Mastering™ Kylix™ 2
Marco Cantù; Uberto Barbini

Chapter 7: VisualCLX

Copyright © 2002 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this
publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to
photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-2873-4

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other
countries.

TRADEMARKS: Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms
by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services
listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said
copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights
reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent
of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release
software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software
manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the
completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to
performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged
to be caused directly or indirectly from this book.

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com
Chapter 7

VisualCLX

Now that you’ve been introduced to the Kylix environment and have seen an overview of the Object Pascal language and the basic elements of CLX component library, you are ready to delve into the second part of the book: the use of components and the development of the user interface of applications. This is really what Kylix is about. Visual programming using components is one of the key features of this development environment.

Kylix comes with a large number of ready-to-use components (well, the open source version doesn’t, but the FreeCLX project should compensate for those). We won’t describe every component in detail; if you want to examine each of its properties and methods, you can do so with the Help system. The aim of this book is to show you how to build applications using the advanced features of Kylix predefined components and to discuss specific programming techniques.

We’ll start by covering the development of the user interface of a Linux application using XWindows, and then we’ll move to the Qt layer and the encapsulation provided by Kylix in the VisualCLX library. We’ll continue by comparing this library with Delphi’s VCL and then focus on some of the core classes (particularly TControl). Finally, we’ll try to list all the various visual components you have, because choosing the right basic controls is often a way to get into a project faster.

In this chapter:

◆ The role and architecture of XWindows
◆ VisualCLX and Qt
◆ TControl and TWidgetControl
◆ An overview of the standard components
◆ Handling input focus
◆ Basic and advanced menu construction
XWindows Programming

The user interface of a Linux system is usually provided (nothing is compulsory in this OS!) by XWindows. If you come from the Microsoft Windows world and think that XWindows is a library just like GDI, you are in for quite a surprise. XWindows, in fact, has its roots in the architecture of terminals. In short, XWindows (or simply “X,” as the system is commonly called by its users) is a client/server system operating through sockets.

We don’t mean Internet applications; we’re talking about showing something on a screen and getting input from the mouse and keyboard. In XWindows the server is the terminal, the video, and the input devices. The client is the application running it, your Kylix application, for example, which can sit on the same computer or on a central computer. (The terms client and server can be quite confusing, as we are used to referring to the single PC as the client and the large computer as the server—the opposite of XWindows’ use of the terms.) Needless to say, this is much more flexible than Windows’ integrated approach.

Rather than discussing the differences between the two architectures, however, let’s focus on XWindows programming. Even if you seldom code directly to the XWindows layer and instead rely on a higher-level library like Qt, it can be helpful to have an idea of how XWindows works at a low level. This can be done in Kylix by using the XWindows API directly through the XLib.pas unit.

At this level of development, the concepts are vaguely similar to those of the Windows API. You need to create a window, and there is something like a window procedure. However, every application determines which events/messages it wants to process and will receive only those events. (This and other design decisions are meant to optimize network traffic, for example, to avoid sending messages that won’t be used.) Overall the X API is much simpler than Win32’s, with only a handful of messages, no predefined windows, no predefined controls, and limited drawing support.

For this reason, you’ll generally use a set of widgets, or “windows gadgets,” which have a role similar to windows controls. In Windows you cannot tell the difference between the core windowing features and the base controls (hosted by the same library), while in Linux these are two totally separate layers. Generally, the widgets even provide multiple output styles for compatibility with other user interfaces or native output styles. Kylix programmers generally use the Qt widgets, which are part of Qt and are encapsulated in the VisualCLX library, as discussed later in this chapter.

XWindows doesn’t even define how to draw the borders of a window. This is generally the role of a window manager, which again can provide multiple styles compatible with commonly used operating systems. The two most common window (and desktop) managers are the K Desktop Environment (KDE) and GNU’s Network Object Model Environment (GNOME).

KDE has a Windows look and feel, embeds a high-level integrated browser and file manager called Konqueror, and is even pushing its own office suite, KOffice (although it’s still not a real alternative to Sun’s StarOffice and OpenOffice suites). What is most important to a Kylix developer, however, is that the KDE is natively built using the Qt class library and its set of widgets, the same library that is wrapped by VisualCLX.

Note: Although we focus on KDE in this chapter, Kylix applications also run perfectly well on GNOME, except for a couple of issues: KDE provides a better integration of the support for styles, clipboard, and dragging operations. Even besides KDE and GNOME, there are many window managers. One of them, which provides a Windows 2000 look and feel, was even written with Kylix and is called W2Kwm (there is now an updated Windows XP clone version, called Xpwm; it’s available at http://xpwm.qadram.com).
Low-Level X Programming
We’re going to show you a simple example of a low-level X application because it is instructive to understand the core system differences from Windows. You can find its complete code in Listing 7.1.

Listing 7.1: The xdemo Example Uses the X API Directly

```
program xdemo;

uses
  SysUtils,
  Xlib;

var
  Display: PDisplay;
  Window: TWindow;
  Screen: Integer;
  Event: XEvent;
  Canvas: TGC;
  CanvasValues: XGCValues;

begin
  Display := XOpenDisplay (nil);
  if not Assigned (display) then
    Exit;
  Screen := XDefaultScreen (Display);
  Window := XCreateSimpleWindow (Display,
                                    XRootWindow (Display, Screen),
                                    100, 100, 300, 200, 3, 0, 256*256*256-1); // white
  XSelectInput (Display, Window,
                ButtonPressMask or ExposureMask or PointerMotionMask);
  Canvas := XCreateGC (Display, Window, 0, @CanvasValues);
  XMapWindow (Display, Window);
  while (true) do
    begin
      XNextEvent (display, @Event);
      case Event.xtype of
        Expose: XDrawString (Display, Window,
                             Canvas, 10, 10, 'hello', 5);
        ButtonPressMask: break;
        MotionNotify: XDrawPoint (Display, Window, Canvas,
                                 Event.xmotion.x, Event.xmotion.y);
      end;
    end;
  XDestroyWindow (Display, Window);
  XCloseDisplay (Display);
end.
```
The xdemo program starts by opening an X display on the local computer (we skipped the optional IP address) and selecting a screen. Keep in mind that a Linux’s user interface generally has a series of desktops or virtual monitors. The next step is to create a window by calling the XCreateSimpleWindow function. This provides a display, a root window (the desktop in this case), the coordinates, the border width, the border, and the background.

Once you have a window, you can choose the system events you are interested in (with the XSelectInput call), create a graphic context (with XCreateGC), and activate the window in a display (with XMapWindow). The final portion of the code is the message loop of the XWindows application. The program keeps checking for the next event, handling only three types of events: Expose, which is the request of repainting the window; ButtonPressMask, which closes the program, breaking out of the loop; and MotionNotify, which implies a movement of the mouse pointer and turns on a small pixel. An example of the output of the xdemo program is available in Figure 7.1.

