How CLX Uses Qt

by Brian Long

Kylix has been here since March and Delphi 6 is announced and as you read this should be very close to being shipped. This means Borland's CLX library (pronounced *clicks*) is now a truly cross-platform component library (CLX is a fabricated acronym for Component Library for Cross-platform, where the word Cross is abbreviated to an X). CLX code can be compiled for Linux in Kylix or for Windows in Delphi 6, whilst those who are not interested in cross-platform development can of course continue using the VCL, pronounced *vee-see-ell*.

This article looks under the hood of the CLX library to see how it uses Trolltech's cross-platform Qt C++ class library (pronounced *cute*). It starts by looking at what Qt is and why it was chosen as the basis for the CLX library, and then moves on to see how it has been integrated with the Object Pascal language in Kylix and Delphi. An understanding of how Qt is used will be helpful when writing custom components, and also when you need to call Qt directly when a component does not offer all the facilities you require.

Since a CLX application involves talking to C++ classes, there is a certain amount of complexity involved to get it working. Whilst I have tried to take things a step at a time, with each step being as small as possible, it may be necessary to re-read sections of this article a number of times until things become clear.

Alternatively, if things get too confusing, read the article through to the end and then start at the beginning again.

More information can be obtained on a variety of topics touched on throughout this article by following the references in the *Further Reading* section at the end.

The Qt Library

Most of us are quite used to the Windows operating system (OS). We are used to the fact that the OS is GUI-based and we are used to the fact that it comes equipped with quite a few built-in controls, such as buttons, listviews, edits, toolbars and treeviews.

Linux is not the same at all. The Linux OS is really a text mode system. The GUI that might be associated with Linux (X Windows) is really just an application running on Linux that implements a GUI. In fact, whilst X has certain functionality built-in (like being able to run on one machine, whilst displaying programs running on another), it does not have any built-in controls. All it knows about are plain windows and some basic keyboard and mouse-based events.

What Is Qt?

In order for applications to present familiar controls to users, the programmer must either implement them or use libraries of pre-built controls. Such control libraries are called *widget sets*, where each control is a *widget*. According to the book *X Window Programming from Scratch* a widget set is *'a group of components that manage different aspects of a graphical user interface. Elements such as menus, buttons, scrollbars, and text fields are entities provided by a widget set.'* There are a variety of these widget sets available, accommodating various requirements, and Qt is one of them.

➤ *Figure 1: The Qt logo.*

Qt has been available since 1995 from the Norwegian development company Trolltech AS (Figure 1 shows the product logo). In the company's own words: *'Qt is a cross-platform C++ GUI application framework. It provides application developers with all the functionality needed to build state-of-the-art graphical user interfaces. Qt is fully object- oriented, easily extensible, and allows true component programming.'*

So Qt is an X widget set, but which also works on other operating systems, notably Microsoft Windows. Qt is a popular widget set, and is used in the development of the KDE desktop environment (note that the desktop environment is another bolt-on option in X, unlike in MS Windows, and there are a variety to choose from, including the popular KDE and GNOME desktops). The GNOME desktop environment, on the other hand, is built using the GTK widget set.

The different choices made by the desktop environment developers do not affect any other applications. An application built with the GTK widget set will happily run in KDE and a Qt application will run perfectly well in GNOME (as long as the appropriate widget set libraries are available).

Why Qt?

When the VCL was written it was designed as an ObjectPascal class library, which included wrappers for all the existing Windows controls, and a number of other utility classes (for such things as database access). The `TButton` class wrapper, for example, does not implement button functionality at all. That is left to the button control in Windows, implemented by Microsoft, which does the job perfectly well.

When planning the development of Delphi for the Linux platform, Borland looked at the VCL and decided it was too Windows-dependent to massage into working on Linux. Instead, it

decided to keep the VCL for just the Windows platforms and create a new cross-platform library, which was ultimately called CLX.

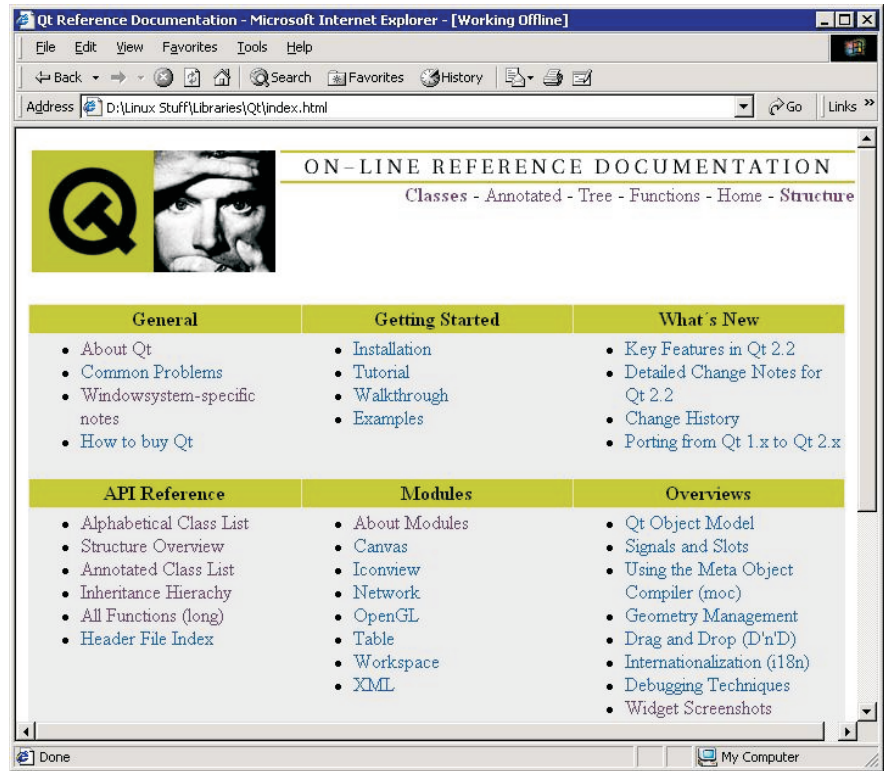
CLX was planned to be suitable for both Linux and Windows, and potentially other platforms in the future, but firstly needed to work on Linux. For this to work, Borland was obliged to choose a widget set to use as a basis. Either that, or it would have to fully implement all the controls from scratch on Linux, which is not a task to be taken lightly. The whole point of widget sets being available is that it saves you doing the job of implementing lots of controls yourself.

The developers looked carefully at a number of widget sets, primarily GTK and Qt, to see what advantages and disadvantages each set held for the job. After much examination, it turned out that:

➢ Qt is closer to the Windows style of development than GTK.
➢ The Qt graphics model is closer to VCL graphics model.
➢ Qt classes look very much like VCL components, with get/set method pairs being used where the VCL uses properties (eg `caption/setCaption` in the button widget).
➢ The Qt object model is very similar to the VCL object model (there is a `QApplication` object which handles the message/event loop, as well as `QButton`, `QListBox`, `QComboBox` and so on).
➢ The Qt event model is quite similar to the VCL event model, although it supports multiplexing (the equivalent of having multiple event handlers responding to one event).

