CHAPTER 6

Visual objects

CHAPTER 6

Visual objects

THE BASIC FORM DESIGN SECTION of the previous chapter covered what you need to know about form layout and the design-time behavior of visual controls. It was, however, rather short on code. This chapter is about writing code that deals with controls, forms, and other visual objects.

Controls

Everything that's visible on your applications' windows at run-time is a control. The form's themselves are controls. The buttons, scrollbars, menus, and edit boxes are controls. The control bars and the static text are controls. (See Figure 6-1.)

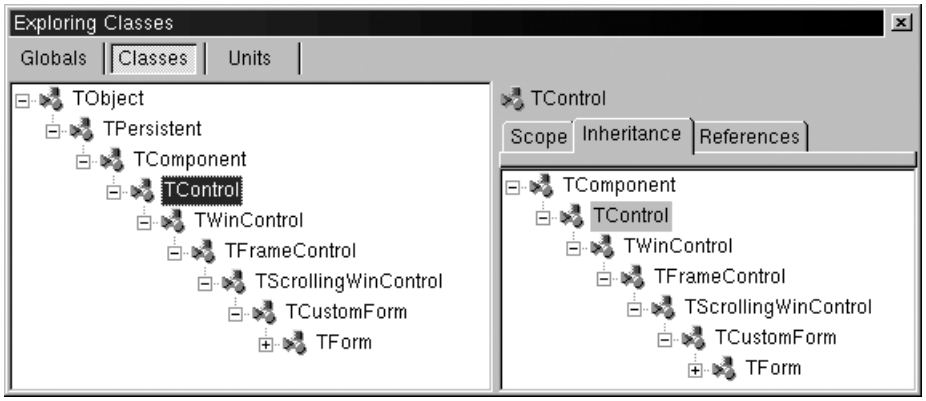


Figure 6-1. Control inheritance

In object terms, every control *is* a component. Every component *is* a persistent object. This means that there are some things that are true of all controls. Obviously, there are also plenty of things that are *not* true of all controls, and these are the details that you'll have to master to work with a particular control, but you'll find it a lot easier to master the specifics once you understand the basics.

Components vs *controls*

I've said it before: All controls are components, but some components are not controls. All components have the published Name and Tag properties; all components have an Owner. The Name property isn't particularly important at run-time—you're much more likely to refer to a component named Joe by the Pascal identifier Joe, a reference to the object, than by the string 'Joe'—but the Owner is. While you'll very rarely write code that explicitly refers to Owner, you implicitly set the Owner every time you create a component at runtime; the AOwner parameter to the Create() constructor becomes the new component's Owner.

It's important to get the Owner right. When a component is freed, it frees all the components it owns. If you pass the wrong Owner to Create(), your component may be freed when you don't expect it. This in turn can lead to segmentation violations as your application tries to refer to memory that has been released.

Kylix is not Delphi

Note

Delphi and Kylix use somewhat different memory management strategies. Delphi uses an internal sub-allocator, which requests relatively large blocks from the operating system, and then splits them into smaller pieces to satisfy requests that are 'too small' to pass on to the OS. This means that dereferencing a 'tombstoned' pointer—one that points to memory that has already been freed–is not guaranteed to cause an exception. Kylix uses an external sub-allocator, that 'lives' in `libc.so.6`*. As with Delphi, dereferencing freed memory may or may not raise an exception. See the* Memory *section of Chapter 8 for more information.*

Fortunately, there are really only two common choices for component ownership, so the chances that you will get it wrong are slim.

1. When dynamically creating a component as part of a form, the Owner should be the form itself. Within an event handler or other form method, you'd refer to the form *via* the `Self` pointer. For example,
   ```
   NewDialog := TOpenDialog.Create(Self);
   ```

2. When manually creating a form, as you might do if it is not used regularly, you would typically set Owner to Application. (See the *TApplication* section of Chapter 7.) This assures that the form is destroyed cleanly, allowing it to close files or do any other necessary cleanup, when the application closes.

Unowned components

There is one common special case where you might set Owner to Nil. Many components and forms are created and destroyed within a single procedure. If an Open File dialog is only occasionally needed, you may prefer to create it on the fly, rather than slow form creation and consume extra system resources by dropping it on the form and having it always available.

```
with TOpenDialog.Create(Nil) do
try
  Title  := 'Dynamic!';
  Filter := 'My files (*.my)|All files (*)';
  if Execute then
    {do something with FileName};
finally
  Free;
end;
```

Similarly, if a dialog like an About box is rarely invoked, you might also prefer to create it on the fly, rather than create it at run-time and/or waste system resources by keeping it around after the user closes it.

```
with TAboutBox.Create(Nil) do
try
  ShowModal;
finally
  Release; // Note that we Release a form, not Free it
end;
```

In both these examples, Owner is set to Nil, by passing Nil to Create. This means that the component has no Owner, and will never be automatically freed. This is obviously risky, so you should only do it where the component is freed in a try/finally block that immediately follows the call to Create. Why might you want to write such risky code? It's an optimization: When you Create an owned component, its Owner adds it to a list of components it owns. When you Free an owned component, the Owner removes it from its list of owned components. Creating and freeing an unowned component is a little faster than creating and freeing an owned component.

Note that components are special in being owned; objects that are **not** components are not owned, and must be explicitly freed.[1] Forms that have object fields (bitmaps, semaphores, and so on) typically Create them in their OnCreate event handler and Free them in their OnDestroy event handler. Non-form objects that have object fields typically Create them in a constructor and Free them in a destructor.

Components property

Every component has a Components array property, and a ComponentCount property. At run-time, the ComponentCount contains the number of components this component owns, and the Components array contains a reference to every component this component owns, indexed from 0 to ComponentCount – 1. Obviously, most components' ComponentCount is 0. A form or a frame's Component's array, however, contains references to every component on the form or frame.

You would write code that walks the Components array if you wanted to do something to every instance of a certain type of component on the form, but didn't want to hardcode the list of which components those were into your application. Walking the list may be slower, but it can be smaller and it is certainly safer. For example, suppose you want to change the color of all the TEdit's on a form in response to various state changes.

```
procedure TGarishFrm.Recolor(Color: TColor);
var
  Index: integer;
begin
  for Index := 0 to ComponentCount - 1 do
    if Components[Index] is TEdit then
      TEdit(Components[Index]).Color := Color;
end; // TGarishFrm.Recolor
```

1. The general rule is "Free what you Create", but there are a number of exceptions:
   - You don't need to free owned components.
   - You Release forms, you don't Free them.
   - You don't need to free objects that you refer to only *via* interface references.