**Figure 7.1**
The output of the xdemo program, which uses the X API directly.

---

**Qt and VisualCLX**

As we’ve mentioned several times, VisualCLX (the visual portion of the CLX library) is a wrapper of the Qt (pronounced “cute”) library, which was built by Trolltech (www.trolltech.com), a Norwegian company with a strong relationship with Borland. Because Qt is a cross-platform library (and not only a Linux one), the Kylix and Delphi (since version 6) versions of CLX are almost identical. Thus, if you use Qt, your code will compile and run the same way under either Windows or Linux.

Qt includes wrappers of the native Qt widgets, which range from basic controls like buttons, edit boxes, and list boxes, to advanced controls like tree and list viewers. Qt also includes painting and graphic support. On Linux, Qt is one of the de facto standard user-interface libraries and is the basis of the KDE desktop environment. On Windows, Qt provides an alternative to the use of the native APIs because Qt controls are very similar to Windows’ standard and common controls.

**Note**  Delphi still includes another library, called VCL, which is mapped directly to the Windows API and isn’t inherently 100 percent portable. However, Borland has minimized the differences between VCL and VisualCLX, so that most applications can be converted with a minimum of effort.

Qt is a C++ library, so it cannot be invoked directly from Kylix Object Pascal code. Instead, the Qt API is accessible through a binding layer defined in the Qt.pas unit. This binding layer consists of a long list of wrappers of almost every Qt class, each suffixed with a final H. For example, the QPainter class of Qt becomes QPainterH type in the binding layer. The Qt.pas unit also includes a very long list with all of the public methods of these classes transformed into standard functions (not class methods) having as their first parameter the object to which they apply. The only relevant
exception to this approach is represented by the constructors of the classes, which are transformed into functions that return the new instance of the class.

This binding layer needs some actual code, which is provided by a C++ shared object library (roughly equivalent to a Windows DLL) built by Borland. This library, called `libqtintf.so`, exports the Qt methods, including every constructor and destructor, as standard C functions.

**WARNING** The need for a mapping library means you have to distribute this library to the end users of your programs. They must have a recent version of Qt installed, of course, but this is seldom a problem.

Finally, using the mapping layer is compulsory for the Qt license that comes with Kylix. Qt is free for noncommercial use under X Windows (it is called Qt Free Edition), but you have to pay a license to Trolltech to develop a commercial application. When you buy Kylix, the Qt license is already paid for by Borland, but you must use Qt primarily through VisualCLX (even if low calls within a CLX application are allowed). What you cannot do is use the Qt-pas unit directly and avoid including the QForms unit (which is mandatory). Borland enforces this by omitting from the Qt interface the `QFormH` and `QApplicationH` constructors.

In most of the Kylix programs in this book, we’ll be using CLX objects and methods. In some cases, we’ll use Qt directly (some examples presented in Chapter 9 will use low-level calls to manipulate bitmaps). It is important to know that some extra Qt features can be used directly, and you can perform low-level calls to bypass CLX bugs. Qt documentation is not included in Kylix help, but it can be found on the Trolltech website ([www.trolltech.com](http://www.trolltech.com)) in HTML and PDF format.

**From Delphi’s VCL to Kylix’s VisualCLX**

There are huge behind-the-scene differences between a native Windows application built with VCL and a portable Qt program developed with VisualCLX. At the low level, Windows uses API function calls and messages to communicate with controls, while Qt uses class methods and direct method callbacks and has no internal messages. Technically, the Qt classes offer a high-level object-oriented architecture, while the Windows API is still bound to its C legacy and a message-based system from 1985 (when Windows was released). VCL offers an object-oriented abstraction on top of a low-level API, while VisualCLX remaps an already high-level interface into a more familiar class library.

**NOTE** Microsoft is apparently starting to abandon the traditional low-level Windows API for a native high-level class library, part of the .NET architecture. Of course, this change won’t happen overnight, but new high-level user-interface technologies might be introduced in .NET. Actually, .NET consists of multiple technologies, including a virtual machine or runtime interpreter, a low-level nonvisual RTL, and a class framework for the native- and browser-based user interfaces. So far, Borland has released an official statement regarding future support for the .NET byte code and virtual machine from Delphi, but there are still no technical details.

If the underlying architectures of the Windows API on one side and Qt on the other side are relevant, the two class libraries built by Borland, VCL, and CLX flatten out most differences, making the code of Delphi and Kylix applications extremely similar.

Having a familiar class library on top of a totally new platform is the advantage for Delphi programmers moving to VisualCLX on Linux. If components are different internally, they are very similar for users. A button is an object of the `TButton` class for both libraries, and it has more or less the same set of methods, properties, and events. Many times, you can recompile your existing programs
for the new class library in a matter of minutes, as long as they don’t use any low-level API calls or legacy features (like the BDE in a Delphi application).

**SAME CLASSES, DIFFERENT UNITS**

One of the cornerstones of the source-code compatibility between CLX and VCL code is the fact that similar classes in the two libraries have the same class name. To return to the previous example, each library has a TButton class that represents a push button. The two classes are so similar that the following code (used to create a button in a given position of the screen with the ‘New’ caption) will work with both libraries:

```pascal
with TButton.Create (Self) do
begin
  SetBounds (20, 20, 80, 35);
  Caption := 'New';
  Parent := Self;
end;
```

The reason it is possible for the two TButton classes to have the same name (even in Delphi 6 where the two libraries coexist) is because they are saved in two different units: StdCtrls in VCL and QStdCtrls in VisualCLX. The entire VisualCLX library is defined by units corresponding to the VCL units, named with the letter Q as a prefix—for example, the QForms unit, the QDialogs unit, the QGraphics unit, and so on. There are also a few units that have no correspondence in VCL, as they map to peculiar features of Qt, such as the QStyle unit.

**DFM AND XFM**

As you create a form at design time, it is saved to a form definition file. Delphi VCL applications use the DFM extension, which stands for Delphi form module. VisualCLX applications use the XFM extension, which stands for cross-platform (that is, X) form module. A form module is the result of streaming the form and its components: the two libraries share the streaming code, so they produce a fairly similar effect. The actual format of DFM or XFM files, which can be based on a textual or binary representation, is identical.

Technically, in Kylix you can use either extension for your applications, unless you want to recompile them in Delphi. (In fact, starting with Delphi 6, the Windows IDE looks at the file extension to determine which library the application should use at design time.) To avoid any possible problem when porting the code, you should probably stick to the XFM extension in Kylix.

If you want to convert a DFM file into an XFM file, you can simply rename the file. However, expect to find some differences in the properties, events, and available components: reopening the form definition for a different library will probably cause quite a few warnings.

**USES STATEMENTS**

The only relevant difference in the source code of a VCL or CLX application relates to the uses statements. The form of the CLX application has the following initial code:

```pascal
unit QLibCompForm;

interface
```
uses
SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;

The form of the VCL program has the traditional uses statement:
unit LibCompForm;

interface

uses
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;

The code of the class and of the only event handler is absolutely identical. Of course, the compiler directive {$R *.dfm} is replaced by a {$R *.xfm} in the CLX version of a standard program.