Ultimately, they felt that using Qt would be quicker than adapting GTK, and so CLX is built on top of Qt. Or, to be more precise, the visual portion of CLX, VisualCLX, is built on top of Qt. The other areas of CLX do not require Qt.

During the development of CLX, Borland worked with the most up-to-date version of Qt and found the odd problem here and there. The Kylix 1.0 CD ships with a patched version of Qt version 2.2.4, current at the time, and includes the source code patches in the /patches/qt directory.



➤ *Figure 2: The comprehensive Qt online documentation.*

Where Do I Find Out About Qt?

To find out about the Qt class library you can read the documentation at http://doc.troll.no. This HTML-based documentation is very comprehensive and warrants quite some browsing (see Figure 2). You can have this documentation locally by downloading the Qt Free Edition from www.trolltech.com/dl/qtfree-dl.html. The download primarily includes the version of the Qt library needed for building free and Open Source C++ applications, which is of course irrelevant for Kylix developers using ObjectPascal.

You can also read about Qt programming concepts and principles in a couple of books on the subject (see the *Further Reading* section).

The Qt Interface Library

As I mentioned, Qt is a C++ class library. The Linux version of Qt 2.2.4 (libqt.so.2.2.4) weighs in at 6.59Mb. A Kylix CLX application does not link directly to this library as there are inherent problems trying to call methods of C++ classes from ObjectPascal. The problems include differences between the layouts of the classes and their vtables, constructs which do not translate such as C++ templates, and also C++'s *name mangling* process (also referred to as *name decoration*), which perhaps warrants some further discussion.

Since any C++ method can be overloaded (redefined with the same name but different argument lists), the compiler uniquely identifies each method by adding characters to its name to specify the type of each argument, the calling convention and so on. This makes the real, compiled, name of any method most unpleasant and would make talking to Qt quite an illegible process.

There is also the problem of the C++ language specification not defining how name mangling takes place, leaving each C++ compiler free to make up its own rules. In short, it is a problem talking to exported C++ class methods from ObjectPascal.

To overcome this problem Borland supplies a Qt interface library (the Linux version is called libqtintf.so.2.2.4 and is 1.5Mb). Since ObjectPascal does not

support exporting of classes (only standalone functions and variables), this interface library does not export ObjectPascal classes that hide the nastiness of calling the C++ classes. Instead, it takes another approach. Each individual member function of each Qt C++ class deemed interesting is individually exported from the library as a standalone function.

What Are Flat Methods?

To understand how this can be achieved relies on knowing how a method call differs from a function call. When you call a function, every argument expected by the function is formally declared in the function declaration. When the function is called, each argument is passed to the function in an order defined by the function's calling convention.

When you call a method through an object reference, much the same thing happens with one difference. An extra (hidden) parameter is passed into the routine. This hidden parameter is called `Self` in ObjectPascal (and `this` in C++) and gets its value from the object reference through which the method was called (the parameter itself is accessible, only the declaration is hidden).

This is done so that any references to class data fields are resolved correctly. Remember that each instance of a class (each object) gets its own copy of the data defined in the class. When a method executes, any references to data fields must access the data field for the correct instance. To accomplish this, the method code executes in the scope of `Self`, as if in a statement like:

```
with Self do
  //method code
```

So, assuming you can get an appropriate function/procedure declaration, you can call any method as if it were a normal routine simply by passing an object reference as the first parameter.

This might be a little clearer if we look at an example where a Delphi method is accessed just like a normal routine. To do this will require a procedural variable (in essence this is a function pointer) whose type matches the method, but with one additional parameter. The variable will be assigned the method's address and is intended to be a representation of one of the routines exported from the Qt interface library.

First, however, let's see the method being called normally (see Listing 1). Nothing particularly unusual to report here. Listing 2 shows the object still being constructed using normal Object Pascal code, but has the method being called through the procedural variable. Note the extra `TFoo` parameter in the argument list, which becomes `Self` in the method implementation.

This can be taken a step further. The call to the constructor can also be made through a function pointer (so long as you know any details of hidden arguments that may be required), as can the destructor if needed.

In the case of Delphi objects, we normally avoid calling the destructor directly, and call the `Free` method instead (which checks that the object reference is

➤ Listing 1: Calling a method in the normal way.

```
type
  TFoo = class
  public
    procedure ShowInfo(const Info: String);
  end;
procedure TFoo.ShowInfo(const Info: String);
begin
  ShowMessageFmt('%s: %s', [ClassName, Info])
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  Foo: TFoo;
begin
  Foo := TFoo.Create;
  Foo.ShowInfo('Called through a normal method');
  Foo.Free;
end;
```

➤ Listing 2: Calling a method as if it were a normal routine.

```
var
  TFoo_ShowInfo: procedure (Handle: TFoo; const Info: String);
...
  TFoo_ShowInfo := @TFoo.ShowInfo;
...
procedure TForm1.Button2Click(Sender: TObject);
var
  Foo: TFoo;
begin
  Foo := TFoo.Create;
  TFoo_ShowInfo(Foo, 'Called through a function pointer');
  Foo.Free;
end;
```

➤ Listing 3: Calling a constructor, method and destructor as normal routines.

```
var
  TFoo_Create: function (ClassRef: TClass; Construct: Boolean): TFoo;
  TFoo_Free: procedure (Handle: TFoo);
  TFoo_Destroy: procedure (Handle: TFoo; Destroy: Boolean);
  TFoo_ShowInfo: procedure (Handle: TFoo; const Info: String);
...
  TFoo_Create := @TFoo.Create;
  TFoo_ShowInfo := @TFoo.ShowInfo;
  TFoo_Free := @TFoo.Free;
  TFoo_Destroy := @TFoo.Destroy;
...
procedure TForm1.Button3Click(Sender: TObject);
var
  Foo: TFoo;
begin
  Foo := TFoo_Create(TFoo, True);
  TFoo_ShowInfo(Foo, 'Called through a function pointer');
  //TFoo_Free(Foo);
  TFoo_Destroy(Foo, True)
end;
```

non-nil before calling the destructor for you), but the choice is there. Listing 3 shows the birth, life and death of a Delphi object controlled solely through function pointers. These three listings have all been taken from the sample QtCopy.dpr project which is included on this month's disk.

The Qt interface library makes function pointers like these available for every method. The process of creating the source for this library was doubtless automated in some way and the library ends up exporting a massive number of routines (over 3,300 of them). Classes exported in this non-object fashion are sometimes called *flattened classes*, so from now on I'll refer to any non-object representation of a method as a *flat method*.

