Beating The System: API-Level Programming With Kylix

We're not in Kansas any more...

by Dave Jewell

This month's *Beating The System* represents something of a milestone (at least, for me!) since it's the first article that I've written entirely from within Linux. I don't want to be accused of banging the Kylix drum too much, and Windows developers will be pleased to know that normal service will be resumed next month!

Nevertheless, the primary aim of this month's article is to 'widen your vision' in terms of what Kylix can do for you. I want to demonstrate that Kylix will do a lot more than simply create CLX-based applications and that it's equally well suited to writing efficient, easily deployable command-line utilities, barefoot X applications, or whatever. There's a whole new universe of APIs to explore here, so get stuck in!

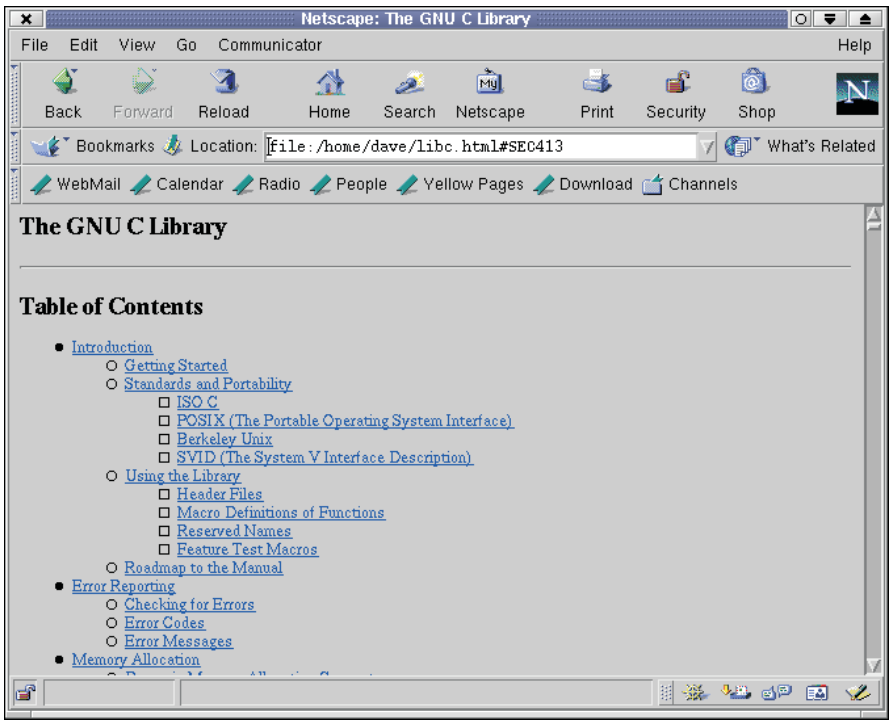
An Architectural Review

At this point, a brief review of the Kylix architecture is in order. As you'll no doubt appreciate, the familiar VCL application framework is specific to Delphi. When using Kylix, the VCL classes are replaced by the equivalent CLX classes and controls, which are also available in Delphi 6. CLX is a relatively thin layer whose main job is to try and provide as much VCL compatibility as possible; the real work is done by the Qt class library upon which CLX sits (if you want to know more, turn back some pages and read Brian Long's in-depth article on CLX and Qt).

What this means is that a Kylix application is arguably running with two application frameworks stacked one on top of another: CLX on Qt. Most of the time, this isn't a problem, but once you start using Kylix in anger, you'll certainly discover a few rough edges which show up as a result of some disagreement between the two class libraries.

For example, if you drop two panels and a vertical splitter control onto a form, set up the various `Align` properties such that the splitter is sandwiched neatly between the two panels and allows their relative widths to be resized, everything will work fine when you run the application. But if you now add a menu bar to the form window, you will find that things go unpleasantly pear-shaped whenever the splitter is resized. Rather than putting the vertical sizing outline where it should, the outline is displaced upwards by an amount equal to the height of the menu bar. Thus, it leaves sizing 'trails' all over the menu bar, which don't disappear until the form is redrawn. This is not a show-stopping bug, but it's untidy and a good illustration of what can happen when frameworks collide.