Converting Existing Applications
Besides starting with new Kylix applications, you might want to convert some of your existing Delphi VCL applications to the new class library and operating system. Even in the unlikely case that you haven't used any Windows-specific or Delphi-specific feature, there is still a series of operations you have to perform to be able to compile your Delphi programs on Kylix (without any specific help from the IDE):

◆ You'll have to rename the DFM file as XFM and update all of the {$R *.DFM} statements to {$R *.xfm} (notice the case difference, which is relevant).

◆ Remember that the case of the XFM file extension in Kylix matters, unlike in Delphi, and you generally have to convert the resource file extension in the project source code from uppercase to lowercase.

◆ You'll have to update all of the uses statements of your program (in the units and project files) to refer to CLX units instead of VCL units. If you miss even a few, you'll bump into trouble when running your application.

Table 7.1 is a comparison of the names of the visual VCL and CLX units, excluding the database portion and some rarely referenced units.

<table>
<thead>
<tr>
<th>VCL</th>
<th>CLX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActnList</td>
<td>QactnList</td>
</tr>
<tr>
<td>Buttons</td>
<td>Qbuttons</td>
</tr>
<tr>
<td>Clipbrd</td>
<td>Qclipbrd</td>
</tr>
<tr>
<td>ComCtrls</td>
<td>QcomCtrls</td>
</tr>
<tr>
<td>Consts</td>
<td>Qconsts</td>
</tr>
<tr>
<td>Controls</td>
<td>Qcontrols</td>
</tr>
</tbody>
</table>

Continued on next page
You might also convert references to Windows and Messages into references to the Qt unit. Some Windows data structures are now also available in the Types unit (see Chapter 5 for details), so you might want to add them to your CLX programs. Note, however, that the QTypes unit is not the CLX version of VCL's Types unit; these two units are totally unrelated.

**THE VCLTOCLX HELPER TOOL**

To help convert some of our own programs, we wrote a simple unit-replacement tool called VclToClx; it is available with its complete source code in the tools folder of the book's CD. You can see an example of its output in Figure 7.2.

**Figure 7.2**

An example of the use of the VclToClx conversion program, which fixes uses statements and DFM/XFM and RES references.
The program converts unit names, based on a configuration file called `remap.conf`, and fixes the DFM issue by renaming the DFM files to XFM and fixing the references in the source code. The program is quite naive, as it doesn't really parse the source code but simply looks for the occurrences of the unit names followed by a comma or semicolon, as happens in a `uses` statement. It also requires that the unit name is preceded by a space, but of course you can modify the program to look for a comma. Don't skip this extra step, or the Forms unit will be turned to QForms, but the QForms unit will be converted again to QQForms!

**TControl and Derived Classes**

In the preceding chapter, we discussed the base classes of the Kylix library, focusing particularly on the `TComponent` class. One of the most important subclasses of `TComponent` is `TControl`, which corresponds to visual components. This base class defines general concepts, such as the position and the size of the control, the parent control hosting it, and more. For an actual implementation, though, you have to refer to its two subclasses: `TWidgetControl` and `TGraphicControl` (in the VCL these are `TWinControl` and `TGraphicControl`). Here are their key features:

**Window-based controls (also called windowed controls)** Visual components based on an operating-system window. A `TWidgetControl` in CLX has a Qt handle, a reference to the internal Qt object. From a user perspective, windowed controls can receive the input focus, and some of them can contain other controls. This is the biggest group of components in the library. Windowed controls can be further divided into two groups: wrappers of native controls of Qt and custom controls of the VisualCLX library, which generally inherit from the generic `TCustomControl`.

**Graphical controls (also called nonwindowed controls)** Visual components that are not based on an operating-system window. Therefore, they have no system handle, cannot receive the focus, and cannot contain other controls. These controls inherit from `TGraphicControl` and are painted by their parent form, which sends them mouse-related and other events. Examples of nonwindowed controls are the Label and SpeedButton components.

**Widget Ownership and Internals**

The relationship of the Kylix class library with Qt starts with the class `TWidgetControl`. Every `TWidgetControl` has an internal Qt object, which is referenced using the `Handle` property (the same name as the corresponding Windows property, but totally different behind the scenes).

A `TWidgetControl` object generally owns the corresponding Qt/C++ object. The CLX class uses delayed construction (the internal object is not created until one of its methods is actually required) implemented in the `InitWidget` and other methods. The CLX class also frees the internal object when it is destroyed. However, it is also possible to create a widget around an existing Qt object; in this case, the CLX object won't own the Qt object and won't destroy it. This behavior is indicated in the `OwnHandle` property.

To be more precise, each VisualCLX component has two associated C++ objects: the Qt Handle (referencing the widget itself) and the Qt Hook (the object receiving the Qt events and signals, which are discussed in the next paragraph). With the design of Qt 2 (upon which Kylix 1 and 2 are based), this has to be a C++ object, which acts as an intermediary to the event handlers of the Object Pascal control. The `HookEvents` method associates the hook object to the CLX control.
Delphi programmers should notice that in Qt there are two different types of events:

- **Events** are the translation of X Windows user events, such as a key press, a mouse move, or a paint event.
- **Signals** are internal component events, such as a click or a change notification. (Note that Qt signals are not related to Linux ones.)

The events of a CLX component, however, merge events and signals. Generic Kylix control events include `OnMouseDown`, `OnMouseMove`, and `OnKeyDown`, `OnChange`, `OnPaint`, and many others, just as in the VCL (which fires most events as a response to Windows messages).

**NOTE** Expert programmers may notice that there is a seldom-used `EventHandler` method, which corresponds more or less to the `WndProc` method of a VCL `TWinControl`.

**Parent and Controls**

The `Parent` property of a control indicates which other control is responsible for displaying it. When you drop a component into a form in the Form Designer, the form will become both parent and owner of the new control. But if you drop the component inside a Panel, ScrollBox, or any other container component, this will become its parent, while the form will still be the owner of the control.

When you create the control at runtime, you need to set the owner (using the `Create` constructor parameter), but you must also set the `Parent` property or the control won’t be visible.

Like the `Owner` property, the `Parent` property has an inverse. The `Controls` array, in fact, lists all of the controls parented by the current control, numbered from 0 to `ControlsCount - 1`. You can scan this property to operate on all of the controls hosted by another one, eventually using a recursive method that operates on the controls parented by each subcontrol.

**Properties Related to Control Size and Position**

Some of the properties introduced by `TControl` and common to all controls are those related to size and position. The position of a control is determined by its `Left` and `Top` properties; its size is determined by the `Height` and `Width` properties. Technically, all components have a position because when you reopen an existing form at design time, you want to be able to see the icons for the non-visual components in exactly the position where you placed them. This position is visible in the form file.

**TIP** To change any of the positional or size properties, you call the single `SetBounds` method. Any time you need to change two or more of these properties at once, calling `SetBounds` directly will speed up the program. Another method, `BoundsRect`, returns the rectangle bounding of the control and corresponds to accessing those four properties.

An important feature of the position of a component is that, like any other coordinate, it always relates to the client area of its parent component (indicated by its `Parent` property). For a form, the client area is the surface included within its borders (excluding the borders themselves). It would be messy to work in screen coordinates, although there are some ready-to-use methods that convert the coordinates between the form and the screen and vice versa.
Note, however, that the coordinates of a control are always relative to the parent control, such as a form or another container component. If you place a panel in a form and a button in a panel, the coordinates of the button relate to the panel and not to the form containing the panel. In fact, in this case, the parent component of the button is the panel.