The CLXDisplay API

According to the DEPLOY file installed with Kylix, the import unit for the Qt interface library is formally known as the CLXDisplay API. Granted, it provides the API for VisualCLX components, as they are based on Qt classes made accessible through the unit, but I have not seen the term used anywhere else yet.

The import unit is called Qt.pas and in Kylix 1.0 is 8,821 lines long. It has a *lot* of stuff in it. It starts with a few basic types such as PInteger, PByte, TPointArray, TIntArray, HANDLE and HBITMAP and then goes into the Qt stuff proper.

Scattered around you find a number of enumerated types defined, which are required by various Qt methods, and here we immediately see something new in the language. In C, the individual values of an enumerated type can be initialised with specified values, since they are all treated as integer constants.

ObjectPascal has not supported this ability before, because the language is more strongly typed than C. Each value in an enumerated type is not considered an integer, but a value from the specified type. Without a suitable typecast an enumerated type value and an integer are not interchangeable, although internally they are stored as integers with incrementing values starting at 0.

Now ObjectPascal has been extended to support this initialised value concept in order to allow the types to be mapped across easily. Listing 4 shows an example from Qt.pas. For more information, look up *enumerated types* in the help and read the section entitled *Enumerated types with explicitly assigned ordinality*.

In truth, according to the new rules of the language, Listing 4 has more in it than it actually requires to get the same outcome. Listing 5 shows a slightly less explicit listing that has the same effect.

Handle Classes

Another thing that strikes you as you browse through the unit is that, despite my assertion that all the classes are exported as collections of flat methods, there are still a number of classes defined (some of them are shown in Listing 6). Whilst all the Qt methods are indeed accessed as flat methods, almost all of them still need object pointers passed to them, which are maintained in the ObjectPascal code. Remember that the Qt objects are C++ objects, not Kylix or Delphi objects, so there are no pre-defined Object Pascal types for the classes.

Rather than just use raw pointer types to keep track of these objects, Borland R&D decided to keep a certain level of type strictness with regard to these object pointers and so defined a number of stub classes to mirror the class hierarchy in Qt itself. A reference to a Qt QButton object is typically stored in a QButtonH Object Pascal object reference. Note that from within Object Pascal, this object reference does not give direct access to the QButton methods, but you can pass a QButtonH object reference (or anything inherited from QButtonH, such as QCheckBoxH, QPushButtonH or QRadioButtonH) as the extra first parameter to any QButton flat method.

Because you cannot call the Qt methods directly through these

➤ *Listing 4: An enumerated type with initialised values.*

```
type
  QSliderTickSetting = (
    QSliderTickSetting_NoMarks = 0 { $0 },
    QSliderTickSetting_Above = 1 { $1 },
    QSliderTickSetting_Left = 1 { $1 },
    QSliderTickSetting_Below = 2 { $2 },
    QSliderTickSetting_Right = 2 { $2 },
    QSliderTickSetting_Both = 3 { $3 });
```

➤ *Listing 5: A more concise version of Listing 4.*

```
type
  QSliderTickSetting = (
    QSliderTickSetting_NoMarks, //1st value will be 0
    QSliderTickSetting_Above,   //this will be 1
    QSliderTickSetting_Left = 1 { $1 },
    QSliderTickSetting_Below,   //this will be 2
    QSliderTickSetting_Right = 2 { $2 },
    QSliderTickSetting_Both);   //this will be 3
```

➤ *Listing 6: Part of the ObjectPascal Qt handle hierarchy.*

```
type
  QtH = class(TObject) end;
    QObjectH = class(QtH) end;
      QApplicationH = class(QObjectH) end;
        QClxApplicationH = class(QApplicationH) end;
      QWidgetH = class(QObjectH) end;
        QOpenWidgetH = class(QWidgetH);
        QButtonH = class(QWidgetH) end;
          QPushButtonH = class(QButtonH) end;
            QClxBitBtnH = class(QPushButtonH) end;
        QComboBoxH = class(QWidgetH) end;
          QOpenComboBoxH = class(QComboBoxH) end;
      QFrameH = class(QWidgetH) end;
        QTableViewH = class(QFrameH) end;
          QMultiLineEditH = class(QTableViewH) end;
```

object references, they are sometimes called *opaque references*. They do point to the Qt C++ object, but do not let you see its methods. Incidentally, since these opaque references are declared as Object Pascal classes for the benefit of type strictness, you must resist any temptations to call `TObject` methods through them, such as `ClassName`, which would typically give you an Access Violation for your trouble.

You will notice the trailing `H` at the end of each class name in Listing 6, which stands for *handle*. Generally, a handle (sometimes historically called a *magic cookie*) is some number which uniquely identifies an instance of something (for example, a file handle or window handle).

Any ObjectPascal object reference is stored as a pointer, which is interpreted as a memory address, but is just a number in reality. An ObjectPascal object reference declared as one of these methodless class types, which points to a Qt C++ object, is often referred to as a *Qt widget handle*, or sometimes just a *Qt handle*. So in this context, a Qt widget handle is an opaque reference to a Qt C++ widget.

➤ *Listing 7: Some of the flat methods of Qbutton.*

```
function QPushButton_create(parent: QWidgetH; name: PAnsiChar): QPushButtonH;
  overload; cdecl;
function QPushButton_create(text: PWideString; parent: QWidgetH; name:
  PAnsiChar): QPushButtonH; overload; cdecl;
procedure QPushButton_destroy(handle: QPushButtonH); cdecl;
procedure QPushButton_setGeometry(handle: QPushButtonH; x: Integer; y: Integer;
  w: Integer; h: Integer); overload; cdecl;
procedure QPushButton_setGeometry(handle: QPushButtonH; p1: PRect); overload;
  cdecl;
```

Qt Flat Methods

We saw some hand-crafted flat methods in Listing 2 and Listing 3, and now it's time to see some of the official Qt flat methods. Listing 7 shows some of the flat methods of the `QPushButton` class. You can see the constructor, which returns a `QPushButtonH` handle, whilst all the other methods (including the destructor) take a `QPushButtonH` as the first parameter.

The `TButton` VisualCLX class hides the need to call these flat methods most of the time. However, just as in VCL programs, where you sometimes need to call the Windows API when a VCL class does not cater for your every requirement, you may also need to call the CLXDisplay API from time to time when a VisualCLX class doesn't satisfy all your needs.

Listing 8 shows how you would create a `QPushButton` widget from scratch, with the form as its parent, call some of its methods (one of which happens to be inherited from `QButton` and one from `QWidget`) and then destroy it. You should see it looks much the same as the code shown in Listing 3.

Note that the code is pointless since the `TButton` class does all this with more convenient properties. However, it should give you the general idea.