It's important to bear in mind that Qt is a very powerful application framework in its own right, forming the architectural foundation of the excellent KDE desktop. If you want to see more Qt source code than you can swing a cat at, go to www.kde.org/anoncvs.html, follow the instructions there, and download the latest KDE development snapshot via anonymous CVS. As the website warns you, you'll definitely be on the 'bleeding edge' if you try to compile and install any of this stuff, but if you want to play with the upcoming KDE 2.2, get some kudos by fixing a few bugs and, most importantly,



➤ *Figure 1: If you want to do serious work with libc, download a copy of the library documentation from the various places it can be found on the internet. This particular version is in the form of a single HTML file.*

gain a thorough understanding of Qt, this is the place to come!

If you're feeling adventurous you might even want to launch out into the unknown by programming Qt directly from Kylix. For an in-depth discussion of how the Qt class library is 'flattened' under Kylix, and why it's an absolute pain to try and communicate with Qt directly (at least from Kylix), again see the excellent article by Brian Long.

Here, however, we're going to dig somewhat deeper. Qt sits on top of the X system, and also makes use of other, lower-level, code which I'll refer to collectively as the Linux API. At the very lowest level, you've got the kernel itself. The kernel doesn't 'export' callable routines in the usual sense, but like the MSDOS kernel of fond memory (or even the NT kernel for that matter), you communicate with it using interrupts. The plain-vanilla MSDOS service interrupt was `INT $21`. Under Linux, the interrupt number used to communicate with the kernel is `$80`. I suppose that there's nothing to stop you writing a chunk of inline assembler code to set up the necessary registers and issue a kernel call, but doing so would be messy, error prone, and not portable. For those rare cases when you must issue a kernel call, take a look at the `syscall` routine in the LIBC.PAS file.

Introducing libc AKA glibc

You'll notice that `syscall` is implemented inside the libc.so.6 shared library, and it's actually the `libc` routines which we're going to focus on for a while here. They form perhaps the lowest level of the Linux API which is easily accessible to Kylix developers. Although originally written for the benefit of C developers, we can call these routines directly from Kylix, thanks to the sterling efforts of Borland R&D who have 'Pascalised' the necessary function declarations (the aforementioned LIBC.PAS file runs to around 24,500 lines of code!).

But I'm getting ahead of myself. Firstly, it's important to understand exactly what `libc` is and how it developed. Back in the early days of C programming, the C runtime library was statically linked to an application, and this was as true under UNIX systems as it was under DOS. Microsoft introduced the concept of DLLs for Windows while Linux now has shared libraries, which amount to pretty much the same thing. This made it possible to use a standard `libc` file which was shared by all interested applications, greatly reducing the memory requirements when several programs are running (because there's only one copy of the library in memory) and also making the size of executable files much smaller.

Strictly speaking, what I'm referring to here as `libc` is actually `glibc2`, or version 2.0 of what started out as the GNU `libc` library. You can find more details on the development history at www.gnu.org/software/libc/. `libc` encapsulates much of the de facto Linux API, including support for threads, processes, sockets, and all the usual stuff you'd expect in a runtime library. The foregoing is necessarily something of an oversimplification, and in fact version 2.x of `glibc` is actually implemented as multiple shared libraries. To see what is inside the library, visit www.gnu.org/manual/glibc-2.0.6/libc.html where you can download library documentation in various formats.

Let's begin with something simple. One of the questions most frequently asked by Linux newbies is 'Does Linux support DLLs?' Once that's been answered, they then want to know what has happened to `LoadLibrary`, `GetProc- Address`, and `FreeLibrary`, the Windows API routines concerned with DLL access. Not surprisingly, these routines don't exist under Linux. Instead, they're replaced by a trio of `libc` routines that perform the same job: see Listing 1.