Parent

Anything you can drop on a form is a component, but anything that you can see at run-time is a control. Controls are components that have a number of new methods and properties, including a bounds rectangle—Top, Left, Height, and Width—and a Parent.

A control's Top and Left are always relative to its Parent, whether this is the form itself or a container nested twelve levels deep. (A form doesn't have a Parent; a form's Top and Left is always relative to the desktop work area, which is the part of the screen not covered with system panels and task bars.) A control is only Visible and Enabled when its Parent is Visible and Enabled; if a container has been obscured by another container being brought to front over it, all the controls on the obscured container are hidden as well.

There is a difference between parentage and ownership. All controls on the form are owned by the form, regardless of their parentage. Only some of the controls that a form owns are its direct children; some are children of container controls whose Parent is the form, and some are children of container controls whose Parent is a container control whose Parent is the form, and so on.

When you create a form in the form designer, Kylix will automatically set the Parent of all the controls you drop on it. When you create them at run-time, the library code gets parentage information from the form resource, and properly sets the Parent of all controls on the form. However, if you create a control at run-time, you have to be sure to set its Parent. If you don't set the control's Parent, it won't be visible.

Tip

Not setting Parent when you create a control dynamically is a common mistake made by people new to Kylix.

When you create a control at run-time, its Top and Left will be set to 0, and it will be created with a default Height and Width. (The default varies from control to control.) You will generally either set Align to alClient, to have the new control fill its container, or you will explicitly set both Top and Left, and perhaps Height and Width.

For example, one common use of dynamically created controls is to drop one of several different possible frames into a container like a panel or

a group box. You might need to do this in a property editor for a word processor that allows you to include several different types of objects, like HTML's <ul>, <ol>, <table>, and <script> elements. Each object would be modeled by a different Pascal object, and each Pascal object would be able to tell you what frame to drop into the editing container to edit the object's properties. Thus, you would free any old frame, create the new frame, and set its Parent as

```
FreeAndNil(CurrentFrame); // Safe even if CurrentFrame = Nil
// Using FreeAndNil means that CurrentFrame always holds either a valid value
// or Nil, even if CurrentObject.EditFrame.Create raises an exception.

CurrentFrame := CurrentObject.EditFrame.Create(Self);
// EditFrame returns a "class of TFrame" result

CurrentFrame.Parent  := EditPnl;
CurrentFrame.Visible := True;
// Cheap if already Visible, and necessary if not
```

and you would set the size and position as either

```
CurrentFrame.Align := alClient;
```

or something like

```
CurrentFrame.Top     := Inset; // Some constant
CurrentFrame.Left    := Inset;
CurrentFrame.Height  := EditFrame.ClientHeight - Inset * 2;
CurrentFrame.Width   := EditFrame.ClientWidth  - Inset * 2;
CurrentFrame.Anchors := [akLeft, akRight, akTop, akBottom];
```

Caution

Delphi programmers may be used to doing something similar to this, except with forms instead of frames. This will not work on Linux, due to differences in the windowing model. You'll have to convert any form-in-form code to use frames.

Kylix is not Delphi

Containers

At design-time, there's a strong distinction between containers and other controls. Containers are controls whose constructor adds csAcceptsControls to the ControlStyle set.[2] When you drop a control on a container, the form designer sets the new control's Parent to the container, and sets Top and Left relative to the container. However, if you drop a control on a non-container, you've just created a control that (partially) obscures another control. The non-container doesn't become the new control's Parent.

This distinction is **not** enforced at run-time. If you create a control and set its Parent to a non-container, the new control will appear on top of its new Parent, and will be clipped to it. While this may be what you want in some special cases, it's generally a typo.

There are controls and then there are controls

Just as some components are not visual controls, so some controls are not windowed controls. Where a TWidgetControl (also known as TWinControl, to simplify porting of Delphi code) represents a Qt widget, and is thus a screen window that can have child windows, a TGraphicControl is not. A TGraphicControl exists only as a Kylix object that multiplexes Paint events and translates mouse events into local coordinates. Because a TGraphicControl does not have an underlying screen window, it has no Handle, cannot be the Parent of another control, cannot receive keyboard focus—and it consumes fewer system resources. Speed buttons and bevels are graphic controls.

Most controls, though, are widget controls. Just as every component has a Components property that lists the components it owns, so every widget control has a Controls array and a ControlCount property that lists all its children. At run-time, the ControlCount contains the number of child controls this control has, and the Control array contains a reference to each one, indexed from 0 to ControlCount – 1. Again, most controls' ControlCount is 0: Only forms, frames, and containers like panels have child controls.

2. The ControlStyle set controls various aspects of a component's design-time and run-time behavior. You'll never need to pay any real attention to it except when writing components.

Note

Be sure to note the difference between Controls and Components. Every control on a form is in the form's Components array, but only those controls whose Parent is the form itself are in the form's Controls array. Conversely, controls on a container are in the container's Controls array but not in its Components array.

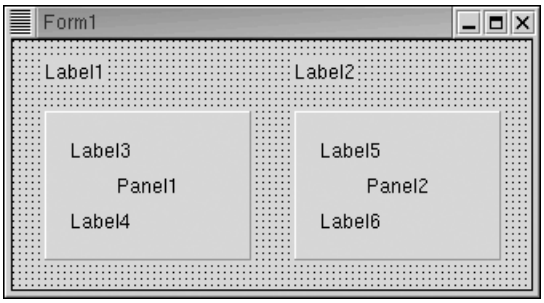


Figure 6-2. Eight components on this form

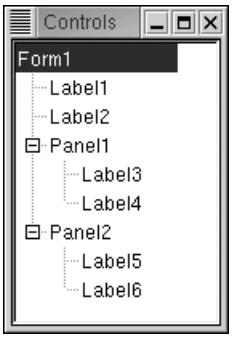


Figure 6-3. The Controls hierarchy

For example, Figure 6-2 shows a small form from the ControlHierarchy project in the ch6/Control project group. This form has two labels placed directly on the form, and two panels, with two more labels each. All eight components are owned by the form, and appear in the form's Components array. However, Panel1 is the Parent of Label3 and Label4, and they appear in Panel1's Controls array. Similarly, Panel2 is the Parent of Label4 and Label5, and they appear in Panel2's Controls array. (See Figure 6-3.)