➤ *Listing 8: Creating, using and destroying a QButton widget.*

```
uses
  Qt, QTypes;
...
var
  Btn: QPushButtonH;
...
procedure TForm1.FormCreate(Sender: TObject);
var
  Msg: TCaption;
begin
  Btn := QPushButton_create(Handle, PChar('Btn'));
  QPushButton_setGeometry(Btn, 10, 10, 75, 25);
  Msg := 'Press me';
  QButton_setText(Btn, PWideString(@Msg));
  QWidget_show(Btn);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  QPushButton_destroy(Btn);
end;
```

More From The CLXDisplay API

If you take a close look at Listing 6, you will see a couple of handle classes being defined which warrant some further explanation. The `QClxBitBtnH` and `QClxApplicationH` classes evidently seem to be CLX-specific. Indeed, there are a number of classes defined with the `QClx` prefix. They represent classes that do not exist in the real Qt library, but have been implemented in the interface library by Borland as classes inherited from Qt classes.

The other noteworthy classes are `QOpenWidgetH` and `QOpenComboBoxH`. Classes with names in the form `QOpenXXX` have also been defined by Borland, but are shallow descendants of the Qt `QXXX` class. These are access classes and are present to allow the interface library to access and ultimately export some of the protected member functions of the underlying Qt classes.

Occasionally in the CLX source, the developers required access to these methods and access classes provide an easy means of accomplishing this. An example of this would be in `TWidgetControl.ReSubmitFlags` where they need to overcome a Qt issue that occurs when a Qt widget is given a new parent and need to call the protected `QWidget clearWFlags` and `setWFlags` methods. To do this they call `QOpenWidget_clearWFlags` and `QOpenWidget_setWFlags`.

Qt In VisualCLX Components

Windowed VCL controls (those derived from `TWinControl`) have a `Handle` property that refers to the underlying control's window handle. VisualCLX controls (derived from the corresponding `TWidgetControl`) still have a `Handle` property, but rather than being a window handle, it is a Qt widget handle (see Figure 3).

The `TWidgetControl` class defines `Handle` as a generic `QWidgetH` and implements a private `GetHandle` function to return a value of the same type. Most components that inherit from `TWidgetControl` actually redefine `Handle` to be of a more

appropriate type inherited from QWidgetH.

Handle is initialised when the widget is created by the virtual method TWidgetControl.CreateWidget (called from TWidgetControl.CreateHandle). More initialisation of the Qt widget, such as setting initial property values, is performed in InitWidget, which is also virtual and called from CreateHandle. The widget gets destroyed in the virtual TWidgetControl.DestroyWidget method, called from TWidgetControl.DestroyHandle or the control's destructor.

Certain aspects of the widget's behaviour are set with widget flags (values from Qt's WidgetFlags enumerated type), which have names like WidgetFlags_WType_Modal, WidgetFlags_WStyle_StaysOnTop, WidgetFlags_WStyle_SysMenu, WidgetFlags_WStyle_Minimize and WidgetFlags_WStyle_Maximize. A component specifies its widget flags by returning them (combined with the or operator) from an overridden WidgetFlags method.

Sometimes, a VisualCLX property change may necessitate recreating the underlying widget. In this case, any widget details that need to be maintained during this process should be saved in an overridden version of SaveWidgetState and restored in RestoreWidgetState.

The Qt Notification Model

Having grasped the purpose of a VisualCLX component's Handle property, the next thing to understand is the mechanism that Qt uses to notify interested parties of things that have happened. In other words, what is the Qt equivalent of Windows messages and events?

Windows Messages

Before diving in, it might be useful to have a recap of the model used in Microsoft Windows and VCL components. In Windows, when (for example) the mouse is moved over a button, or keys are pressed, *messages* are sent or posted to the button, namely WM_MOUSEMOVE, WM_KEYDOWN and WM_KEYUP. When the mouse is clicked on a button control, WM_LBUTTONDOWN and WM_LBUTTONUP messages are sent to the button. Each of these messages has additional information packaged with it (such as the mouse co-ordinates, or an indication of which key was pressed).

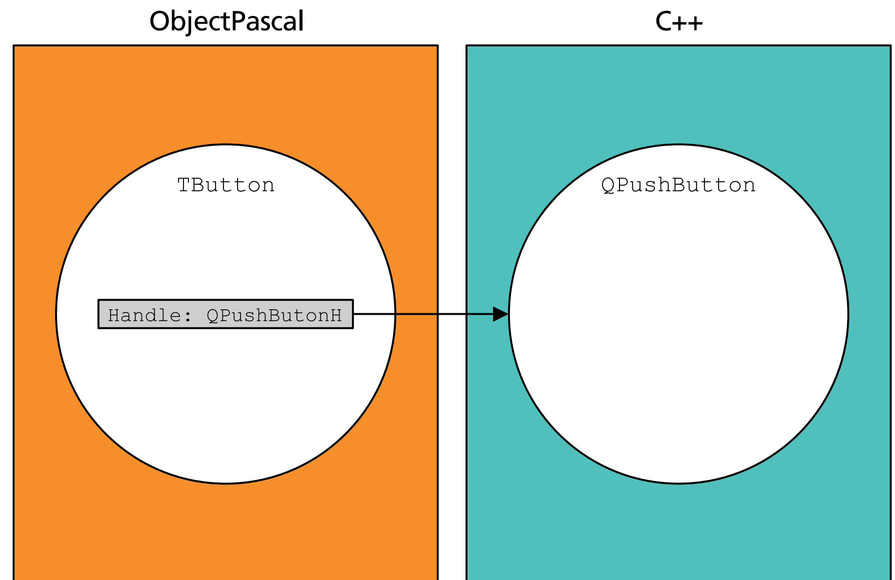
VCL Message Handlers

VCL components pick up Windows messages of interest using dedicated message handling methods (for example, WMLButtonDown and WMLButtonUp). However, they also offer a generic catch-all message handler in their virtual WndProc method. Additionally, a component's messages can be intercepted by another component assigning a compatible method to its WindowProc property (which defaults to pointing at WndProc).

VCL Events

Since the VCL TButton class is interested in when the button control has been clicked, it has message handling methods for WM_LBUTTONDOWN and WM_LBUTTONUP. If the messages arrive in the appropriate order meaning the button has been clicked, it responds by triggering a *VCL event*, in this case, OnClick.

Another part of the program can connect to this event in order to be notified when the event occurs. To connect to the event, the program implements a compatible method called an *event handler*, and assigns it to the button object's event in order to respond to it. An event handler is basically a callback method, and any object's event can have at most one event handler, although any event handler can be shared across multiple events. This is different to the Qt widget approach.



➤ *Figure 3: A VisualCLX component with a reference to a Qt widget.*

X Windows Events

Now let's look at the model used in X Windows and the Qt widgets. In X Windows, when (for example) the mouse is moved over an X client window that acts as a button (such as a QPushButton widget in the Qt library), or keys are pressed, *X events* are sent to the button, namely MotionNotify, KeyPress and KeyRelease. When the mouse is clicked on a button control, ButtonPress and ButtonRelease X events are sent to the button. Each of these X events has additional information packaged with it (such as the mouse co-ordinates, or an indication of the pressed key).