These function declarations are taken from the LIBC.PAS file. Don't be misled by the references to 'shared object handles' in the accompanying file comments. In Linux-speak, a 'shared object' in this context is simply a shared library or DLL while a shared object handle is merely a reference to the library: a module instance in Win32 terminology.

```
function dlopen (Filename: PChar; Flag: Integer): Pointer; cdecl;
function dlsym (Handle: Pointer; Symbol: PChar): Pointer;  cdecl;
function dlclose (Handle: Pointer): Integer;  cdecl;
```

➤ *Listing 1*

As you'd expect, the `dlopen` routine takes the name of a shared library and returns a pointer, the so-called shared object handle. If the return value is `Nil`, then the requested library wasn't found. Bear in mind that, under Linux, the algorithm used to search for shared libraries is completely different to the way in which `LoadLibrary` works under Windows, the most important thing being the environment variable `LD_LIBRARY_PATH`, which defines a list of directories where the search is performed. The `Flag` parameter to `dlopen` can take a number of values, the most useful of which are:

- `RTLD_LAZY`. This defers the resolution of undefined symbols (ie external symbols needed by the incoming dynamic library) until the library code executes.
- `RTLD_NOW`. This forces undefined symbols to be resolved before the call to `dlopen` returns.

Either of these two flags can be combined (using `OR`) with another flag, `RTLD_GLOBAL`, which causes any external symbols exported by the library to become available to other dynamic libraries which are opened via `dlopen`, thus explaining the real reason behind the first two flags. I won't go into more detail here, but if you're getting the impression that shared libraries under Linux are potentially more sophisticated than what is available under Windows, you would be right!

`dlclose` is obviously used to close the shared library whereas `dlsym` is the Linux equivalent of `GetProcAddress`. It takes a handle to the loaded library together with the name of the required function, returning the routine address.

As an example of how to use these routines, take a look at the code in Listing 2 which I've copied from SYSUTILS.PAS. This routine, CreateGUID, is in the business of generating a GUID. Under Windows, this call is simply mapped onto the routine of the same name in OLE32.DLL, which is obviously not an option under Linux. Instead, this code makes use of a shared library, libuuid.so.1. Unfortunately, it's not absolutely guaranteed that this library is present on every Linux distribution, and therefore the code uses dlopen, etc, to programmatically load the library and call the uuid_generate_time routine contained therein. If you're wondering why the library never gets closed, libuuidHandle is a global variable which gets passed to dlclose in the finalization clause of SYSUTILS.

If you don't need fine control over those dlopen flags, you can make use of a set of handy wrappers provided by Borland. Peek inside SYSUTILS.PAS and you will find LoadLibrary, FreeLibrary, GetProcAddress and even GetModuleHandle, all provided courtesy of some wrapper code inside the aforementioned DLL. If you're trying to write portable code, you'd be advised to stick with these routines unless your requirements are more specialised. Of course, if you're trying to create the smallest possible executable, then you can always pull these routines out of SYSUTILS altogether.

Fun With Processes

For those coming from a Windows background, some of the most interesting routines in libc are those that relate to process creation and management. Probably the simplest method of running another process is to make use of the system routine which is defined like this:

```
function system(Command:
  PChar): Integer; cdecl;
```

Since the identifier 'system' has a unique meaning as far as Object Pascal is concerned, the compiler won't appreciate you using this routine unless you qualify the identifier name, like this:

```
Libc.system('gimp');
```

In this example, the code fires up the Gimp image manipulation program, always assuming, of course, that you've got it installed. In order to do this, a search is made through the list of directories specified in the PATH environment variable, looking for an executable with a matching name.

The system call actually works by making use of the default shell. In other words, whatever you specify as an argument to this call, you can think of it as being 'typed' at the command line. Here's another example which illustrates the point:

```
Libc.system(
  'ls -l /usr/local/'+
  'kylix/lib/*.dcu >frodo');
```

In this example, we use the list command to create a verbose directory listing of all the DCU files located in Kylix's 'lib' directory. Of course, this assumes that you've installed Kylix into the same place as I have! You'll also notice that this example uses shell redirection to redirect the results to a text file called frodo. If you were nervous about using the directory lookup routines contained in libc, you could use this sort of technique to create a text file containing the required info, and then parse the resulting file afterwards using your own code *[Oooh, that's a blast from the past: I remember doing just this in good old DOS! Ed]*.