**Activation and Visibility Properties**

There are two basic properties you can use to let the user activate or hide a component. The simpler is the `Enabled` property. When a component is disabled (that is, `Enabled` is set to False), usually some visual hint indicates this state to the user. At design time, the disabled property does not always have an effect, but at runtime, disabled components are generally grayed.

For a more radical approach, you can completely hide a component, either by using the corresponding `Hide` method or by setting its `Visible` property to False. Be aware, however, that reading the status of the `Visible` property does not tell you whether the control is actually visible. In fact, if the container of a control is hidden, even if the control is set to `Visible`, you cannot see it. For this reason, there is another property, `Showing`, which is a runtime and read-only property. You can read the value of `Showing` to know whether the control is really visible to the user; that is, if it is visible, its parent control is also visible, the parent control of the parent control is also visible, and so on.

**Fonts**

Two properties often used to customize the user interface of a component are `Color` and `Font`. Several properties are related to the color. The `Color` property itself usually refers to the background color of the component. There is also a `Color` property for fonts and many other graphic elements. Many components also have a `ParentColor` and a `ParentFont` property, indicating whether the control should use the same font and color as its parent component, which is usually the form. You can use these properties to change the font of each control on a form by setting only the `Font` property of the form itself.

When you set a font, either by entering values for the attributes of the property in the Object Inspector or by using the standard font selection dialog box, you can choose one of the fonts installed in the system. The fact that Kylix allows you to use all the fonts installed on your system has both advantages and drawbacks. The main advantage is that if you have a number of nice fonts installed, your program can use any of them. The drawback is that if you distribute your application, these fonts might not be available on your users’ computers.

If your program uses a font that your users don’t have, XWindows will select some other font to use in its place. A program’s carefully formatted output can be ruined by the font substitution. For this reason, you should probably rely only on standard fonts.

**Colors**

There are various ways to set the value of a color. The type of this property is `TColor`. For properties of this type, you can choose a value from a series of predefined name constants or enter a value directly. The constants for colors include `clBlue`, `clSilver`, `clWhite`, `clGreen`, `clRed`, and many others.

As a better alternative, you can use one of the colors used by the system to denote the status of given elements. CLX includes color constants like `clBackground`, which is the standard color of a
form; clBase, used by edit boxes and other visual controls; clActiveForeground, the foreground color for active controls; and clDisabledBase, the background color for disabled text controls. All the color constants mentioned here are listed in CLX Help file under the “TColor type” topic.

**NOTE** Note that Delphi’s VCL has a set of totally different and incompatible system color constants, based on the predefined Windows colors such as the background of a window (clWindow), the color of the text of a highlighted menu (clHighlightText), the active caption (clActiveCaption), and the ubiquitous button face color (clBtnFace).

Another option is to specify a `TColor` as a number (a 4-byte hexadecimal value) instead of using a predefined value. If you use this approach, you should know that the low 3 bytes of this number represent RGB color intensities for blue, green, and red, respectively. For example, the value $00FF0000 corresponds to a pure blue color, the value $0000FF00 to green, the value $000000FF to red, the value $00000000 to black, and the value $00FFFFFF to white. By specifying intermediate values, you can obtain any of 16 million possible colors.

Instead of specifying these hexadecimal values directly, you can write an `RGB` function with three parameters, all ranging from 0 to 255 and indicating the amount of red, green, and blue. The `RGB` function is natively available in Delphi, but you can define your own in Kylix like this:

```pascal
function RGB (red, green, blue: Byte): Cardinal;
begin
  Result := blue + green * 256 + red * 256 * 256;
end;
```

The highest-order byte of the `TColor` type is used to indicate which palette should be searched for the closest matching color, but palettes are too advanced a topic to discuss here. (Sophisticated imaging programs also use this byte to carry transparency information for each display element on the screen.) However, regarding palettes and color matching, note that the system sometimes replaces an arbitrary color with the closest available solid color, at least in video modes that use a palette. This is always the case with fonts, lines, and so on. At other times, the system uses a dithering technique to mimic the requested color by drawing a tight pattern of pixels with the available colors.

### Opening the Component Tool Box

So you want to write a Kylix application. You open a new Kylix project and find yourself faced with a large number of components (at least, you are if you have the Professional version or have installed FreeCLX—the Open Edition of Kylix doesn’t have all of these components). The problem is that for every operation, there are multiple alternatives. For example, you can show a list of values using a list box, a combo box, a radio group, a string grid, a list view, or even a tree view if there is a hierarchical order. Which should you use? That’s difficult to say. There are many considerations, depending on what you want your application to do. For this reason, we’ve provided a highly condensed summary of alternative options for a few common tasks.

**NOTE** For some of the controls described in the following sections, Kylix also includes a data-aware version, usually indicated by the DB prefix. As you’ll see in Chapter 14, the DB version of a control typically serves a role similar to that of its “standard” equivalent, but the properties and the ways you use it are often quite different. For example, in an Edit control you use the `Text` property, while in a DBEdit component you access the `Value` of the related field object.
The Text Input Components
Although a form or component can handle keyboard input directly by using the OnKeyPress event, this isn't a common operation. Qt provides ready-to-use controls you can use to get string input and even build a simple text editor.

The Edit Component
The Edit component allows the user to enter a single line of text. You can also display a single line of text with a Label or a StaticText control, but these components are generally used only for fixed text or program-generated output, not for input. By the way, there is also a native LCD digit control you can use to display numbers.

The Edit component uses the Text property, whereas many other controls use the Caption property to refer to the text they display. The only condition you can impose on user input is the number of characters to accept. If you want to accept only specific characters, you can handle the OnKeyPress event of the edit box. For example, you can write a method that tests whether the character is a number or the Backspace key (which has a numerical value of 8). If it's neither, you change the value of the key to the null character (#0) so that it won't be processed by the edit control and will produce a warning beep:

```pascal
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  // check if the key is a number or backspace
  if not (Key in ['0'..'9', #8]) then
    begin
      Key := #0;
      Beep;
    end;
  end;
end;
```

Unlike in VCL, the Edit control has no Undo mechanism built in. Another difference is that the PasswordChar property is replaced by the EchoMode property. You don't determine the character to display, but rather whether to echo the entered text or display an asterisk instead.

The MaskEdit Component
To customize the input of an edit box further, you can use the MaskEdit component, which has an EditMask property. This is a string indicating whether each character should be uppercase, lowercase, a number, or other similar conditions. You can see the editor of the EditMask property in Figure 7.3.

**Figure 7.3**
The MaskEdit component's EditMask property editor.
The Input Mask Editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the literals present in the mask, together with the final string. For example, you can choose either to display the parentheses around the area code of a phone number only as an input hint, or to save them with the string holding the resulting number. These two entries in the Input Mask Editor correspond to the last two fields of the mask (separated by semicolons).