Qt Events And Virtual Methods

Qt widgets pick up X events, wrap them up as Qt event objects (inherited from QEvent) and pass them to dedicated virtual methods (the Windows version of Qt will instead package up information from Windows messages into Qt event objects).

For example, a QMouseEvent object can represent a mouse click

thanks to its `type` method returning either `QEventType_MouseButtonPress` or `QEventType_MouseButtonRelease`. Such an event object might be passed to a Qt widget's `mousePressEvent` or `mouseReleaseEvent` virtual methods. Qt widgets also offer a generic catch-all event filter in their virtual `event` method.

C++ programmers need to inherit from the widget in order to override these virtual methods and add functionality, but we are not able to inherit an ObjectPascal class from a C++ class. Instead, a Qt widget's events can be intercepted by another object (including one written in ObjectPascal) by passing the address of an *event filter* method to the widget's `installEventFilter` method (see later).

CLX Events

Since the CLX `TButton` class is interested in when the button control has been clicked, it installs an event filter (this is actually done at the `TWidgetControl` level) and keeps an eye out for Qt events with event types of `QEventType_MouseButtonPress` and `QEventType_MouseButtonRelease`. If the Qt events arrive in the appropriate order, meaning the button has been clicked, it responds by triggering a *CLX event*, in this case, `OnClick`.

Another part of the CLX program can connect to this event in order to be notified when the event occurs in much the same way as in a VCL program. However, this is not the end of the story.

Qt Signals And Slots

As well as these low-level events (mouse move, mouse press and so on), various Qt widgets already look out for specific high-level events (and also state changes that may be of interest) and allow interested parties to respond using another mechanism called a *signal*. For example, the `QSpinBox` widget wants to notify anyone who is interested when its value has changed. To allow the program to respond to the spin box value changing without having to inherit a new C++ class, the widget defines a signal called `valueChanged`. At an appropriate point the `valueChanged` signal is *emitted* (using a C++ macro).

Some other part of the program can connect to the widget in order to be notified when the signal occurs. The program provides a compatible method called a *slot*, which is then connected to the signal. When the signal occurs, the slot executes in response. Many different slots can be connected to any given signal so this model supports multiplexing, unlike the VCL. When a VCL event fires, a single event handler executes in response. When a Qt signal fires, potentially many C++ slots will execute, one after another.

One of the key differences between a Qt event and a Qt signal is that an event can be handled in an event filter and then killed off, meaning the event will not get to its intended recipient. However, it is not possible to kill off a signal: all registered slots will be triggered one after the other, regardless of what they do.

Unfortunately, because of the way signals and slots are connected in C++, ObjectPascal methods are not suitable for direct use as slots (another reason for the existence of the Qt interface library). Instead, VisualCLX uses hook objects as a way of linking them together (see later section). For example, the VisualCLX `TSpinEdit` ancestor, `TCustomSpinEdit`, uses a hook object to hook the `QSpinBox valueChanged` signal, and triggers its `OnChanged` event when the signal is emitted.

Note that Qt signals should not be confused with the signals that occur in the Linux operating system, despite the same term being used for both. The Qt signal is simply a way for a Qt widget to indicate to other code in the program that something possibly interesting has happened to it; it's an intra-application signal. A Linux signal is typically sent by the operating system to a program to indicate that something important has occurred: it's usually an inter-application signal.

Message Handlers, RIP?

Since we do not have MS Windows sending messages around on Linux, Windows message handlers are redundant. Ultimately, the same dynamic methods called by the VCL message handlers will now be called in VisualCLX through Qt slots (Qt signal handlers).

The underlying message dispatching logic implemented in `TObject.Dispatch` still exists, so you can still use messages internally in an application without a problem, but messages that used to come from outside the program are no longer supported in Kylix. For cross-platform portability, message handlers should also be avoided in Delphi CLX applications as well. Outside application stimuli now come primarily through Qt widget signals.

In the VCL, Windows messages are automatically routed to the window procedure method (typically `WndProc`) and possibly then deferred to message handling methods. We now need to start looking at how a VisualCLX application can react to Qt widget signals and events.

Hook Objects

Hook objects are instances of Borland-written classes implemented in the Qt interface library. There is a hierarchy of hook classes, each of which can hook all the signals supported by the

➤ *Listing 9: Part of the Qt hook handle class hierarchy.*

```
type
  QObject_hookH = class(TObject) end;
    QApplication_hookH = class(QObject_hookH) end;
    QWidget_hookH = class(QObject_hookH) end;
      QButton_hookH = class(QWidget_hookH) end;
        QPushButton_hookH = class(QButton_hookH) end;
      QComboBox_hookH = class(QWidget_hookH) end;
      QFrame_hookH = class(QWidget_hookH) end;
        QTableView_hookH = class(QFrame_hookH) end;
          QMultiLineEdit_hookH = class(QTableView_hookH) end;
```

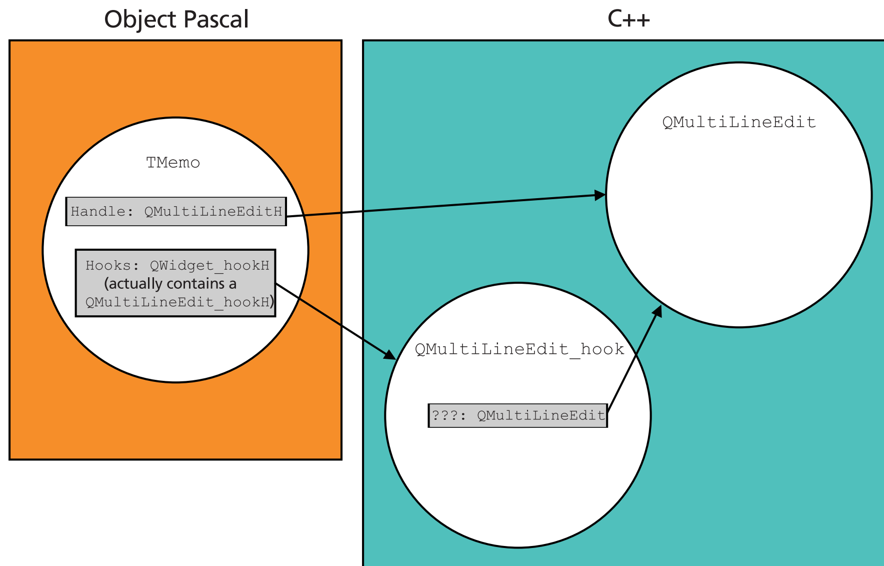
corresponding Qt class (a portion of the corresponding hook handle class hierarchy is shown in Listing 9).

