` units, including the base `Qt` unit which is responsible for interfacing with the QTINTF.DLL library. This DLL is the Windows equivalent of the libqtintf.so.2.2.4 Linux shared library which I've mentioned in the past. As under Kylix, its job is to map the procedural, non-OOP, calls made by CLX onto true object-oriented C++ method calls in the Qt code itself.

Actually, that's not 100% accurate. Under Linux, libqtintf.so.2.2.4 acted as a sort of go-between, connecting the procedural CLX calls to the 'real' Qt implementation which existed in *another* shared library. Under Windows, the situation is slightly different because the full Qt implementation is contained inside QTINTF.DLL. This explains

why libqtintf.so is around 1.5Mb in size, whereas QTINTF.DLL is 4Mb!

The bottom line is that you *must* include this 4Mb DLL when deploying CLX-built executables. If you think about it, Borland's decision to put the entire Qt implementation inside QTINTF.DLL actually makes perfect sense. Under Linux, a typical system will almost certainly have the Qt libraries already installed: remember that the KDE desktop system is itself architected on top of Qt, so unless you are the most fastidious of GNOME/GTK+ purists, Qt will be there somewhere. Thus, it makes sense for Kylix applications to be deployed with only the libqtintf.so 'glue' code needed to connect CLX to the core Qt library.

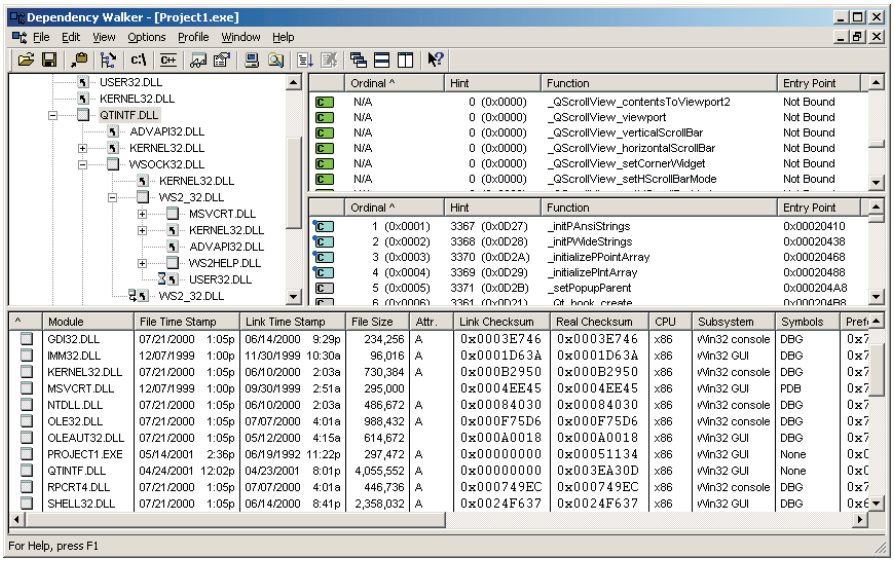
Under Windows, however, the scenario is quite different. It's unlikely that a high proportion of Windows systems have the regular (C++ callable) Qt DLL installed, and therefore, by bundling the core Qt code into QTINTF.DLL itself, we have a single DLL that's required for deployment purposes.

Figure 3 is a screenshot taken using Microsoft's Dependency Walking utility (DEPENDS.EXE) which makes it possible to see which DLLs are required by an application, which DLLs are required by the DLLs and so on, ad nauseum! As you can see, this clearly shows the various 'flattened' entry points into QTINTF.DLL which are required by the procedural CLX code. Thus we have `QWidgetList_next`, `QWidgetList_first`, and so on (the leading underscores simply indicate the use of the `cdecl` calling convention).