You would write code that walks a Controls array if you needed to change or examine only the controls in a particular container. For example, this function from lib/QGrabBag.pas (see the RadioButtons project of the ch6/Controls project group for a silly example) returns the Checked radio button on a container control, if any.

```
function CheckedButton(Container: TWidgetControl): TRadioButton;
var
  Index:   integer;
  Control: TControl;
begin
  for Index := 0 to Container.ControlCount - 1 do
  begin
    Control := Container.Controls[Index];
    if Control is TRadioButton then
    begin
      Result := TRadioButton(Control);
      if Result.Checked then EXIT;
    end;
  end;
  Result := Nil; // No Checked radio button on this Container
end; // CheckedButton
```

Mouse events

All controls can respond to mouse events. Usually, of course, you only care about the standard response to mouse events—whether the button was pushed, or the selection changed, and so on—but sometimes you care about where on the control the button was pushed, which button was pushed, whether any shift keys were held down, and so on. For example, you might want to synthesize a shift-click event. Or a graphical editor for music notation, org charts, or circuit diagrams would need to be able to know where on the control the mouse was pressed, so it could translate that to a click on a particular piece of data.

In either case, you would begin by handling the OnMouseDown event for the graphical editor, which might be a TPaintBox. The OnMouseDown events is a

```
TMouseEvent = procedure(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer) of object;
```

You'll get this event on right clicks and middle clicks as well as left clicks, so your first task is to check which Button is generating the event. If it's the button you want, you may or may not care to check the Shift state; some apps distinguish between a shift-click and an unshifted-click, while others don't.

Kylix is not Delphi

Note

If you do examine the Shift state, be sure to do so defensively: Write tests like `if ssShift in Shift` *rather than* `if Shift = [ssShift]` *or* `if Shift = [ssShift, ssLeft]`. *While both Qt documentation and the QControls.pas source code suggest that,* eg, *a left SHIFT+CLICK should generate a MouseDown event with* `Shift = [ssShift, ssLeft]`, *what you actually get is just* `[ssShift]`. *The MouseUp event gets the Shift = [ssShift, ssLeft]. This directly reflects the information in X11 mouse button up and down events; it's just a tad … unfortunate … that it's exactly the opposite of the way Delphi acts. Note, though, that Kylix's mouse events* ***do*** *act like Delphi's in that they do show "chording". That is, if you hold down the left button and press the right button, the OnMouseDown for the right button press will show* `ssLeft in Shift`.

To synthesize a left click, you'd just need to check Button and Shift. In the graphical editor example, if the Button and the Shift tell you that this is the event you care about, you also need to check the X and Y parameters to make sure that the user clicked on a piece of data, not on white space. Fortunately, the X and the Y parameters are both in client coordinates of the control receiving the event, so you just have to walk a list of the data you're drawing on the screen, seeing if the click point is in any of its bounding boxes.

Tip

Kylix includes versions of the familiar Windows rectangle functions: PtInRect, IntersectRect, UnionRect, IsRectEmpty, and OffsetRect.

Of course, a mouse down is only half of a click. The user also has to release the mouse on the control she pressed it on, or it doesn't count. This poses a problem: Just as your control only gets a mouse down event when the mouse pressed over the control, so it normally only gets a mouse up event when the mouse is released over the control. The solution is "mouse capture", a request for **all** mouse events, until you release the capture.

If you're writing a custom component, you can use the protected MouseCapture property. When you set it to True, the control has mouse capture until you set it False. This option is not available to you when you're writing an event handler, so you have to set Mouse.Capture to the control you want to own the mouse. Whichever approach you take, be sure to free the mouse when you're done!

Note

*If you're writing a custom component that wants to capture the mouse on **every** mouse down event, your component's constructor can add csCaptureMouse to the component's ControlStyle. This automatically captures the mouse whenever it's clicked on your component, and automatically 'sets the mouse free' when the button is released.*

So, the following two schematic events are what you need to create your own customized click events:

```
procedure TSynthesizer.ControlMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if ThisIsTheRightClick then
    Mouse.Capture := CapturingControl;
end; // TSynthesizer.ControlMouseDown
```

```
procedure TSynthesizer.ControlMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Mouse.Capture = CapturingControl then
  begin
    Mouse.Capture := Nil;
    if ThisIsTheRightRelease then
      SyntheticClickAction;
  end;
end; // TSynthesizer.ControlMouseUp
```

The ThisIsTheRightRelease test would check that the mouse cursor was in the right rectangle. For a GUI editor, you'd need to check the bounding box of the datum that was clicked on. For the simpler shift-click synthesis, you'd just need to be sure that the click point is still in the control's bounding box: `PtInRect(Rect(0, 0, Width, Height), Point(X, Y))`.

Note

While the OnMouse events include the cursor position, the OnClick event is a simple TNotifyEvent—`procedure (Sender: TObject) of object`—*that doesn't include click location information. When you need that, you can read the* `Mouse.CursorPos` *property. (See Chapter 7 for more information about the* `Mouse` *global.)*

Keyboard events

As with mouse events, you have a choice between 'raw' and 'cooked' keyboard events. Most controls have an OnKeyPress event, which gives you the keypress info cooked to a standard ANSI character. The cooked character is perfectly adequate for entering filenames, email, or program source, but if you need to respond to non-character keys, like the cursor movement keys, function keys, or combinations like SHIFT+CTRL+INSERT, you need to handle the raw keyboard events, OnKeyDown and/or OnKeyUp.

These events pass you both a shift state map, Shift, and a numeric key code, Key. For the standard Latin-1 characters (*ie*, #32 to #255), the Key code is the Ord() of the character. (Alphabetic keystrokes are all implicitly uppercase. That is, if the B key is pressed, Key will equal Ord('B'), or 66.) The keys that produce Latin-1 characters are also named in Qt.pas, with the English alphabet keys being Key_A through Key_Z and the 'European' alphabet keys having longer names. See the sidebars on pages 358 and 361 for Key_ names for punctuation and 'European' alphabet characters.

Kylix is not Delphi

Any Delphi code that uses Windows' VK_ key names will have to be rewritten to use Qt's Key_ names.

For keys like F1 through F12, the cursor keys, functions keys, pad, and the other non-visual keys, you have to rely on the Key_ names. These are summarized in the sidebar on page 359. Strangely, you can distinguish between the two Enter keys, but you cannot distinguish between the left and right Shift, Ctrl, or Alt keys, nor can you distinguish between 'cursor pad' and 'number pad' keys. (You can tweak your `xmodmap` settings a bit so that you can distinguish these keys, but obviously you can't rely on a user's having a customized `/etc/X11/XModmap` unless you're selling turnkey systems.)

Note that the (English) alphabet keys are handled a bit differently than non-alphabet keys. When you press the B key, you get a Key_B event, and `ssShift` is not `in Shift`. When you hold down the shift key and press the B key, you get a Key_Shift event, followed by a Key_B with `ssShift in Shift`. That is, you get the same Key_ code for 'b' and 'B'. However, when you press the 6 key you get a Key_6 event, while SHIFT+6 gives you a Key_Shift event, followed by a Key_AsciiCircum. That is, the key codes have been translated to match the glyph's on the user's keyboard.