For example, the QObject class defines the destroyed signal, and the QObject_hook class has an appropriate slot that can be connected to it. The hook object's slot is connected to the signal using its hook_destroyed method, accessible through the QObject_hook_hook_destroyed flat method in the interface library. Similarly, the QApplication class (inherited from QObject) defines the lastWindowClosed, aboutToQuit and guiThreadAwake signals, and the QApplication_hook class (inherited from QObject_hook) has slots that can react to those signals, as well as the destroyed signal.

For each Qt class that adds signals, there is a corresponding hook class that allows VisualCLX methods to hook onto its signals. There are also hook classes for Qt base classes that do not define new signals, such as QWidget_hook. Whenever a VisualCLX component needs to hook signals from a Qt class, it uses the corresponding hook class (the same name as the Qt class with a _hook suffix).

When constructing a hook object for a Qt widget, the hook constructor takes the Qt widget handle (the Handle property of the corresponding VisualCLX component); however, the parameter is actually defined as a generic QObject handle (QObjectH).

So, for example, to hook signals in a QMultiLineEdit, the TMemo component creates a QMultiLineEdit_hook object, passing the Handle property to the constructor. This is done immediately after creating the Qt widget itself in the CreateWidget method. The hook constructor returns a hook object handle which is stored in the component's protected Hooks property, defined as type QWidget_hookH in TWidgetControl. Figure 4 illustrates the relationships between the various objects that have been discussed so far.



➤ *Figure 4: A VisualCLX component with a reference to a Qt widget and a hook object.*

Hooking A Qt Signal

In order to actually connect VisualCLX methods to the Qt signals, TWidgetControl has a virtual HookEvents method, called from CreateHandle directly after InitWidget. This method first verifies that there is a hook object (creating a generic QWidget_hook if not) and hooks the widget's destroyed signal (which it inherits from QObject) with its own DestroyedHook method (the effective equivalent of an ObjectPascal slot).

DestroyedHook has the exact same signature as the corresponding Qt signal (which you find in the Qt documentation for QObject, albeit in C++ syntax). Trolltech defines the signal as:

```
void QObject::destroyed()
```

A void function in C++ corresponds to an ObjectPascal procedure, and all the Qt C++ code is compiled with the C calling convention, so DestroyedHook is declared as:

```
procedure DestroyedHook; cdecl;
```

The job of this method is to call the virtual WidgetDestroyed method when the underlying Qt widget ceases to exist, and WidgetDestroyed by default destroys the hook object referred to by the Hooks property.

All the hook object methods take the obligatory hook object handle as the first parameter and one extra parameter called hook. This parameter represents the ObjectPascal hook method and is consistently defined as type QHookH, which itself is defined as a TMethod (a record which can hold a pointer to a method as well as a pointer to an object instance that will execute the method, in other words Self).

Various signals have specific parameter requirements, so there are a whole host of procedural types defined in Qt.pas, one for each signal. Listing 10 shows a small number of them. Note that despite being procedural types for Pascal slots, they all include the word Event, which can be quite confusing.

When hooking a signal, it is typical for a component to declare a TMethod or QHookH variable,

➤ *Listing 10: Procedural types for some Qt signals.*

```
type
  QObject_destroyed_Event = procedure () of object cdecl;
  QApplication_lastWindowClosed_Event = procedure () of object cdecl;
  QApplication_aboutToQuit_Event = procedure () of object cdecl;
  QApplication_guiThreadAwake_Event = procedure () of object cdecl;
  QButton_pressed_Event = procedure () of object cdecl;
  QMultiLineEdit_textChanged_Event = procedure () of object cdecl;
  QMultiLineEdit_returnPressed_Event = procedure () of object cdecl;
  QListBox_highlighted_Event = procedure (index: Integer) of object cdecl;
```

typecast it to the relevant procedural type for the signal in question, and then assign the hook method to it. This variable is then passed to the hook object method that hooks the desired signal. You can see the code from TCustomMemo.HookEvents hooking two signals in Listing 11. Also, Figure 5 illustrates the QMultiLineEdit signal textChanged occurring and triggering the corresponding slot in the hook object, which then calls the TextChangedHook method in the TMemo component.

This approach allows a component to hook into any individual signal it chooses, using an appropriate C calling convention method. All that is left now is to see how it can use an event filter to hook into the stream of events delivered to the widget, as mentioned earlier.

```
procedure TCustomMemo.HookEvents;
var
  Method: TMethod;
begin
  inherited;
  QMultiLineEdit_textChanged_Event(Method) := TextChangedHook;
  QMultiLineEdit_hook_hook_textChanged(QMultiLineEdit_hookH(Hooks), Method);
  QMultiLineEdit_returnPressed_Event(Method) := ReturnPressedHook;
  QMultiLineEdit_hook_hook_returnPressed(QMultiLineEdit_hookH(Hooks), Method);
end;
```

➤ *Listing 11: TCustomMemo hooking the signals it needs.*

```
procedure TWidgetControl.HookEvents;
var
  Method: TMethod;
begin
  if FHooks = nil then begin
    HandleNeeded;
    FHooks := QWidget_hook_create(Handle);
  end;
  TEventFilterMethod(Method) := MainEventFilter;
  Qt_hook_hook_events(FHooks, Method);
  QObject_destroyed_event(Method) := Self.DestroyedHook;
  QObject_hook_hook_destroyed(FHooks, Method);
end;
```

➤ *Listing 12: TWidgetControl hooking the destroyed signal.*

Events And Event Filters

The solution to this lies in event filters. TWidgetControl installs an event filter in its HookEvents method (see Listing 12). The filter method (MainEventFilter) again uses C calling conventions and is passed every event sent to the widget. It first filters out any design-time events and then passes the remainder onto the virtual EventFilter method.

The filter is installed by calling the Qt_hook_hook_events flat method of whatever hook object you are using, which then calls the relevant widget's installEventFilter method for you. It seems that the hook_events method exists in all hook objects and is probably defined (at the C++ level) in the base hook class (Qt_hook) that all the C++ hook classes inherit from. The hook handle class hierarchy in Qt.pas (shown in Listing 9) is therefore incomplete, since QObject_hook inherits from Qt_hook, which inherits from QObject.

We can now see that hook objects perform two distinct jobs. Firstly, they can hook any individual signal in the corresponding widget they work with, using an internal slot to chain back to a CLX method. Secondly, they can install a CLX event filter for the widget.