This technique is actually a great deal more useful that you might think, especially once you realise just what riches are contained inside the /proc directory! As seasoned Linux users will know, the /proc directory actually contains a set of what might be called 'pseudo-files'. They're not really 'there' in the sense of physical disk files. Rather, they correspond to a set of in-memory buffers which provide all sorts of interesting information about the system. This is directly analogous to the way in which Windows NT maintains a number of dynamic keys inside the system registry which correspond to certain on-the-fly aspects of the operating system's behaviour.

To see this in action, try using the system command like this:

```
Libc.system(
  'cat /proc/cpuinfo > ~/cpu');
```

This will create a text file called cpu in your home directory. Open this text file, and you'll discover the model name of your CPU, the stepping number, precise CPU speed and a hundred and one other anorakish things! The point here is that this sort of thing is trivially easy to discover under Linux, but could take page after page of obscure Win32 code.

Some of the entries under /proc are themselves directories. The screenshot in Figure 2 shows the results of peeking inside the /proc/bus/usb/devices file which lists all the connected USB devices. No, I don't understand half of the information presented here, and I realise that USB support under Linux is still in its infancy, but nevertheless you can

➤ *Listing 2*

```
var
  libuuidHandle: Pointer;
  uuid_generate_time: procedure (out Guid: TGUID) cdecl;
function CreateGUID(out Guid: TGUID): HResult;
const
  E_NOTIMPL = HRESULT($80004001);
begin
  Result := E_NOTIMPL;
  if libuuidHandle = nil then begin
    libuuidHandle := dlopen ('libuuid.so.1', RTLD_LAZY);
    if libuuidHandle = nil then Exit;
    uuid_generate_time := dlsym (libuuidHandle, 'uuid_generate_time');
    if @uuid_generate_time = nil then Exit;
  end;
  uuid_generate_time (Guid);
  Result := 0;
end;
```

see from this that I've got an Iomega ZIP CD plugged into a USB port. Again, how much effort would it take to figure that out under Windows?

There's one problem with the `system` command, which is that it effectively 'blocks' while the other process is executing. Taking our original example of running Gimp: your Kylix application will be frozen until Gimp is closed. Is there a way round this? With Linux, there's usually a way round anything! Once again, those familiar with the UNIX command line will know that if you append an ampersand character (`&`) to the end of the line, you'll get a 'fork'. Without the fork, the parent process is blocked until the child process dies, but with a fork, they can both continue executing together. Think of it as a fork in the road: afterwards, you've got two roads instead of one! If you try appending an ampersand to the Gimp example given earlier, you'll see that the original application no longer waits for Gimp to terminate.

UNIX programmers traditionally refer to this sort of scenario as a 'fork' because of a `libc` library routine of the same name. Whenever you start off with a single process and end up with two, this routine is generally involved somewhere. Listing 3 has a little example program for you.

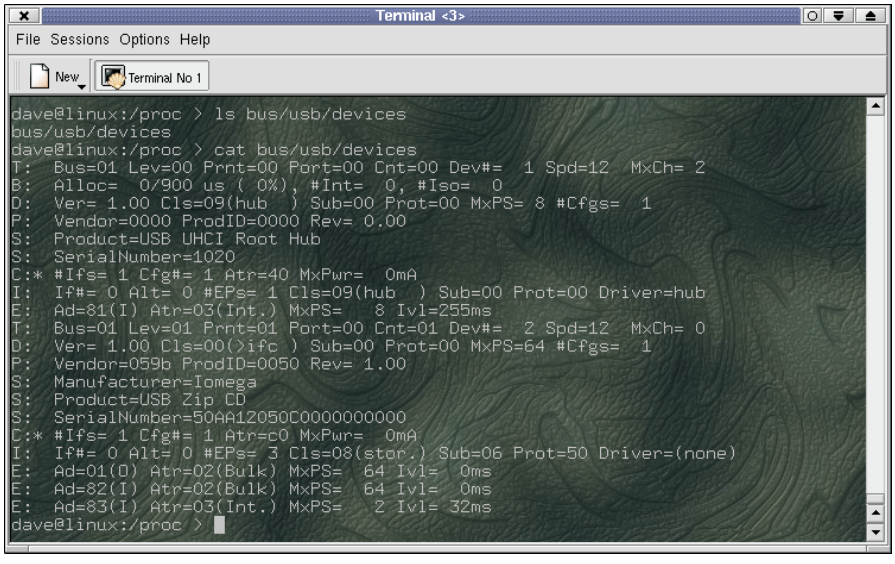
If you were to execute this program, not knowing what `fork` actually does, you might reasonably expect to see either `New Process` or `Old Process` printed on the console. In fact, you'll see both messages! Immediately before the call to `fork`, only one instance of your program is running. Immediately after the call to `fork`, there are two! It's crucially important to understand that the second, new, process doesn't start executing back at the beginning of the program code. Rather, a carbon copy of the process is created in memory, and it starts executing from the point immediately after the call to `fork`. Because the new process inherits a copy of all the global variables from the parent process, everything is set up the way it should be.