The KbdLookup project in the ch6/Controls project group will let you explore these behaviors on your own.

KeyPreview

Keystrokes go to the control with keyboard focus. This means that, by default, form-wide keyboard event handlers will never fire. If you set the form's KeyPreview property to True, the form will 'see' keyboard events before the event with the keyboard focus.

Setting the Key code to 0 suppresses further processing. If you do this in an OnKeyDown handler, for example, you'll still get an OnKeyUp event when the key is released, but you will **not** get an OnKeyPress.

Table 6-1. ASCII punctuation and Key_ codes

Glyph	Decimal	Hex	Key_ code
	32	20	Key_Space, KeyAny
!	33	21	Key_Exclam
"	34	22	Key_QuoteDbl
#	35	23	Key_NumberSign
$	36	24	Key_Dollar
%	37	25	Key_Percent
&	38	26	Key_Ampersand
'	39	27	Key_Apostrophe
(	40	28	Key_ParenLeft
)	41	29	Key_ParenRight
*	42	2A	Key_Asterisk
+	43	2B	Key_Plus
,	44	2C	Key_Comma
-	45	2D	Key_Minus
.	46	2E	Key_Period
/	47	2F	Key_Slash
:	58	3A	Key_Colon
;	59	3B	Key_Semicolon
<	60	3C	Key_Less
=	61	3D	Key_Equal
>	62	3E	Key_Greater
?	63	3F	Key_Question
@	64	40	Key_At
[	91	5B	Key_BracketLeft
\	92	5C	Key_Backslash
]	93	5D	Key_BracketRight

Table 6-1. ASCII punctuation and Key_ codes (Continued)

Glyph	Decimal	Hex	Key_ code
^	94	5E	Key_AsciiCircum
_	95	5F	Key_Underscore
`	96	60	Key_QuoteLeft
{	123	7B	Key_BraceLeft
|	124	7C	Key_Bar
}	125	7D	Key_BraceRight
~	126	7E	Key_AsciiTilde

Table 6-2. Non-visible characters and Key_ codes

Key_ code	Produced by
Key_Escape	Esc
Key_Tab	Tab
Key_Backtab	Shift+Tab
Key_Backspace	Backspace
Key_Return	'Normal' Enter
Key_Enter	Number pad Enter
Key_Insert	Insert
Key_Delete	Delete
Key_Pause	Pause / Break
Key_Print	Print Screen / SysRq
Key_SysReq	?
Key_Home	Home
Key_End	End
Key_Left	'Inverted T' left
Key_Up	'Inverted T' up
Key_Right	'Inverted T' right

Table 6-2. Non-visible characters and Key_ codes (Continued)

Key_ code	Produced by
Key_Down	'Inverted T' down
Key_Prior	PgUp
Key_PageUp	PgUp
Key_Next	PgDn
Key_PageDown	PgDn
Key_Shift	Right or left Shift
Key_Control	Right or left Ctrl
Key_Meta	Shift then Alt, but not Alt then Shift. On some–but not all–distributions, the Windows key produces Key_Meta.
Key_Alt	Right or left Alt
Key_CapsLock	Caps Lock
Key_NumLock	Num Lock
Key_ScrollLock	Scroll Lock
Key_F1	F1 (etc)
Key_Super_L	?
Key_Super_R	?
Key_Menu	?
Key_Hyper_L	?
Key_Hyper_R	?
Key_Help	?
Key_unknown	Number pad 5

Notes:

1. Keys “Produced by” ? are not produced on a standard US 104-key keyboard with a standard Redhat 7.0 `/etc/X11/Xmodmap`. Other distributions may differ.

2. Though you can distinguish between the two Enter keys, you cannot reliably distinguish between the left and right Shift, Ctrl, or Alt keys, nor can you distinguish between 'cursor pad' and 'number pad' keys. This depends on `xmodmap`, and varies from system to system and from distribution to distribution.

3. Under Gnome, at least, the three Windows keys come through as Key code 0 by default. I edited my `/etc/X11/Xmodmap` so I could assign the three Windows keys to Gnome events: Don't expect to be able to detect these keys.

Table 6-3. Latin-1 'European' characters and Key_ codes

Description	Glyph	Decimal	Hex	HTML name	Key_ code
Non-breaking space		160	A0	nbsp	Key_nobreakspace
Inverted exclamation mark	¡	161	A1	iexcl	Key_exclamdown
Cent sign	¢	162	A2	cent	Key_cent
Pound sign	£	163	A3	pound	Key_sterling
Currency sign	¤	164	A4	curren	Key_currency
Yen sign	¥	165	A5	yen	Key_yen
Broken vertical bar	¦	166	A6	brvbar	Key_brokenbar
Section sign	§	167	A7	sect	Key_section
Spacing diaresis	¨	168	A8	uml	Key_diaeresis
Copyright sign	©	169	A9	copy	Key_copyright
Feminine ordinal indicator	ª	170	AA	ordf	Key_ordfeminine
Angle quotation mark, left	«	171	AB	laquo	Key_guillemotleft
Negation sign	¬	172	AC	not	Key_notsign

Table 6-3. Latin-1 'European' characters and Key_ codes (Continued)

Description	Glyph	Decimal	Hex	HTML name	Key_ code
Soft hyphen	-	173	AD	shy	Key_hyphen
Circled R registered sign	®	174	AE	reg	Key_registered
Spacing macron	¯	175	AF	hibar	Key_macron
Degree sign	°	176	B0	deg	Key_degree
Plus-or-minus sign	±	177	B1	plusmn	Key_plusminus
Superscript 2	²	178	B2	sup2	Key_twosuperior
Superscript 3	³	179	B3	sup3	Key_threesuperior
Spacing acute	´	180	B4	acute	Key_acute
Micro sign	µ	181	B5	micro	Key_mu
Paragraph sign	¶	182	B6	para	Key_paragraph
Middle dot	·	183	B7	middot	Key_periodcentered
Spacing cedilla	¸	184	B8	cedil	Key_cedilla
Superscript 1	¹	185	B9	sup1	Key_onesuperior
Masculine ordinal indicator	º	186	BA	ordm	Key_masculine
Angle quotation mark, right	»	187	BB	raquo	Key_guillemotright
Fraction 1/4	¼	188	BC	frac14	Key_onequarter
Fraction 1/2	½	189	BD	frac12	Key_onehalf
Fraction 3/4	¾	190	BE	frac34	Key_threequarters
Inverted question mark	¿	191	BF	iquest	Key_questiondown
Capital A, grave accent	À	192	C0	Agrave	Key_Agrave
Capital A, acute accent	Á	193	C1	Aacute	Key_Aacute