**TIP** Clicking the Masks button of the Mask Editor lets you choose predefined input masks for different countries. Some are available in the Kylix bin folder.

**The Memo Component**
Both of the controls discussed so far allow a single line of input. The Memo component, by contrast, can host several lines of text but allows only a single font for the entire text. You can work on the text of the memo line by line (using the Lines string list) or access the entire text at once (using the Text property).

If you want to host a large amount of text or change fonts and paragraph alignments, in VCL you can use the RichEdit control, something not available in Kylix. As the latest version of Qt has a similar native control, these might be supported by future versions of Kylix.

**The TextViewer Control**
What Qt and CLX offer that Windows and the VCL lack is a full-blown HTML viewer control, which is very powerful for displaying formatted text (but not for typing it). This HTML viewer is embedded in two different controls, the single-page TextViewer control and the TextBrowser control with active links.

As a simple demo, we added a memo and a text viewer to a CLX form and connected them so that everything you type on the memo is immediately displayed in the viewer. We called the example HtmlEdit not because this is a real HTML editor, but because this is the simplest way we know of to build an HTML preview inside a program. The form of the program is visible at runtime in Figure 7.4 while typing some text inside a cell of the table.

**Selecting Options**
There are two standard controls that allow the user to choose different options, as well as controls for grouping sets of options.

**The CheckBox and RadioButton Components**
The first standard option-selecting control is the check box, which corresponds to an option that can be selected regardless of the status of other check boxes. Setting the AllowGrayed property of the check box allows you to display three different states (selected, not selected, and grayed), which alternate as a user clicks the check box.

The second type of control is the radio button, which corresponds to an exclusive selection. Two radio buttons on the same form or inside the same radio group container cannot be selected at the same time, and one of them should always be selected. As programmer, you are responsible for selecting one of the radio buttons as the program starts, either at design time or in code.
The GroupBox Components

To host several groups of radio buttons, you can use a GroupBox control to hold them together, both functionally and visually. To build a group box with radio buttons, simply place the GroupBox component on a form and then add the radio buttons to the group box.

You can handle the radio buttons individually, but it’s easier to navigate through the array of controls owned by the group box, as discussed in the previous chapter. Here is a small code excerpt used to get the text of the selected radio button of a group:

```pascal
var
  I: Integer;
  Text: string;
begin
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Text := (GroupBox1.Controls[I] as TRadioButton).Caption;
```

The RadioGroup Component

Kylix has a similar component that can be used specifically for radio buttons: the RadioGroup component. A RadioGroup is a group box with some radio buttons inside it. The difference is that these internal radio buttons are managed automatically by the container control.

Using the RadioGroup component is generally easier than using the GroupBox, since the various items are part of a list, as in a list box. This is how you can get the text of the selected item:

```pascal
Text := RadioGroup1.Items [RadioGroup1.ItemIndex];
```

Another advantage is that the RadioGroup component can automatically align its radio buttons in one or more columns (as indicated by the Columns property), and you can easily add new choices.
at runtime by adding strings to the Items string list. By contrast, adding new radio buttons to a GroupBox would be quite complex.

**Lists**
When you have many selections, radio buttons are not appropriate. The usual number of radio buttons is no more than five or six to avoid cluttering the user interface; when you have more choices, you can use a list box or one of the other controls that displays lists of items and allows the selection of one of them.

**The ListBox Component**
The selection of an item in a list box uses the Items and ItemIndex properties as in the code just shown for the RadioGroup control. If you often need access to the text of selected list box items, you can write a small wrapper function like this:

```pascal
function SelText (List: TListBox): string;
var
  nItem: Integer;
begin
  nItem := List.ItemIndex;
  if nItem >= 0 then
    Result := List.Items [nItem]
  else
    Result := '';
end;
```

Another important feature is that by using the ListBox component, you can choose between allowing only a single selection, as in a group of radio buttons, and allowing multiple selections, as in a group of check boxes. You make this choice by specifying the value of the MultiSelect property. There are two kinds of multiple selections in list boxes: *multiple selection* and *extended selection*. In the first case, a user selects multiple items simply by clicking them, while in the second case the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items, respectively. The two alternatives are determined by the status of the ExtendedSelect property.

For a multiple-selection list box, a program can retrieve information about the number of selected items by using the SelCount property, and it can determine which items are selected by examining the Selected array. This array of Boolean values has the same number of entries as the list box. For example, to concatenate all the selected items into a string, you can scan the Selected array as follows:

```pascal
var
  SelItems: string;
  nItem: Integer;
begin
  SelItems := '';
  for nItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [nItem] then
      SelItems := SelItems + ListBox1.Items[nItem] + ' ';
```
As opposed to the VCL, in CLX a ListBox can be configured to use a fixed number of columns and rows, using the Columns, Row, ColumnLayout, and RowLayout properties.

THE COMBOBOX COMPONENT
List boxes take up a lot of screen space, and they offer a fixed selection—that is, a user can choose only among the items in the list box and cannot enter any choice that the programmer did not specifically foresee.

You can solve both problems by using a ComboBox control, which combines an edit box and a drop-down list. The behavior of a ComboBox component changes a lot depending on the value of its Style property:

◆ The csDropDown style defines a typical combo box, which allows direct editing and displays a list box on request.
◆ The csDropDownList style defines a combo box that does not allow editing (but uses the keystrokes to select an item).
◆ The csSimple style defines a combo box that always displays the list box below it.

Note also that accessing the text of the selected value of a combo box is easier than doing the same operation for a list box, since you can simply use the Text property. A useful and common trick for combo boxes is to add a new element to the list when a user enters some text and presses the Enter key, a feature that CLX supports automatically.

WARNING In CLX, some of the combo box events fire at different times from VCL, so your Delphi code might compile properly but not work as expected.

Kylix 2 includes two new events for the combo box. The OnCloseUp event corresponds to the closing of the drop-down list and complements the preexisting OnDropDown event. The OnSelect event fires only when the user selects something in the drop-down list, as opposed to typing in the edit portion.

Another very nice feature of Kylix is the AutoComplete property. When it is set, the ComboBox component automatically locates the string nearest to the one the user is entering, suggesting the final part of the text. The core of this feature is implemented in the TCustomListBox.KeyPress method.

THE LISTVIEW AND TREEVIEW COMPONENTS
If you want an even more sophisticated list, you can use the ListView control, which will make the user interface of your application look very modern. This component is slightly more complex to use, as described at the beginning of Chapter 8. The ListView control in CLX hasn’t got the small/large icon styles of the Windows counterpart, but a companion control, the IconView, provides this capability.

Other alternatives for listing values are the TreeView common control, which shows items in a hierarchical output, and the StringGrid control, which shows multiple elements for each line. The string grid control is described in Chapter 10.
Ranges
Finally, there are a few components you can use to select values in a range. Ranges can be used for numeric input and for selecting an element in a list.

The ScrollBar Component
The stand-alone ScrollBar control is the original component of this group, but it is seldom used by itself. Scroll bars are usually associated with other components, such as list boxes and memos, or are associated directly with forms. In all these cases, the scroll bar can be considered part of the surface of the other components. For example, a form with a scroll bar is actually a form that has an area resembling a scroll bar painted on its border. By resembling, we mean that it is not technically a separate window of the ScrollBar component type. These “fake” scroll bars are usually controlled in Kylix using two specific properties of the form (VertScrollBar and HorzScrollBar) or other properties for the various components hosting them.