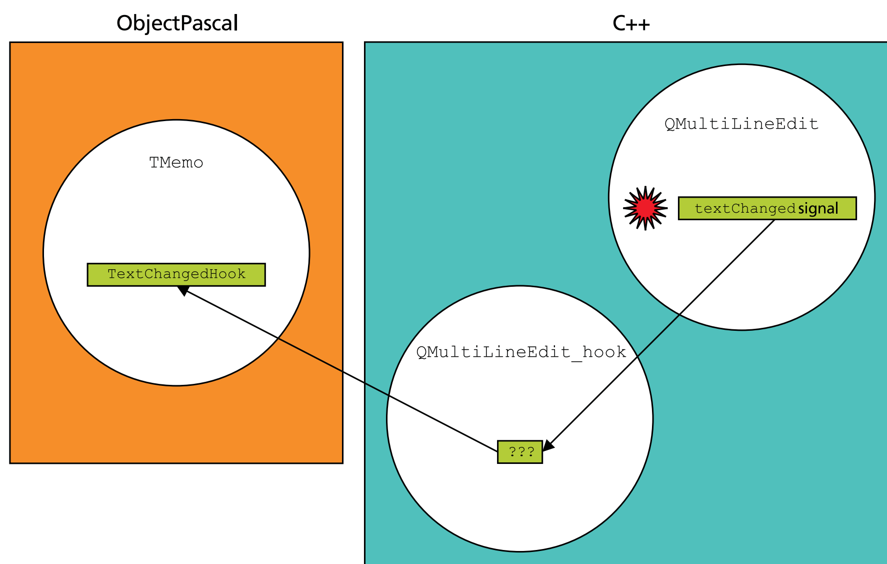
The EventFilter method has this signature:

```
function EventFilter(Sender:
  QObjectH; Event: QEventH):
  Boolean;
```

The Sender parameter represents the Qt widget generating the event and Event is the handle for an object that represents the event. The Qt documentation describes the QEvent base class which only has one method, called type. This returns a value from the QEventType enumerated type to tell you what the event is. The Boolean return value is set to True to indicate the event has been handled and should not go any further (it gets successfully filtered away). A value of False lets the event be passed through to the next object in line in the object's event filter list, which may or may not be the intended recipient. If all event filters return False, the event is given to the underlying widget.

Once you know the event type, you can typecast the Event parameter into an appropriate descendant class. For example, if the QEvent_type flat method returns QEventType_KeyPress, you can typecast Event into a QKeyEventH handle which then allows you to call the

➤ *Figure 5: A Qt widget signal triggering a VisualCLX method.*



flat methods of the QKeyEvent class (for example, QKeyEvent_key, which returns the pressed key as a Qt key constant).

Listing 13 shows the EventFilter method from the TCustomRadioGroup class (the ancestor of TRadioGroup). If a QEventType_show event comes along, the ForceLayout method is called. If a key press event is generated from the Tab key, the event is unconditionally accepted, otherwise it defers to the inherited version of EventFilter.

```
function TCustomRadioGroup.EventFilter(Sender: QObjectH; Event: QEventH):
  Boolean;
begin
  try
    if (QEvent_type(Event) = QEventType_show) then
      ForceLayout;
    if (QEvent_type(Event) = QEventType_KeyPress) and
    (QKeyEvent_key(QKeyEventH(Event)) = Key_Tab) then
      Result := False
    else
      Result := inherited EventFilter(Sender, Event);
  except
    Application.HandleException(Self);
    Result := False;
  end;
end;
```

➤ *Listing 13: The radio group event filter.*

As you can appreciate, there are lots of different event types defined by Qt (see the definition of QEventType in Qt.pas) and many of them have corresponding event classes that will make potentially useful information available to you (see Listing 14). If there is no information to supply, there is usually no specific event class.

Custom Events

Many VCL programmers are used to defining custom Windows messages and sending them to windows (usually controls) in their own application. The equivalent process in Qt involves sending *custom events*.

You make custom events by defining event type values. These must be values of type QEventType, but with ordinal values greater than or equal to the QEventType_ClxUser constant defined in the Qt.pas unit. The VisualCLX code defines its own internal custom events with values greater than or equal to QEventType_ClxBase (the same as the Qt constant QEventType_User), but lower than QEventType_ClxUser.

When you check the type of an event object in EventFilter, QEvent_type will return your custom event type value for you. You can then typecast the Event parameter into a QCustomEventH handle. A QCustomEvent object allows arbitrary data to be stored by passing a raw pointer through as one of its constructor parameters (which begs the question of how you send these custom events, more on this shortly). You can access this data using the QCustomEvent_data flat method.

As an example from the VisualCLX code, the TCustomListBox class's EventFilter method checks for QEventType_MouseButtonPress events and also for a custom event, QEventType_LBClick, defined in QStdCtrls.pas (see Listing 15).

➤ *Listing 14: The event class hierarchy.*

```
QEventH = class(QtH) end;
  QChildEventH = class(QEventH) end;
  QCloseEventH = class(QEventH) end;
  QCustomEventH = class(QEventH) end;
  QDragLeaveEventH = class(QEventH) end;
  QDragResponseEventH = class(QEventH) end;
  QDropEventH = class(QEventH) end;
    QDragMoveEventH = class(QDropEventH) end;
      QDragEnterEventH = class(QDragMoveEventH) end;
  QFocusEventH = class(QEventH) end;
  QHideEventH = class(QEventH) end;
  QKeyEventH = class(QEventH) end;
  QMouseEventH = class(QEventH) end;
  QMoveEventH = class(QEventH) end;
  QPaintEventH = class(QEventH) end;
  QResizeEventH = class(QEventH) end;
  QShowEventH = class(QEventH) end;
  QTimerEventH = class(QEventH) end;
  QWheelEventH = class(QEventH) end;
```

➤ *Listing 15: The listbox's EventFilter method.*

```
const
  QEventType_LBClick = QEventType(Integer(QEventType_ClxBase) + $20);
...
function TCustomListBox.EventFilter(Sender: QObjectH;
  Event: QEventH): Boolean;
begin
  Result := inherited EventFilter(Sender, Event);
  case QEvent_type(Event) of
    QEventType_MouseButtonPress:
      // don't select on a right-click (Qt default behavior)
      Result := QMouseEvent_button(QMouseEventH(Event)) =
        ButtonState_RightButton;
    QEventType_LBClick:
      if not FClicking then begin
        FClicking := True;
        try
          Click;
          Result := True;
        finally
          FClicking := False;
        end;
      end;
  end;
end;
```

Sending Events

For the most part, events are sent by the Qt widgets themselves, but sometimes components need to dispatch events (often custom events) to widgets under program control. Fortunately this is reasonably easy, thanks to a couple of methods in the QApplication widget. The QApplication_sendEvent flat method sends an event directly to a widget and returns the Boolean value from the event handler, whilst QApplication_postEvent adds the event to a queue, which is emptied next time control returns to the main event loop.

According to the Qt documentation, these are both static methods

(like ObjectPascal class methods) and so do not require the QApplication handle to be passed as the first parameter. Instead the two parameters (common to both flat methods) describe the widget intended to receive the event (receiver of type QObjectH) and the event itself (event of type QEventH).

In order to call either of these routines, you need to construct a Qt event object, more often than not a custom event created with QCustomEvent_create. There are two versions of the constructor you can choose from. One takes just an event type and the other takes an event type along with a pointer value which can be used for any custom data. Listing 16 shows a snippet from the TWidgetControl.DestroyWidget method which creates a custom event and posts it to the QApplication widget. The custom data in this case is the widget's own Handle (albeit copied to the local TmpHandle variable).