Table 6-3. Latin-1 'European' characters and Key_ codes (Continued)

Description	Glyph	Decimal	Hex	HTML name	Key_ code
Capital A, circumflex accent	Â	194	C2	Acirc	Key_Acircumflex
Capital A, tilde	Ã	195	C3	Atilde	Key_Atilde
Capital A, dieresis or umlaut mark	Ä	196	C4	Auml	Key_Adiaeresis
Capital A, ring	Å	197	C5	Aring	Key_Aring
Capital AE diphthong (ligature)	Æ	198	C6	AElig	Key_AE
Capital C, cedilla	Ç	199	C7	Ccedil	Key_Ccedilla
Capital E, grave accent	È	200	C8	Egrave	Key_Egrave
Capital E, acute accent	É	201	C9	Eacute	Key_Eacute
Capital E, circumflex accent	Ê	202	CA	Ecirc	Key_Ecircumflex
Capital E, dieresis or umlaut mark	Ë	203	CB	Euml	Key_Ediaeresis
Capital I, grave accent	Ì	204	CC	Igrave	Key_Igrave
Capital I, acute accent	Í	205	CD	Iacute	Key_Iacute
Capital I, circumflex accent	Î	206	CE	Icirc	Key_Icircumflex
Capital I, dieresis or umlaut mark	Ï	207	CF	Iuml	Key_Idiaeresis

Table 6-3. Latin-1 'European' characters and Key_ codes (Continued)

Description	Glyph	Decimal	Hex	HTML name	Key_ code
Capital Eth, Icelandic	Ð	208	D0	ETH	Key_ETH
Capital N, tilde	Ñ	209	D1	Ntilde	Key_Ntilde
Capital O, grave accent	Ò	210	D2	Ograve	Key_Ograve
Capital O, acute accent	Ó	211	D3	Oacute	Key_Oacute
Capital O, circumflex accent	Ô	212	D4	Ocirc	Key_Ocircumflex
Capital O, tilde	Õ	213	D5	Otilde	Key_Otilde
Capital O, dieresis or umlaut mark	Ö	214	D6	Ouml	Key_Odiaeresis
Multiplication	×	215	D7		Key_multiply
Capital O, slash	Ø	216	D8	Oslash	Key_Ooblique
Capital U, grave accent	Ù	217	D9	Ugrave	Key_Ugrave
Capital U, acute accent	Ú	218	DA	Uacute	Key_Uacute
Capital U, circumflex accent	Û	219	DB	Ucirc	Key_Ucircumflex
Capital U, dieresis or umlaut mark	Ü	220	DC	Uuml	Key_Udiaeresis
Capital Y, acute accent	Ý	221	DD	Yacute	Key_Yacute
Capital THORN, Icelandic	Þ	222	DE	THORN	Key_THORN

Table 6-3. Latin-1 'European' characters and Key_ codes (Continued)

Description	Glyph	Decimal	Hex	HTML name	Key_ code
Small sharp s, German (sz ligature)	ß	223	DF	szlig	Key_ssharp
Small a, grave accent	à	224	E0	agrave	Key_agrave_lower
Small a, acute accent	á	225	E1	aacute	Key_aacute_lower
Small a, circumflex accent	â	226	E2	acirc	Key_acircumflex_lower
Small a, tilde	ã	227	E3	atilde	Key_atilde_lower
Small a, dieresis or accent	ä	228	E4	auml	Key_adiaeresis_lower
Small a, ring	å	229	E5	aring	Key_aring_lower
Small ae diphthong (ligature)	æ	230	E6	aelig	Key_ae_lower
Small c, cedilla	ç	231	E7	ccedil	Key_ccedilla_lower
Small e, grave accent	è	232	E8	egrave	Key_egrave_lower (grave_lower)
Small e, acute accent	é	233	E9	eacute	Key_eacute_lower (acute_lower)
Small e, circumflex (ligature)	ê	234	EA	ecirc	Key_ecircumflex_lower (circumflex_lower)
Small e, dieresis or umlaut mark	ë	235	EB	euml	Key_ediaeresis_lower (diaeresis_lower)
Small i, grave accent	ì	236	EC	igrave	Key_igrave_lower
Small i, acute accent	í	237	ED	iacute	Key_iacute_lower

Table 6-3. Latin-1 'European' characters and Key_ codes (Continued)

Description	Glyph	Decimal	Hex	HTML name	Key_ code
Small i, circumflex umlaut mark	î	238	EE	icirc	Key_icircumflex_lower
Small i, dieresis or umlaut mark	ï	239	EF	iuml	Key_idiaeresis_lower
Small eth, Icelandic	ð	240	F0	eth	th_lower
Small n, tilde	ñ	241	F1	ntilde	Key_ntilde_lower
Small o, grave accent	ò	242	F2	ograve	Key_ograve_lower
Small o, acute accent	ó	243	F3	oacute	Key_oacute_lower
Small o, circumflex umlaut mark	ô	244	F4	ocirc	Key_ocircumflex_lower
Small o, tilde	õ	245	F5	otilde	Key_otilde_lower
Small o, dieresis or umlaut mark	ö	246	F6	ouml	Key_odiaeresis_lower
Division	÷	247	F7		Key_division
Small o, slash	ø	248	F8	oslash	Key_oslash
Small u, grave accent	ù	249	F9	ugrave	Key_ugrave_lower
Small u, acute accent	ú	250	FA	uacute	Key_uacute_lower
Small u, circumflex accent	û	251	FB	ucirc	Key_ucircumflex_lower
Small u, dieresis or umlaut mark	ü	252	FC	uuml	Key_udiaeresis_lower

Table 6-3. Latin-1 'European' characters and Key_ codes (Continued)

Description	Glyph	Decimal	Hex	HTML name	Key_ code
Small y, acute accent	ý	253	FD	yacute	Key_yacute_lower
Small thorn, Icelandic	þ	254	FE	thorn	Key_thorn_lower
Small y, dieresis or umlaut mark	ÿ	255	FF	yuml	Key_ydiaeresis

Note

The key codes for the characters from 232 to 235 ('decorated' lower case e's) are declared in Qt.pas without the leading Key_e. Eg, *232 is declared as* `grave_lower`, *not* `Key_egrave_lower`. *This may be fixed in later releases, or in a patch to release 1.*

Owner draw

Some controls have events that let you draw them—or parts of them—yourself. These "owner draw" controls include list and combo boxes, string and draw grids, tab and page controls, and list, icon, and tree views. All of these except the draw grid **can** draw themselves; owner draw is an override, for when you want to add a state graphic or custom font.