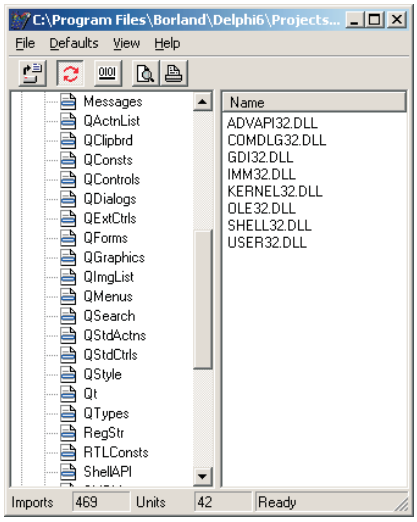
Prior to browsing the exports from QTINTF.DLL, I had hoped that Borland might retain the full C++ entry points, so that you could, for example, make direct Qt calls which bypass CLX, assuming that you can live with the pain of making cludgy C++ method calls direct from Pascal, perhaps using a judicious smattering of inline assembler code. However, this option definitely isn't available to adventurous developers, the only `QTINTF` exports are the flattened ones. Bearing in mind the stringent licence conditions on the use of Qt from Kylix and Delphi 6, it's not surprising that this particular back door has been closed!

On the negative side, I was surprised to discover that the `QTINTF` library has been erroneously built with a preferred load address of `$400000`. As the Borland documentation rightly states, *'This value is typically only changed when compiling to a DLL'*. Whoever built `QTINTF` should have read this, since giving the DLL the same load address as the application guarantees that the operating system will have to relocate the DLL before program execution begins! That's going to have a negative effect on program start-up time, particularly bearing in mind the large number of exports and fix-ups required by `QTINTF`. Microsoft's documentation states that a non-system DLL should have a load address between `$60000000` and `$68000000`. With modern hardware, the overhead is likely to be slight, but a little more attention to detail would have been nice here. If this bothers you, you can always use Microsoft's command-line `REBASE` program to change the load address of the DLL.

Switching over to a packaged CLX application brings the EXE file size down to a very trim 15Kb, but naturally introduces other dependencies such as RTL60.BPL (635Kb) and VISUALCLX60.BPL (1Mb). As ever, the simplest deployment option is to go with a non-packaged executable, unless you're deploying a suite of fairly large applications, all of which are built around CLX.



➤ *Figure 3: Here's Microsoft's dependency-walker application, DEPENDS, showing all the flattened procedure named exported from the QTINTF library.*



➤ *Figure 2: Merlin (my favourite EXE-file sniffer) demonstrates that a surprising number of 'Q' units get linked into even a do-nothing application.*

Performance Issues: CLX Versus VCL

A key concern for any developer who carefully deposits his eggs into the CLX basket is going to be what sort of performance he or she will get from using CLX. It's surely inevitable that there will be *some* hit, because a Delphi 6 or Kylix

application is essentially using two different application frameworks in tandem, CLX sat on top of Qt.

Or is it inevitable? As I've pointed out in the past, Qt is a highly optimised, high performance class library, and Trolltech has pointed out to me a number of cases where using its code gives superior performance to what you'd get by calling the equivalent Microsoft API routines. In order to experiment with relative performance figures for a VCL versus CLX application, I devised a simplistic testbed around the code shown in Listing 1.

Yes, I realise that testing the speed of drawing rectangles on screen isn't exactly a thorough test of the performance of an application framework! But even so, I thought it'd be interesting to compare the speed of drawing operations that go through the VCL → API route versus the same thing going through the CLX → Qt → API route.

In case you're not familiar with these particular timing routines, a few words of explanation are in order. You're probably familiar with routines such as `Time` and `Now` (from the `SYSUTILS` unit) and experienced Windows API veterans will also have come across the `GetTickCount` routine which you can find in `WINDOWS.PAS`. However, if you're after really high resolution timing, of the order of microseconds, then `QueryPerformanceCounter` and `QueryPerformanceFrequency` are the API routines of choice. Each time you call `QueryPerformanceCounter`, the single `var` parameter will retrieve a 64-bit integer which corresponds to the current tick count, not to be confused with the ticks retrieved from `GetTickCount`: this is a much faster tick!