So how do we know which is the child and which is the parent? This should be obvious from the Listing 3 code. The return value of `fork` will be zero for the *newly created* process, and it will be non-zero when `fork` returns to the original process. If the return value is -1, this indicates an error: it wasn't possible to create the child process. Any other non-zero value is taken as being a `pid` or process identifier. Process identifiers are very important in Linux and you can use them to obtain information on other processes, wait for a specific process to terminate, kill another process (if you have the necessary privileges) and so on.

```
program xapp;
uses Libc;
begin
  if fork = 0 then
    Writeln('New Process')
  else
    Writeln('Old Process');
end.
```

➤ *Listing 3*

If you look back at the /proc directory, you'll see that there are a number of 'pseudo-directories' here, each of which has a number rather than a name. Here again, Linux makes it dead easy to enumerate all the running processes on a machine, because each of those numbered directories corresponds to a current process. Thus, if you've got a process with a `pid` of 37, you'll find that there's a pseudo-directory (this is probably not the right terminology, but you know what I mean) under the /proc directory with the name '37'. Within each of these directories, there are various other entries which enable you to determine the amount of memory used by the process, the command line that was used to start the process, and so forth. You can even enumerate the file handles currently in use by the process (hint: look in the `fd` subdirectory). Of course, Linux won't allow you to examine or fiddle with the attributes of a process which you don't own. In order to get the full picture you'll have to be running as root. Nevertheless, this technique of mapping the operating system state onto pseudo file system entries is a very powerful mechanism, and one is left wishing that life were as simple for Windows developers!



➤ *Figure 2: The /proc directory gives access to a host of system and process information. Don't make the mistake of thinking that this is an inefficient way of obtaining such info; many of the runtime library routines are simply wrappers around code that accesses the /proc directory!*

The Joy Of X

I'm not sure that I'd recommend that you go down this route unless you're a serious anorak, but Kylix will actually enable you to program the X Window system directly. Up until now, we've been talking about `libc`, which is encapsulated in the LIBC.PAS file. However, Borland also provides

another hefty header file in the shape of XLIB.PAS. Actually, this is nowhere near as enormous as `libc`, weighing in at a mere 7,000 lines of code.

As you'll be aware, a Kylix application needs a special shared library, `LIBQTINTFSO`, which acts as the glue connecting the Qt class library to a Kylix application. What if we were to dispense with Qt completely? Immediately we do this, we get a big reduction in the number of ancillary files that we need to deploy along with our application. For your amusement, here's how to create the smallest possible executable using Kylix: first, select `New` from the `File` menu and then choose a `Console` application. Of course, we don't actually *want* a console application, so you'll need to delete the line:

```
{$APPTYPE CONSOLE}
```

from the generated source code. If you now compile this do-nothing application, you'll get a tiny executable (oops, I nearly said EXE file!) that's only 14Kb to 15Kb in size. And remember, folks, that's without using packages. If you rebuild this application with packages, your executable will shrink to around 5Kb in size. However, this has the effect of introducing a dependency on `BPLBASECLX`, although I'm not sure why. Since hard disks are so cheap these days, there's not much point in introducing non-standard dependencies (`BPLBASECLX` isn't part of any standard Linux distro, at least, not yet) for the sake of 10Kb, so I'd advise you to stick with a non-packaged application.