The date picker component of Chapter 9 uses an owner draw grid for a calendar. The OwnerDraw project in the ch6/Controls project group (Figure 6-4) is a simpler example, with an owner draw dropdown that uses each font name as an example of itself. If you run that program, you'll see that it's actually a rather bad example—rendering each font means that the dropdown is very slow—but it's a **simple** example, as I don't have to obscure the logic with any bogus state generation. The key routine is this FontsDrawItem event handler, which draws each font name in itself.

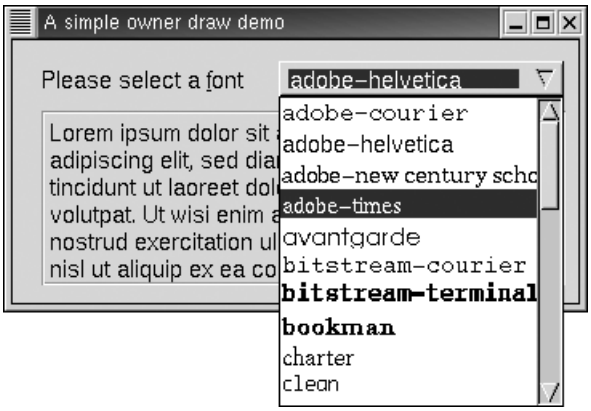


Figure 6-4. An owner draw dropdown

```
procedure TOwnerDrawFrm.FontsDrawItem(Sender: TObject; Index: Integer;
  Rect: TRect; State: TOwnerDrawState; var Handled: Boolean);
var
  Dropdown: TComboBox;
  ThisFont: string;
begin
  Assert(Sender is TComboBox);
  Dropdown := TComboBox(Sender);

  with Dropdown.Canvas do
  begin
    ThisFont  := Dropdown.Items[Index];
    Font.Name := ThisFont;
    TextRect(Rect, Rect.Left + 1, Rect.Top + 1, ThisFont);
  end;
  Handled := True;
end; // TOwnerDrawFrm.FontsDrawItem
```

This should seem pretty straightforward.[3] First, I check that the Sender is indeed a TComboBox, and cast the Sender to a TComboBox. Then, I 'open' DropDown.Canvas (see the *Canvas* section of this chapter, below) with a `with` statement. The Index parameter tells me which item in the combo box's Items list I need to draw (see Chapter 7 for a discussion of Items' TStrings class), so I make a local copy of the string to avoid having to index Dropdown.Items twice. `Font.Name := ThisFont` sets `DropDown.Canvas.Font` to the current font—that's

3. It's sure lucky I don't have to pay royalties on the phrase "pretty straightforward", isn't it?

all I need to do to change fonts. Then, I call TextRect to draw the font name string, clipped to the Rect that was passed in. Note that this Rect is in the Sender's coordinates, not form or screen coordinates.

Owner draw vs *application styles*

Owner draw relies on the form's event handlers to do the custom drawing. While you can easily have two or more controls on the same form share an event handler, sharing event handler code across forms is more cumbersome. You can push the code into a routine that's called by different event handlers. Or, you can try risky tricks like setting the event handler for a control on *this* form to an event handler on *that* form in *this* form's OnCreate handler. Basically, though, owner draw controls are best restricted to displaying custom data.

If you want to give your application a distinctive look and change the way **all** buttons or all list boxes look, you should use Application.Style. See the *Application* section of Chapter 7.

Drag and drop

Kylix makes it easy for you to add drag and drop to your applications. All controls[4] have a DragMode property and four drag events: OnStartDrag, OnDragOver, OnDragDrop, and OnEndDrag. Normally, DragMode is dmManual, which means that you have to initiate drag and drop activity by calling BeginDrag, typically in an OnMouseDown handler. If you set DragMode to dmAutomatic, StartDrag is called automatically whenever the user drags the mouse more than Mouse.DragThreshold pixels from the click point.

OnDragOver

Once you start dragging a control, the two key events are OnDragOver and OnDragDrop. OnDragOver is called often as you drag a control over a form or over other controls. OnDragOver is called when the cursor enters a control's bounding box; when the cursor leaves a control's bounding box; and whenever the cursor moves about inside of a control's bounding box. The point of the OnDragOver event is setting its Accept parameter to True or False. If you drag a control over a control that doesn't have an OnDragOver event handler, or if

4. Well, "almost all". TBevel, for instance, doesn't have a DragMode property and drag events.

that handler sets Accept to False, the cursor will (by default—you can set your own cursors) be a slashed circle: You can't drop here. (See Figure 6-5.) If the OnDragOver event handler sets Accept to True, the cursor will (again, only by default) be a normal drag cursor. (See Figure 6-6.)

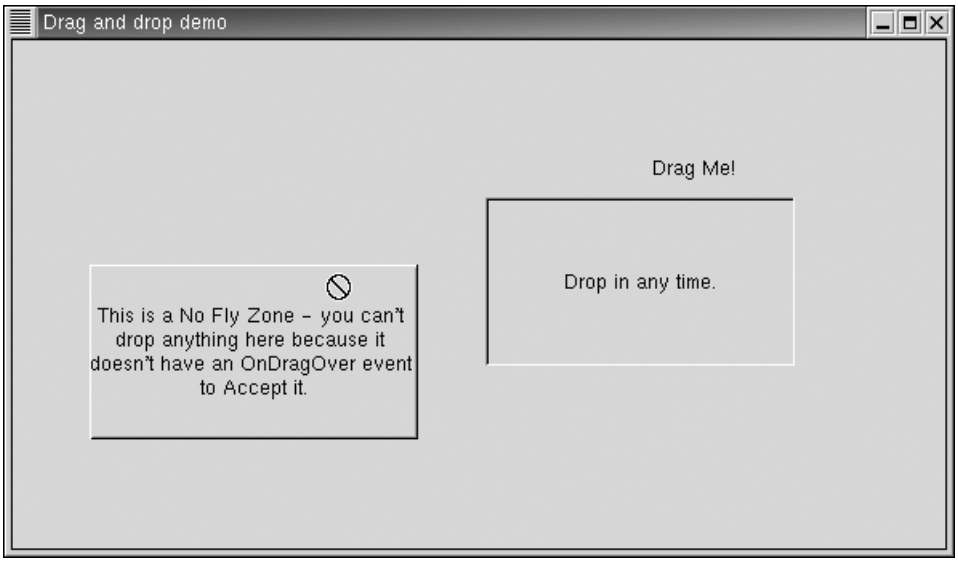


Figure 6-5. Can't drop here

The OnDragOver event handler has, as usual, a Sender parameter. It also has a Source parameter, which is also a TObject. The Sender is, as usual, the control that's sending the event, the one the event handler belongs to. In other words, the Sender is the control that's being dragged over. The Sender is being asked whether or not to Accept the control that's being dragged over it, the Source.