The TrackBar and ProgressBar Components
Direct use of the ScrollBar component is quite rare, especially with the availability of the TrackBar, which has a more modern user interface for range selections. The companion ProgressBar control, which allows the program to output a value in a range, typically shows the progress of a lengthy operation.

The SpinEdit Component
Another related control is the SpinEdit component, which has two up and down buttons on the side of an edit box (these buttons are displayed as arrows or plus and minus signs, depending on the value of the ButtonStyle property). The SpinEdit is used to enter a number as well as to select and display a number within a range. You can also add a prefix or a suffix to the numeric value.

The ScrollBox Component
The ScrollBox control represents a region of a form that can scroll independently from the rest of the surface. For this reason, the ScrollBox has two scrollbars that are used to move the embedded components. You can easily place other components inside a ScrollBox, as you do with a panel. In fact, a ScrollBox is basically a panel with scroll bars to move its internal surface, an interface element used in many applications. When you have a form with many controls and a toolbar or status bar, you might use a ScrollBox to cover the central area of the form, leaving its toolbars and status bars outside of the scrolling region. By relying on the scrollbars of the form, in fact, you might allow the user to move the toolbar or status bar out of view, a very odd situation indeed.

Handling the Input Focus
Using the TabStop and TabOrder properties available in most controls, you can specify the order in which controls will receive the input focus when the user presses the Tab key. Instead of setting the tab order property of each component of a form manually, you can use the shortcut menu of the Form Designer to activate the Edit Tab Order dialog box, as shown in Figure 7.5.
Aside from these basics settings, it is important to know that each time a component receives or loses the input focus, it receives a corresponding `OnEnter` or `OnExit` event. This allows you to fine-tune and customize the order of the user operations. Some of these techniques are demonstrated by the InFocus example, which creates a fairly typical password-login window. Its form has three edit boxes with labels indicating their meaning, as shown in Figure 7.6. At the bottom of the window is a status area with prompts guiding the user. Each item needs to be entered in sequence.

For the output of the status information, we used the StatusBar component with a single output area (obtained by setting its `SimplePanel` property to `True`). Here is a summary of the properties for this example. Notice the & character in the labels, indicating a shortcut key, and the connection of these labels with corresponding edit boxes (using the `FocusControl` property):

```pascal
object FocusForm: TFocusForm
  ActiveControl = EditFirstName
  Caption = 'InFocus'
object Label1: TLabel
  Caption = '&First name'
  FocusControl = EditFirstName
end
object EditFirstName: TEdit
  OnEnter = GlobalEnter
  OnExit = EditFirstNameExit
end
object Label2: TLabel
```
The program is very simple and performs only two operations. The first is to identify, in the status bar, the edit control that has the focus. It does this by handling the controls' `OnEnter` event, possibly using a single generic event handler to avoid repetitive code. In the example, instead of storing some extra information for each edit box, we checked each control of the form to determine which label is connected to the current edit box (indicated by the `Sender` parameter):

```pascal
procedure TFocusForm.GlobalEnter(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    // if the control is a label
    if (Controls[I] is TLabel) and
      // and the label is connected to the current edit box
      (TLabel(Controls[I]).FocusControl = Sender) then
      // copy the text, leaving off the initial & character
      Statusbar1.SimpleText := 'Enter ' +
        Copy (TLabel(Controls[I]).Caption, 2, 1000);
end;
```

The second event handler of the form relates to the `OnExit` event of the first edit box. If the control is left empty, it refuses to release the input focus and sets it back before showing a message to the user. The methods also look for a given input value, automatically filling the second edit box and moving the focus directly to the third one:

```pascal
procedure TFocusForm.EditFirstNameExit(Sender: TObject);
begin
  if EditFirstName.Text = '' then
    begin
      // don't let the user get out
```
EditFirstName.SetFocus;
MessageDlg('First name is required', mtError, [mbOK], 0);
end
else if EditFirstName.Text = 'Admin' then
begin
  // fill the second edit and jump to the third
  EditLastName.Text := 'Admin';
  EditPassword.SetFocus;
end;
end;

Working with Menus

Working with menus and menu items is generally quite simple. This section offers only some very brief notes and a few more advanced examples. The first thing to keep in mind about menu items is that they can serve different purposes:

- **Commands** Menu items used to execute an action.
- **State-setters** Menu items used to toggle an option on and off to change the state of a particular element. These commands usually have a check mark on the left to indicate they are active.
- **Radio items** Grouped to represent alternative selections, like radio buttons. To obtain radio menu items, simply set the RadioItem property to True and set the GroupIndex property for the alternative menu items to the same value. If you check one, all of the other items of the groups will be automatically unchecked.
- **Dialog menu items** Cause a dialog box to appear and are usually indicated by an ellipsis (three dots) at the end of their caption.

As you enter new elements in the Menu Designer, Kylix creates a new component for each menu item and lists it in the Object Inspector (although nothing is added to the form). To name each component, Kylix uses the caption you enter and appends a number (so that Open becomes **Open1**). Because the IDE removes spaces and other special characters in the caption when it creates the name and the menu item separators are set up using a hyphen as caption, these items would have an empty name. For this reason Kylix adds the letter **N** to the name, appending the number and generating items called **N1**, **N2**, and so on.

**Accelerator Keys**

Although captions of menu items should generally have an accelerator key, indicated by entering the & character in front of it, Kylix provides an automatic accelerator key if you omit one. The automatic accelerator-key system of CLX can figure out if you have entered conflicting accelerator keys and fix them on-the-fly. However, this doesn’t mean you should stop adding custom accelerator keys with the & character—the automatic system simply uses the first available letter and doesn’t follow any default standards. You’ll generally be able to provide more mnemonic keys yourself than if you use those chosen by the automatic system.
This feature is controlled by the `AutoHotkeys` property, which is available in the main menu component and in each of the pull-down menus and menu items. In the main menu, this property defaults to `maAutomatic`, while in the pull-downs and menu items it defaults to `maParent`, so that the value you set for the main menu component will be used automatically by all the subitems, unless they have a specific value of `maAutomatic` or `maManual`.

The engine behind this system is the `RethinkHotkeys` method of the `TMenuItem` class, and the companion `InternalRethinkHotkeys`. There is also a `RethinkLines` method, which checks whether a pull-down has two consecutive separators or begins or ends with a separator. In all these cases, the separator is automatically removed.

One of the reasons Kylix includes this feature is to support applications that load packages dynamically, each one adding commands to the main form menu. This happens, for example, in the Kylix IDE itself.

**Pop-Up Menus and the `OnContextMenu` Event**

In addition to the `MainMenu` component, you can use the similar `PopupMenu` component. This is typically displayed when the user right-clicks a component that uses the given pop-up menu as the value for its `PopupMenu` property.