And just in case: please don't berate me for using uppercase letters when referring to these filenames: I appreciate that under Linux filenames are case-sensitive, but it just helps me to make things stand out in this simple-minded Linux text editor I've been using here!

➤ *Figure 3: Here's our little do-nothing X application doing, err, nothing. The essential point, though, is that Kylix makes it possible to create applications which are tiny, easily deployable, and require no special runtime support other than the standard shared libraries that ship with any distro.*

➤ *Listing 4*

```
program xapp;
uses
  Xlib, Libc;
procedure Main;
var
  dpy: PDisplay;
  w: TWindow;
begin
  dpy := XOpenDisplay(Nil);
  w := XCreateWindow(dpy,
    XDefaultRootWindow (dpy), 0, 0,
    200, 100, 0, CopyFromParent,
    CopyFromParent, Nil, 0, Nil);
  XMapWindow(dpy, w);
  XFlush(dpy);
  __sleep (10);
end;
begin
  Main;
end.
```

Ok, this might be the smallest possible Kylix executable, but it's not a windowed executable, is it? How can we add a bare-bones window to this program while keeping the code tiny? Once you add the `Xlib` unit to your `uses` clause, you are all ready to start experimenting with X! Take a look at Listing 4, which shows you how to do it.

I shamelessly adapted this source code from a C-based X tutorial which I found on the internet at http://tronche.com/gui/x. The program begins by specifying `Xlib` and `Libc` in the `uses` clause. `Libc` is only required because we make use of one routine from this unit, `__sleep`, about which more later.

The first job of the program code is to call `XOpenDisplay` which opens a connection to the X server. As you might know, X is based around a client-server model. No, it's nothing to do with databases, I'm delighted to say!

The basic idea is that the display device could potentially be a dumb terminal which is some distance away from the server, with something like a TCP/IP connection between them. Most of the time, of course, you'll be working with an X server which is sat on the same machine as the client software, but we will shortly see how this client-server architecture is evident in the `Xlib` API. When you use the `Xlib` routines, you are effectively calling code in a large shared library which (at least on my machine) is called libX11.so.6.2. The main job of this library is to convert client requests into something called the X Protocol, passing these requests on to the server itself.

In keeping with this client-server approach, the one and only parameter to `XOpenDisplay` is a string, which specifies the host machine to which the display is attached and the server and screen number on that machine. Fortunately, we don't have to get bogged down with all that stuff. For our purposes, we can just pass `Nil`, which causes `Xlib` to use a string value obtained from the `DISPLAY` environment variable. This will typically have the value `:0.0` which simply means, this machine, first display server, first screen!

Having got a reference to the X display, the program next calls `XCreateWindow` to create an actual window. I won't discuss the purpose of all these parameters (you can read up on it by working through the freely downloadable X reference material) but briefly, the first parameter to this routine represents the connection to the X server which we've just established. The second parameter allows us to specify a parent window for the window being created. Because we're creating a top-level application window, we