Incidentally, if you have an event handler that executes a lot of code and takes a long time to execute, you can still call Application.ProcessMessages periodically to allow queued events to be processed. This method calls down to the QApplication_processEvents flat method which processes events for three seconds, or until all the events are cleared, whichever is shorter.

Nomenclature Summary

Having gone through the key points of Qt's usage in VisualCLX, it might be useful to see a summary of the naming conventions used by Borland for all the Qt-related entities. As an aid to making a general template, let's assume there is a fictitious Qt widget called QFoo that can generate a signal called bar, and Borland has written a CLX component wrapper called TFoo. Given these starting points, Table 1 lists all the key identifiers described so far in this article.

Licensing

Whilst this article focuses heavily on what you can find in the Qt.pas unit, you must tread carefully when using it, so as not to contravene any licensing requirements

➤ Listing 16: Posting a custom event.

```
const
  QEventType_CMDestroyWidget = QEventType(Integer(QEventType_ClxBase) + $01);
...
QApplication_postEvent(Application.Handle,
  QCustomEvent_create(QEventType_CMDestroyWidget, TmpHandle))
```

➤ Table 1: Summary of entities used in wrapping a Qt widget.

Identifier	Comments
QFoo	This class is implemented in C++ and so this identifier is not present in ObjectPascal
QFooH	The QFoo handle class, defined in Qt.pas (a typed opaque pointer). Used to access the C++ widget from ObjectPascal in a type-safe manner (but the QFoo methods cannot be directly accessed through this class)
QFoo_create	The widget class flat constructor
QFoo_destroy	The widget class flat destructor
QFoo_hook	The C++ hook class for the QFoo widget. This identifier is not present in ObjectPascal
QFoo_hookH	The hook handle class defined in Qt.pas (a typed opaque pointer)
QFoo_hook_create	The hook class flat constructor
QFoo_hook_destroy	The hook class flat destructor
QFoo_hook_hook_bar	The hook class flat method for hooking the QFoo bar signal
QFoo_bar_Event	Procedural type that defines the signature of an ObjectPascal handler for the QFoo bar signal (via a QFoo_hook hook object), despite the type name including the word Event
TFoo	The VisualCLX component class that represents the QFoo object
TFoo.Handle	This property (of type QFooH) is a handle to the Qt widget
TFoo.Hooks	This property, of type QWidget_hookH, actually holds a QFoo_hookH handle to a QFoo-specific hook object
TFoo.CreateWidget	The virtual method that constructs the QFoo widget and then the QFoo_hook hook object
TFoo.InitWidget	The virtual method that sets up any additional properties of the QFoo widget after having been constructed
TFoo.SaveWidgetState	The virtual method that saves any important QFoo properties during widget reconstruction
TFoo.RestoreWidgetState	The virtual method that restores important QFoo properties after widget reconstruction

laid out by Borland. Apart from what I found in Kylix's license.txt file, the key restrictions were in the DEPLOY file:

'2.6 Restrictions on CLXDisplay API (Qt.pas) usage

'CLXDisplay API, the Qt.pas interface to the Qt runtime, is only licensed for use in VisualCLX applications or a component that derives from `TControl` *in the QControls unit. A VisualCLX application is an application that uses the* `TApplication` *object and uses at least one component derived from* `TControl`*. You are not licensed to use Qt.pas to create applications or components that exclusively call the Qt.pas interfaces. The above restrictions do not apply to applications or components licensed under the GPL. A separate commercial development license from Trolltech is required for use of Qt.pas in any manner other than authorized above.'*

This tells us that since Trolltech allows the development of Linux GPL applications (well, actually a modified form of GPL called QPL), a Kylix GPL application can be written that talks directly to Qt.pas and does not use CLX at all. However, non-GPL applications can only make use of Qt.pas if they are considered a VisualCLX application, which means including the `Application` object and at least one `TControl`-based component.

If you want to write a pure Qt non-GPL application, you must purchase an appropriate licence from Trolltech. Also, since Trolltech does not support QPL development on non-Linux platforms, you are not allowed to write GPL applications for Windows.

More details about the rules concerning what a GPL application must do (it must come with source, or at least a link to where source can be downloaded) will be found in the DEPLOY file.

Oh, by the way, if you are worried that Trolltech may one day go out of business and cause maintenance problems for your CLX applications, there are two things to note. One is that Borland bought a chunk of Trolltech to help protect its investment in Qt (the press release describes the agreement as Borland making a minority investment in Trolltech). The other is the existence of the KDE Free Qt Foundation which guarantees the existence of Qt Free Edition for the development of free software at all times. If the product is discontinued by Trolltech, the Foundation will release the latest version under the BSD licence.

Summary

This has been quite some journey, looking at a number of classes which don't look much like classes, and some of which are present only to bridge a gap between Object Pascal and C++. I hope that now the journey is over, some of the mystery surrounding the use of the Qt widget set in the CLX component library has been laid to rest. As CLX gets used more, and more CLX components get developed, there will undoubtedly be more questions to answer, so keep an eye on *The Delphi Clinic* in future issues.

Acknowledgements

Thanks are due to Adam 'Sparky' Markowitz, from Borland Research & Development, for very helpful feedback on this article.

Brian Long is a freelance trainer and problem solver specialising in Delphi, Kylix and C++Builder work. Visit www.blong.com or email him on brian@blong.com

Copyright ©2001 Brian Long

Further Reading

Details of the availability of FreeCLX, the Open Source project for the Linux CLX library, at http://community.borland.com/article/ 0,1410,27100,00.html.

Trolltech's website, www.trolltech.com.

Trolltech's documentation site, http://doc.troll.no.

Download page for Qt Free Edition, which includes the comprehensive HTML-based Qt documentation, www.trolltech.com/dl/qtfree-dl.html.

The full text of the GPL (GNU Public License) agreement at www.gnu.org/copyleft/gpl.html.

The full text of Trolltech's QPL (Q Public License) agreement at www.trolltech.com/products/downloads/freelicense/license.html.

Announcement of the KDE Free Qt Foundation at www.trolltech.com/company/announce/foundation.html. This ensures the continued existence of Qt Free Edition for development of free software.

Press release discussing the Borland's collaboration with Trolltech at www.borland.com/about/press/2000/trolltech.html.

Programming With Qt, Matthias Kalle Dalheimer, O'Reilly. Good coverage of how Qt works and how you use it as a C++ programmer.

Qt Programming in 24 Hours, Daniel Solin, Sams. This also has good coverage, but I didn't get on with the writing style as well as the O'Reilly book above.

X Window Programming from Scratch by J. Robert Brown, Que. Not that useful for Kylix developers but there is some interesting coverage of the X event system which Qt hides from us.