However, besides connecting the pop-up menu to a component with the corresponding property, you can call its `Popup` method, which requires the position of the pop-up in screen coordinates. The proper values can be obtained by converting a local point to a screen point with the `ClientToScreen` method of the local component, as in this code fragment for a label:

```pascal
procedure TForm1.Label3MouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ScreenPoint: TPoint;
begin
  // if some condition applies...
  if Button = mbRight then
  begin
    ScreenPoint := Label3.ClientToScreen (Point (X, Y));
    PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
  end;
end;
```

An alternative approach is the use of the `OnContextMenu` event. This event fires when a user right-clicks a component—exactly what we traced above with the test if `Button = mbRight`. You can use this event to modify and then fire a pop-up menu, with code like this:

```pascal
procedure TFormPopup.Label1ContextPopup(Sender: TObject; MousePos: TPoint; var Handled: Boolean);
var
  ScreenPoint: TPoint;
begin
  // add dynamic items
  PopupMenu2.Items.Add (NewLine);
  PopupMenu2.Items.Add (NewItem (TimeToStr (Now), 0, False, True, nil, 0, ''));
```

This example adds some dynamic behavior to the shortcut menu, adding a temporary item indicating when the pop-up menu is displayed. This is not particularly useful, but we did it to highlight that if you need to display a plain pop-up menu, you can easily use the `PopupMenu` property of the control in question or one of its parent controls. Handling the `OnContextMenu` event makes sense only when you want to do some extra processing.

The `Handled` parameter is preinitialized to False, so that if you do nothing in the event handler, the normal pop-up menu processing will occur. If you do something in your event handler to replace the normal pop-up menu processing (such as popping up a dialog or a customized menu, as in this case), you should set `Handled` to True, and the system will stop processing the message. However, setting `Handled` to True should be fairly rare, as you'll generally use the `OnContextPopup` to dynamically create or customize the pop-up menu, although then you can let the default handler actually show the menu.

The handler of an `OnContextPopup` event isn’t limited to displaying a pop-up menu. It can do any other operation, such as directly displaying a dialog box. Here is an example of a right-click operation used to change the color of the control:

```pascal
procedure TFormPopup.Label2ContextPopup(Sender: TObject; MousePos: TPoint; var Handled: Boolean);
begin
  if ColorDialog1.Execute then
    Label2.Color := ColorDialog1.Color;
  Handled := True;
end;
```

All the code snippets in this section are available in the simple CustPop example on the book’s companion CD.

### Creating Menu Items Dynamically

Besides defining the structure of a menu with the Menu Designer and modifying the status of the items using the `Checked`, `Visible`, and `Caption` properties, you can create an entire menu or portions of one at runtime. This makes sense, for example, when you have many repetitive items, or when the menu items depend on some system configuration or user permissions.

The basic idea is that each object of the `TMenuItem` class—which Kylix uses for both menu items and pull-down menus—contains a list of menu items. Each of these items has the same structure, in a kind of recursive way. A pull-down menu has a list of submenus, and each submenu has a list of submenus, each with its own list of submenus, and so on. The properties you can use to explore the structure of an existing menu are `Items`, which contains the actual list of menu items, and `Count`,
which contains the number of subitems. Adding new menu items or entire pull-down menus to an
existing menu is fairly easy, particularly if you can write a single event handler for all of them.

This is demonstrated by the DynaMenu example, which also illustrates the use of menu check
marks, radio items, and many other features of menus that aren't described in detail in this text. As
soon as you start this program, it creates a new pull-down to change the font
size of a big label hosted by the form. Instead of creating a bunch of menu items with captions indicating sizes ranging from 8 to 48, you can let the program do this repetitive work for you.

The new pull-down menu should be inserted in the Items property of the MainMenu1 component.

You can calculate the position by asking the MainMenu component for the previous pull-down menu:

```pascal
procedure TFormColorText.FormCreate(Sender: TObject);
var
    PullDown, Item: TMenuItem;
    Position, I: Integer;
begin
    // create the new pull-down menu
    PullDown := TMenuItem.Create(Self);
    PullDown.AutoHotkeys := maManual;
    PullDown.Caption := '&Size';
    PullDown.OnClick := SizeClick;
    // compute the position and add it
    Position := MainMenu1.Items.IndexOf(Options1);
    MainMenu1.Items.Insert(Position + 1, PullDown);
    // create menu items for various sizes
    I := 8;
    while I <= 48 do
    begin
        // create the new item
        Item := TMenuItem.Create(Self);
        Item.Caption := IntToStr(I);
        // make it a radio item
        Item.GroupIndex := 1;
        Item.RadioItem := True;
        // handle click and insert
        Item.OnClick := SizeItemClick;
        PullDown.Insert(PullDown.Count, Item);
        I := I + 4;
    end;
    // add extra item at the end
    Item := TMenuItem.Create(Self);
    Item.Caption := 'More...';
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle it by showing the font selection dialog
    Item.OnClick := Font1Click;
    PullDown.Insert(PullDown.Count, Item);
end;
```
As you can see in the preceding code, the menu items are created in a while loop, setting the radio item style and calling the Insert method with the number of items as a parameter to add each item at the end of the pull-down. At the end, the program adds one extra item, which is used to set a different size than those listed. The OnClick event of this last menu item is handled by the Font1Click method (also connected to a specific menu item), which displays the font selection dialog box. You can see the dynamic menu in Figure 7.7.

**Figure 7.7**
The Size pull-down menu of the DynaMenu example is created at runtime, along with all of its menu items.

**Warning** Because the program uses the Caption of the new items dynamically, you should either disable the AutoHotkeys property of the main menu component, or disable this feature for the pull-down menu you are going to add (and thus automatically disable it for the menu items). This is what we did in the previous code by setting the AutoHotkeys property of the dynamically created pull-down component to Manual. An alternative approach is to let the menu display the automatic captions and then call the StripHotkeys function before converting the caption to a number. There is also a GetHotkey function, which returns the active character of the caption.

The handler for the OnClick event of these dynamically created menu items uses the caption of the Sender menu item to set the size of the font:

```pascal
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
    with Sender as TMenuItem do
    begin
        Label1.Font.Size := StrToInt (Caption);
    end;
end;
```

This code doesn't set the proper radio-item mark next to the selected item, because the user can select a new size by changing the font. The proper radio item is checked in the OnClick event handler of the entire pull-down menu, which is connected just after the pull-down is created and activated just before showing the pull-down. The code scans the items of the pull-down menu (the Sender object) and checks whether the caption matches the current Size of the font. If no match is found, the program checks the last menu item, to indicate that a different size is active:

```pascal
procedure TFormColorText.SizeClick (Sender: TObject);
var
    I: Integer;
```
Found: Boolean;

begin
  Found := False;
  with Sender as TMenuItem do
  begin
    // look for a match, skipping the last item
    for I := 0 to Count - 2 do
      if StrToInt (Items [I].Caption) = Label1.Font.Size then
        begin
          Items [I].Checked := True;
          Found := True;
          System.Break; // skip the rest of the loop
        end;
      if not Found then
        Items [Count - 1].Checked := True;
  end;
end;

When you want to create a menu or a menu item dynamically, you can use the corresponding components, as we did in the DynaMenu examples. As an alternative, you can also use some global functions available in the Menus unit: NewMenu, NewPopupMenu, NewSubMenu,NewItem, and NewLine.