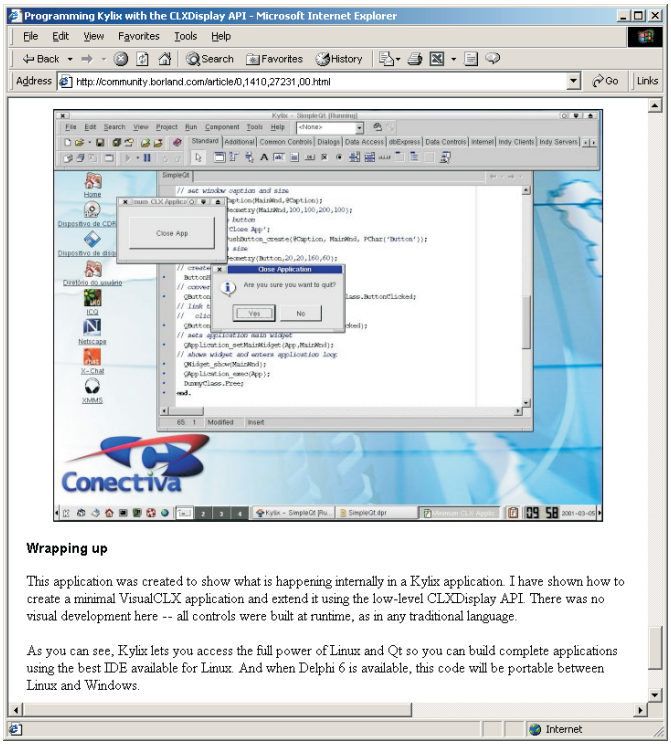
can get by with a call to `XDefaultRootWindow`, which gives us the root window for the default screen. Presumably, this corresponds to the desktop under Linux, but don't quote me! The next four parameters specify the X, Y, width and height positions of the window. Where the window actually appears is to some extent a function of the installed window manager, and won't necessarily correspond to the X, Y values that you supply here.

At this point, we've got a window, but it isn't visible yet. To make it visible, we need to call `XMapWindow` which causes the window to appear on the actual display. Well, almost. If you were to remove the call to `XFlush`, you'd never see the window. I mentioned above that X uses a client-server model, and in the interests of efficiency, the client code (`Xlib`) buffers up a number of calls to the server before sending them together. `XFlush` causes this output buffer to be flushed, yielding an immediate response from the server. In the normal course of events (pun strictly intentional!) a 'proper' X application would sit in a message loop, processing events as they occur, and the X routines used here guarantee that `XFlush` is called as needed. However, this little application cheats by dispensing with an event loop, and that's why the call to `XFlush` is required.

And, of course, this is also the reason we need to call `__sleep`. Since there is no message handling in this minimalist application, the window would normally appear and disappear in the blink of an eye, but the `__sleep` call ensures that it sticks around for ten seconds.

Experienced `libc` developers should note that this is actually the standard `libc sleep` function, renamed by Borland so as not to clash with the `Sleep` routine in SYSUTILS.PAS. Personally, I don't see why they had to rename it: after all, Object Pascal gives us the ability to disambiguate identical names in different units by prefixing the identifier with the name of the unit. I think that `Libc.sleep` would have been clearer, but then again I'm biased: to me, one or more underscores before an identifier name is ugly.

The result of all this can be seen in Figure 3. You are probably wondering where that nice textured background came from, and how come there's a Kylix icon sitting in the window? The reason, of course, is that the application doesn't attempt to render any sort of window content, and for that reason, whatever background was behind the window at the time it was created is what gets mapped into the window.



➤ *Figure 4: Check out the http://community.borland.com site where you'll find an article entitled 'Programming Kylix with the CLXDisplay API'. Once again, this is an excellent eye-opener piece in terms of what Kylix can do.*

Conclusions

`Xlib` has a reputation for being a difficult-to-program, cumbersome API, and although there are some weak areas, I'd say it's not much more difficult to program than the barefoot Windows API. In any event, I hope that this article will inspire you to delve deeper into both X and the powerful `libc` library. And if you're feeling sufficiently inspired to rewrite Qt in Object Pascal, then I'd love to hear from you!

In the last couple of weeks, I've seen some people comment that Kylix isn't yet sufficiently mature for desktop application programming (lack of third-party support, slow IDE, bugs in CLX, etc) and that it's more suited for server development right now. Whether or not such criticisms are valid, I think they miss the essential point that Kylix gives us a native-code, high performance compiler with which to explore the world of Linux and do it using the programming language we all know and love.

Next Time

Next month, we'll be back in Windows land with a vengeance when I'll be taking a look under the hood of Windows XP, (alias Whistler) and examining some of the new goodies contained therein from a developer's perspective. See you then.

Dave is a freelance consultant, programmer and technical journalist specialising in system-level Windows programming and cross-platform issues. He is the Technical Editor of *The Delphi Magazine*. You can contact Dave at TechEditor@itecuk.com