Real apps might have relatively complex tests to determine which controls they'll accept and where, but the DragAndDrop project of the ch6/Controls project group is very simplistic. It will Accept anytime Sender <> Source. (If it blindly sets Accept to True, it could end up trying to drop a control onto itself.)

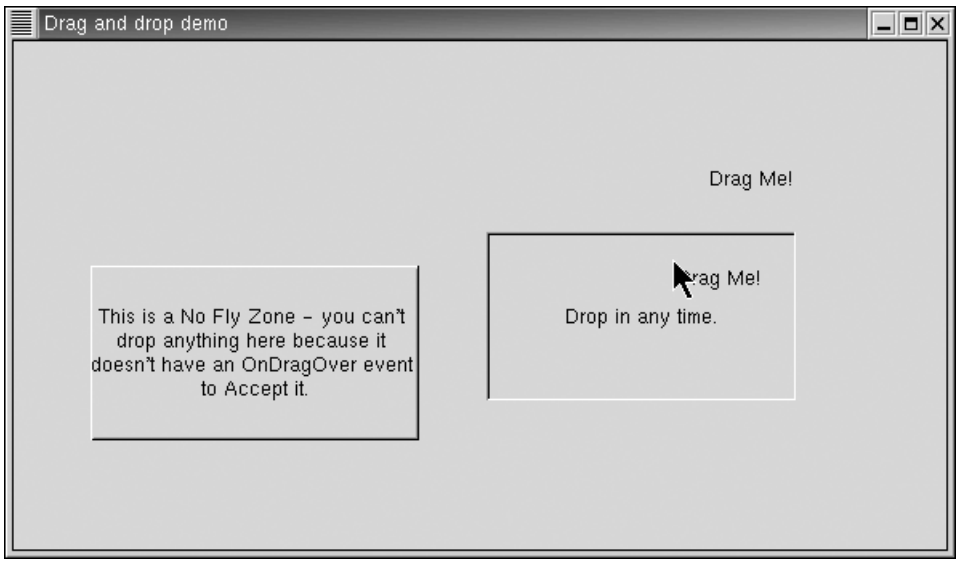


Figure 6-6. Can drop here

```
procedure TDragDropDemoFrm.VeryAccepting(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := Sender <> Source;
end; // TDragDropDemoFrm.VeryAccepting
```

If you run the DragAndDrop project, you'll see that you can drag all three labels, but that you can't drop onto the 'No Fly Zone' with the raised wall around it. That's because the form and the DropIn label share the above VeryAccepting OnDragOver event handler, but the NoFlyZone label does not have an OnDragOver event handler.

OnDragDrop

When the user drops a control over a control that will accept it, the receiving control gets an OnDragDrop event. If it doesn't handle this event, nothing will happen except that the cursor changes back to normal. Kylix mediates a protocol for you, but it's up to you to make anything happen. In the OnDragDrop event, as in the OnDragOver event, the Sender is the control whose event handler is being fired, the drop target, while Source is the control that's being

dropped, and X and Y are the drop location in Sender's coordinates. Thus, the DragAndDrop project moves controls where you drop them:

```
procedure TDragDropDemoFrm.ReparentOnDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  Assert(Sender is TWidgetControl, 'Sender is ' + Sender.ClassName);
  Assert(Source is TControl,       'Source is ' + Source.ClassName);
  with TControl(Source) do
  begin
    Parent := TWidgetControl(Sender);
    Left := X;
    Top  := Y;
  end;
end; // TDragDropDemoFrm.ReparentOnDrop
```

If you run the project, you'll see that you can drop the DragMe label onto the DropIn label; if you then drag the DropIn label, the DragMe label will move with it. If you drag the DragMe label off the DropIn label, it will no longer move with the DropIn label.

Of course, a real project would typically have a more complicated OnDragDrop event handler than this one. For example, you don't have to use drag and drop to move controls; you might drag and drop the 'contents' of one control into another, or you might use drag and drop to let users reorder a list.

TDragObject

The default drag cursor isn't very enlightening. It doesn't show **what** you're dragging, and it doesn't show **where** the cursor is in relationship to the object you're dragging. The OnStartDrag event lets you create a TDragObject descendant that can help with this. If you don't set the event handler's DragObject parameter, Kylix will automatically create a TDragObject for you that produces the default cursor &c. However, if you create a customized descendant of TDragObject and assign it to the DragObject parameter, you can specify the image that's being dragged as well as its "hot spot", which is the point in image coordinates that should be under the cursor as it's dragged. This lets you both tailor the drag image to what's being dragged, and to give some indication of where it will be placed when dropped. If you create a TDragObject, be sure to Free it in an OnEndDrag event handler.

The DragAndDrop project sets DragObject for the DragMe label, but not for either of the two bordered labels. This is why dragging DragMe shows the text being dragged, but dragging the other labels just shows the default drag cursor.

The DragMe label's OnStartDrag handler creates a specialized descendant of TDragControlObject, which is what you'd typically use for simply dragging a whole control. When you use a TDragControlObject, the Source parameters of the OnDragDrop and OnDragOver events are the control you're dragging, just as if you hadn't set a DragObject. If you use a drag object that descends directly from TDragObject, the Source parameter is the TDragObject itself, which is a bit more complicated but can simplify things when several different controls might be drag sources. They'd all create the same type of DragObject, which the OnDragDrop and OnDragOver events would know what to do with.