The main wrinkle here is that the actual tick rate can vary from one PC to another: it is a function of the hardware-based timer, which is actually being monitored by the `QueryPerformanceCounter` routine. You need to call `QueryPerformanceFrequency`. In order to determine the tick rate. This retrieves another 64-bit integer which tells us how many ticks are occurring every second. On my venerable 500MHz Pentium III, this returns a value of 3,.579,545. If it is to be believed, this means that the high resolution timer is running a little over 3.5MHz, giving us a tick resolution of around 279 nanoseconds.

Sure enough, this timer is so fast that if you call `QueryPerformanceCounter`, and then call the same routine again, you'll find that the returned counter value differs from the first by just a few ticks, around 7 or 8 on the aforementioned Pentium III. This means that there's an API-calling overhead of around 2.2 microseconds on my machine, running Windows 2000 Pro: this certainly sounds as if it's in the right ballpark. Thus, to convert a time delay measured in ticks into seconds, it's just a matter of dividing by the value returned from `QueryPerformanceFrequency`.

The code above simply retrieves the starting tick count, draws 10,000 filled rectangles onto the screen and then obtains the finishing tick value to calculate the actual delay that's taken place. As you can see from the code, each rectangle receives a random size, position and colour. In the case of the CLX code, WINDOWS.PAS naturally isn't included, so I copy/pasted the RGB function into my code, together with the function declarations for `QueryPerformanceCounter` and `QueryPerformanceFrequency`.

Well, what about results? This program yielded figures for the CLX code of around 1.15 seconds,

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  I: Integer;
  R: TRect;
  T1, T2, Freq: Int64;
begin
  Randomize;
  QueryPerformanceCounter (T1);
  QueryPerformanceFrequency (Freq);
  for I := 0 to 10000 do begin
    R := Rect (Random (Width), Random (Height), Random (Width), Random (Height));
    PaintBox1.Canvas.Brush.Color := RGB (Random (256), Random (256),
      Random (256));
    PaintBox1.Canvas.FillRect(R);
  end;
  QueryPerformanceCounter (T2);
  Caption := FloatToStr ((T2 - T1) / Freq);
end;
```

➤ *Listing 1*

Qt 3.0: What Does The Future Hold?

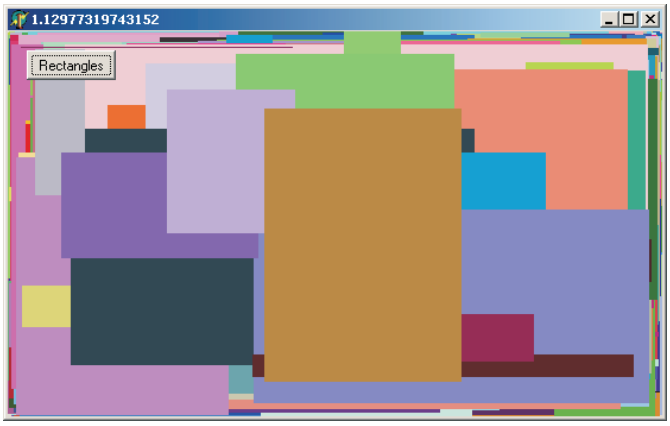
It's interesting to speculate on exactly what sort of licensing deal Borland has with TrollTech. Is there a provision for Borland to take and use later versions of Qt in Kylix? I only mention this because, at the time of writing, Qt 3.0 has recently been released to beta testers and from where I'm sitting it looks very tasty indeed.

Qt 3.0 includes a number of fascinating new features including a platform-independent API for working with SQL databases. This database-independent API is based around a driver model, and includes drivers for Oracle, MySQL and others. Naturally, new custom drivers can be added. The new database API has been put to good use in a new set of data-aware controls that perform automatic each-way synchronisation between the database and the user-interface.

The Qt Designer is rapidly evolving into a full-blown RAD design tool in its own right, which may well set it on a collision course with C++Builder and Delphi/Kylix in the future. It now supports custom widgets, direct editing of your C++ source code and also has support for the new data-aware controls.

Other Qt 3.0 goodies include multiple monitor support (including Xinerama under Linux), a new internationalisation tool which is Unicode 3 compatible, regular expressions, support for new 64-bit hardware, direct editing of rich-text and also support for new docking/floating windows.

It'll be interesting to see how many of these features make their way into future versions of Kylix. Certainly, the docking window support comes just in time to save Borland R&D's bacon as they remove Wine/Winelib from the Kylix 2.0 IDE implementation.



➤ *Figure 4: My test program demonstrates that the CLX/Qt combination can often be as fast as, and sometimes faster than, the VCL.*

compared to 0.14 seconds for the equivalent VCL program. In both cases, I ran the program many times, and took an average figure for the timings. Occasionally, the program threw up a value which was considerably longer than the average, and it's reasonable to attribute such anomalous figures to a context switch to another process. Matt Pietrek spent some time running performance comparisons of ANSI versus Unicode API routines (see *MSDN: Under the Hood, Periodicals 1997*) and he rightly points out that the best way of avoiding this sort of issue is to set your thread's priority to the maximum possible value like this:

```
SetThreadPriority(hMyThread,
  Thread_Priority_Time_Critical);
```

Such a thread will consume CPU time until it voluntarily yields control to another thread or process, thereby eliminating the anomalous timings mentioned above.

So CLX programs are guaranteed to execute more slowly than the equivalent VCL code, right? Wrong. Just for the hell of it, I replaced the above call to `FillRect` with this:

```
PaintBox1.Canvas.Ellipse(R);
```

In other words, draw 10,000 filled ellipses rather than rectangles. Since drawing an ellipse is more computationally intensive than drawing a rectangle, you'd naturally expect the VCL code to be slower, and the CLX code to be slower still. Surprisingly, I got a figure of 5.8 seconds for the VCL code and only 5.4 seconds for the CLX/Qt code. Hmmm...

Surprising though these figures may be, they do bear out Trolltech's assertion that its code can frequently provide superior performance to Microsoft's own routines, another good example being the way in which it applies various transformations to vector-based text characters. I'm sure you're familiar with the estate agent's motto: location, location, location! In the realms of software development, I suspect the equivalent would be algorithms, algorithms, algorithms! Ultimately, it's not the programming language you use that counts, it's the algorithms you use which make the biggest difference to the performance of your code, and although this is by no means an exhaustive analysis, I can only conclude that in real-world applications, you're unlikely to see any significant performance hit from using CLX/Qt. Indeed, you might find just the reverse.

I don't foresee performance as being a problem (at least, not on the Windows platform) when creating new CLX applications, or porting your existing code to CLX. A more significant problem is likely to be the breadth of functionality that's available to CLX applications. As I've already pointed out, docking windows are conspicuous by their absence, although this might change for the better if Borland are able to use Qt 3.0 in the next major release of Kylix, see the *Qt 3.0: What Does The Future Hold?* sidebar.

As Brian mentioned, Borland have enhanced the conditional compilation capabilities of the compiler, and this will certainly be a big help when writing code that's designed to be portable between Windows and Linux. You should also be aware of the new `platform` directive (similar to the `deprecated` and `library` directives). This is used to indicate that a particular declaration is platform-specific, thus:

```
var
  // Win2000, WinME, WinXP
  LayeredWindows: Boolean
    platform;
```

As with `deprecated` and `library`, this directive has no effect unless you actually refer to an identifier which has been so marked. If you do, the compiler will spit out an appropriate warning to indicate that you are doing something platform-specific.

Conclusions

In recent discussions on CIX, a number of folks have been poking fun at The Gimp, which is generally regarded (by Linux users) as something of a flagship application. From the perspective of a Photoshop user, however, The Gimp isn't a serious tool because of its lack of support for CMYK, and other, even more serious limitations, such as its inability to deal with decent-sized bitmaps.

And the relevance of that comment? There's no doubt that the Delphi 6/Kylix combination constitutes a great way of getting your applications running on a new platform. What's less certain is the current level of user demand for new Linux applications. I sincerely hope that Kylix succeeds and that it's used to create a whole new breed of killer Linux applications that give folks a compelling reason for moving to Linux. I'd like to see Linux regarded as a viable platform for serious desktop applications, and I'd like to see Kylix developers creating those applications. Time will tell...

Dave Jewell is the Technical Editor of *The Delphi Magazine*: contact him at TechEditor@itecuk.com