Using Menu Images

In Kylix it is very easy to improve a program’s user interface by adding images to menu items. This is becoming common in many applications, and Borland has kindly added all the required support, making the development of graphical menu items trivial.

All you have to do is add an image list control to the form, add a series of bitmaps to the image list, connect the image list to the menu using its Images property, and set the proper ImageIndex property for the menu items. (You can also associate a single bitmap with the menu item directly, using the Bitmap property.)

To create the image list, double-click the component, activating the corresponding editor (shown in Figure 7.8), and then import existing bitmap or icon files. You can prepare a single large bitmap and let the image editor divide it according to the Height and Width properties of the ImageList component, which refer to the size of the individual bitmaps in the list.

**Figure 7.8**
The ImageList editor with the bitmaps of the MenuImg example
TIP  As an alternative, you can use the series of images that ship with Kylix and are stored by default in the images/buttons subdirectory of the Kylix folder. Each bitmap contains both an enabled and a disabled image. As you import them, the ImageList editor will ask you whether to split them in two, a suggestion you should accept. This operation adds to the image list a normal image and a disabled one, which is not generally used (as it can be built automatically when needed). For this reason, we generally delete the disabled part of the bitmap from the image list.

The program’s code is very simple. The only element we want to emphasize is that if you set the Checked property of a menu item with an image instead of displaying a check mark, the item paints its image as sunken, or recessed. You can see this in the Large Font menu of the MenuImg example in Figure 7.7. Here is the code for that menu item selection:

```pascal
procedure TForm1.LargeFont1Click(Sender: TObject);
begin
  if Memo1.Font.Size = 8 then
    Memo1.Font.Size := 12
  else
    Memo1.Font.Size := 8;
  // changes the image style near the item
end;
```

Qt Styles

Contrary to the Microsoft attitude of deciding what’s good for the user interface, in Qt—and in Linux in general—the users can choose the user interface style of the controls they prefer. The system offers a few basic styles, such as the Windows look-and-feel, the Motif style, and others. A user can also install new styles in the system and make them available to applications.

NOTE  The styles we discuss here refer to the user interface of the controls, not the forms and their borders. This is generally configurable on Linux systems but is technically a separate element from the user interface.

The Application global object of CLX has a Style property, which can be used to set a custom style or a predefined one, indicated by the DefaultStyle subproperty. For example, you can select a Motif look and feel for a program with this code:

```pascal
Application.Style.DefaultStyle := dsMotif;
```

In the StylesDemo program, we added, among various sample controls, a list box with the names of the default styles, as indicated in the TDefaultStyle enumeration, and this code for its OnDblClick event:

```pascal
procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  Application.Style.DefaultStyle := TDefaultStyle (ListBox1.ItemIndex);
end;
```

The effect is that, by double-clicking the list box, you can change the current application style and immediately see its effect onscreen, as demonstrated in Figure 7.9. While predefined styles determine the generic look and feel of the application and follow one of the standard alternatives, custom styles
are generally used to customize the output of one or a few controls within a specific application. As an alternative to custom styles (not covered in this book), you can still use the traditional VCL technique of owner-draw, which is available for list boxes also in CLX, as discussed in the sidebar “Owner-Drawing a List Box.”

**Figure 7.9**
In the StylesDemo program, a user can change the interface style (here you can see a Motif layout).

**Owner-Drawing a List Box**

In VisualCLX, some of the controls, such as ListBoxes and ComboBoxes, surface events very similar to Windows owner-draw events. This is a technique to allow some of the components to customize their output. The native approach of Qt to solve this type of issue is the use of custom styles, demonstrated in the next chapter.

To create an owner-draw list box, we set its Style property to `lbOwnerDrawFixed`, in case every item has the same height (indicated in the ItemHeight property), or `lbOwnerDrawVariable`, in case every item has its own size (indicated at runtime in the OnMeasureItem event).

In the ODList example, we stick with the first, simpler, approach. The example stores color information along with the items of the list box and then draws the items in colors (instead of using a single color for the whole list). The height of the items is determined by calling the TextHeight in the FormCreate method:

```pascal
Canvas.Font := ListBox1.Font;
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

The next thing you have to do is add some items to the list box. Since this is a list box of colors, you need to add color names to the Items of the list box and the corresponding color values to the Objects data storage related to each item of the list. Instead of adding the two values separately, we wrote a procedure to add new items to the list:

```pascal
procedure TODListForm.AddColors (Colors: array of TColor);
var
    I: Integer;
begin
    for I := Low (Colors) to High (Colors) do
        ListBox1.Items.AddObject (ColorToString (Colors[I]), TObject(Colors[I]));
end;
```

Continued on next page
Owner-Drawing a List Box (continued)

This method uses an open-array parameter, an array of an undetermined number of elements of the same type. For each item passed as a parameter, you add the name of the color to the list, and you add its value to the related data by calling the AddObject method. To obtain the string corresponding to the color, you call the ColorToString function. This returns a string containing either the corresponding color constant, if any, or the hexadecimal value of the color. The color data is added to the list box after casting its value to the TObject data type (a 4-byte reference), as required by the AddObject method.

In the ODLList example, this method is called in the OnCreate event handler of the form (after previously setting the height of the items):

```
AddColors ([clRed, clBlue, clYellow, clGreen, clFuchsia, clLime, clPurple, clGray, RGB (213, 23, 123), RGB (0, 0, 0), clAqua, clNavy, clOlive, clTeal]);
```

To compile this code, we added to it the RGB function described earlier in the section “Colors.” The code used to draw the items is not particularly complex. You simply retrieve the color associated with the item, set it as the color of the font, and then draw the text:

```
procedure TODListForm.ListBox1DrawItem(Control: TWinControl; Index: Integer; Rect: TRect; State: TOwnerDrawState);
begin
with Control as TListbox do
begin
  // erase
  Canvas.FillRect(Rect);
  // draw item
  Canvas.Font.Color := TColor (Items.Objects [Index]);
  Canvas.TextOut(Rect.Left, Rect.Top, ListBox1.Items[Index]);
end;
end;
```

The system already sets the proper background color, so the selected item is displayed properly even without any extra code on your part. The example also allows you to add new items by double-clicking the list box. If you try using this capability, you'll notice that some colors you add are turned into color names (one of the CLX color constants), while others are converted into hexadecimal numbers.

What’s Next?

In this chapter, we explored the foundations of VisualCLX, the visual component library available in Kylix for building user interfaces. We discussed the TControl class, its properties, and its most important derived class, TWidgetControl, which encapsulates a Qt object.

We began to explore some of the basic components available in Kylix. These components correspond to the standard Qt controls and some of the so-called common controls. You also learned how to create main menus and pop-up menus and how to add extra graphics to some of these controls.

The next step, however, is to explore in depth the elements of a complete user interface; discuss other common controls, multipage forms, action lists; and discuss technical details of forms. All of these topics will be covered in the next two chapters.