The TextDragObject I create in the DragAndDrop project doesn't do anything so subtle, but it does illustrate how to create a custom drag cursor. The constructor saves the Source label's Caption in a Text field, and the GetDragImageIndex method adds an image to the global DragImageList and returns the index of the new image.

```
function TextDragObject.GetDragImageIndex: Integer;
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    with DragImageList do
    begin
      Bitmap.Height := Height; // just so we have Canvas &c
      Bitmap.Width  := Width;
      Width  := Max(Width,  Bitmap.Canvas.TextWidth(Text));
      Height := Max(Height, Bitmap.Canvas.TextHeight(Text));
                // The Max() fn is in the Math unit
      Bitmap.Height := Height;
      Bitmap.Width  := Width;
      Bitmap.Canvas.TextOut(0, 0, Text);
      Result := DragImageList.Add(Bitmap, Nil);
      ImageIndex := Result;
    end;
  finally
    Bitmap.Free;
  end;
end; // TextDragObject.GetDragImageIndex
```

This function is a bit longer than some of the others I've presented so far, but it should be obvious enough. I create a TBitmap (see below) to draw on, because we can Add a bitmap to an image list, or Replace an existing one, but we can't directly modify an image in an image list. I set the new bitmap's Height and Width to the image list's current Height and Width, so that I have a valid Canvas which can calculate TextHeight and TextWidth. I resize the global drag image list so it will fit the text image I want to Add to it, then draw text on the bitmap with TextOut. I then Add the bitmap, which gives me the image index I need to return, and save that index in a field of the TextDragObject so that I can remove it from the global drag image list in the TextDragObject's destructor. Once I've added the bitmap to the image list, I no longer need the bitmap, so I Free the bitmap, and return.

In the OnStartDrag handler, I create the TextDragObject and assign it to the DragObject parameter. Kylix will not automatically Free a DragObject that we explicitly Create, so I save a reference to the DragObject in a field of the form object, and Free it in the OnEndDrag event handler.

```
procedure TDragDropDemoFrm.StartTextDrag(Sender: TObject;
  var DragObject: TDragObject);
begin
  Assert(Sender is TLabel);
  DragObject := TextDragObject.Create( TLabel(Sender),
                                       TLabel(Sender).Caption);
  fDragObject := DragObject; // 'cause Kylix doesn't Free it by itself
end; // TDragDropDemoFrm.StartTextDrag

procedure TDragDropDemoFrm.EndTextDrag(Sender, Target: TObject; X,
  Y: Integer);
begin
  FreeAndNil(fDragObject);
end; // TDragDropDemoFrm.EndTextDrag
```

Canvas

I've used a Canvas a few times now, and it should be obvious what's going on: A Canvas is a drawing surface. Forms have a Canvas property, as do bitmaps and all controls with owner-draw abilities. Container controls typically don't have a Canvas property, but you can drop a PaintBox component (from the Additional tab) onto a container if you need to draw on it.

Note

Paint boxes and other graphic controls are drawn before—or, visually underneath—windowed controls like labels and buttons.

Canvases have three basic drawing tools—a pen, a brush, and a font—and a variety of basic two-dimensional graphics primitives.

Drawing tools

Designers of graphics primitives have two basic choices: They can fully parameterize every call, so that drawing a line requires you to specify color and line style along with end points, or they can use a concept of "drawing tools", so that before you can draw a line you have to setup the tools, but then you only have to specify a pair of end points. The fully parameterized approach makes for fewer system calls, at the expense of each call being more complex. The drawing tools approach requires system calls to configure the tools, but each call is simpler—and faster overall, as often you will draw several items using the same border, interior, or font.

Kylix's Canvas takes the drawing tools approach. Primitives that draw a line or outline a region use the Canvas' Pen, which has Color, Width, and a couple of drawing mode properties. Primitives that fill a region use the Canvas's Brush, which has Color, Bitmap, and Style properties. Primitives that draw text use the Canvas' Font, which has the same font Name, Color, Size, and Style properties that you've seen already.

Drawing tools Assign() on assignment

A Canvas' Pen, Brush, and Font tools are all objects in their own right. This means that when you want to set a Canvas's properties, you can set the various tools' properties one by one, or you can simply write code like `ThisCanvas.Pen := ThatPen` to change all the Canvas' Pen's properties to match ThatPen's.

Does this look dangerous to you? Like you've changed ThisCanvas's Pen property to point to a new object, without freeing the old one? Or that ThatPen might now have two 'owners' who might think that they ought to Free it?

Good.

The assignment is safe, though, because CLX objects that expose drawing tool objects do so *via* `write` methods like

```
procedure TCanvas.SetPen(const Value: TPen);
begin
  FPen.Assign(Value);
end;
```

TFont, TPen, and TBrush all descend from TGraphicsObject, which in turn descends directly from TPersistent (see Chapter 7). This lets them stream in from a form resource; it also gives them an Assign() method. The default Assign() implementation in TPersistent just raises an exception, but descendants override it to make themselves functionally identical to the object they're Assign()ed. Thus, when you say `ThisCanvas.Pen := ThatPen`, pointer(ThisCanvas.Pen) does not change; rather, the Canvas's Pen object copies ThatPen's Color, Mode, Style, and Width properties.

You can—and should—use this technique in your own objects that expose object properties.

```
property Pen: TPen read fPen write fPen;
```

can lead to memory leaks and/or crashes, while

```
property Pen: TPen read fPen write SetPen;
```

where SetPen does an Assign(), is safe and convenient.

Pen

You can set a Pen's Color to a TColor. There are many named colors (available in the Object Inspector's dropdown for a Color property) which can be grouped broadly into "absolute" colors like clRed, clBlue, and clBlack, and "functional" colors like clButton, clShadow, and clBase. In addition to these named colors, you can use any 32-bit integer from 0 to $00FFFFFF as a RGB value. The standard function ColorToRGB will convert a named TColor to its RGB values.

Kylix's RGB encoding is "little endian"—the low (or first) byte is the Red component, the second byte is the Green component and the third byte is the Blue component—and thus the reverse of HTML's more hexadecimal friendly RRGGBB. The following functions from my lib/QGrabBag may be useful:

```
function RGB(Red, Green, Blue: integer): TColor;
const
  Mask = $000000FF; // Mask off all but low-byte
begin
  Result := Red   and Mask         or
            Green and Mask shl  8  or
            Blue  and Mask shl 16  ;
end; // RGB

function TColorToHtml(Color: TColor): string;
begin
  Result := IntToHex( Color and $00FF0000 shr 16 or
                      Color and $0000FF00        or
                      Color and $000000FF shl 16, 6 );
end; // TColorToHtml

function HtmlToTColor(const Color: string): TColor;
begin
  Result := StrToInt('$' + Color);
  Result := Result and $00FF0000 shr 16 or
            Result and $0000FF00        or
            Result and $000000FF shl 16 ;
end; // HtmlToTColor
```

It should be obvious what these functions do. I'll only make two quick notes. First, these functions rely on the rather dubious Object Pascal operator precedence (Chapter 2) notion that "bitwise and" and the shift operators are "multipliers" and so have a higher precedence than "bitwise or", which is an "adder". Some people would prefer to use parentheses to make the order of execution explicit. Second, the HtmlToTColor function converts a hexadecimal string to an integer by prepending a '$' character—thus making it look like a hexadecimal constant in Object Pascal source—and calling StrToInt. The lower-level Val() procedure also supports this '$' notation.

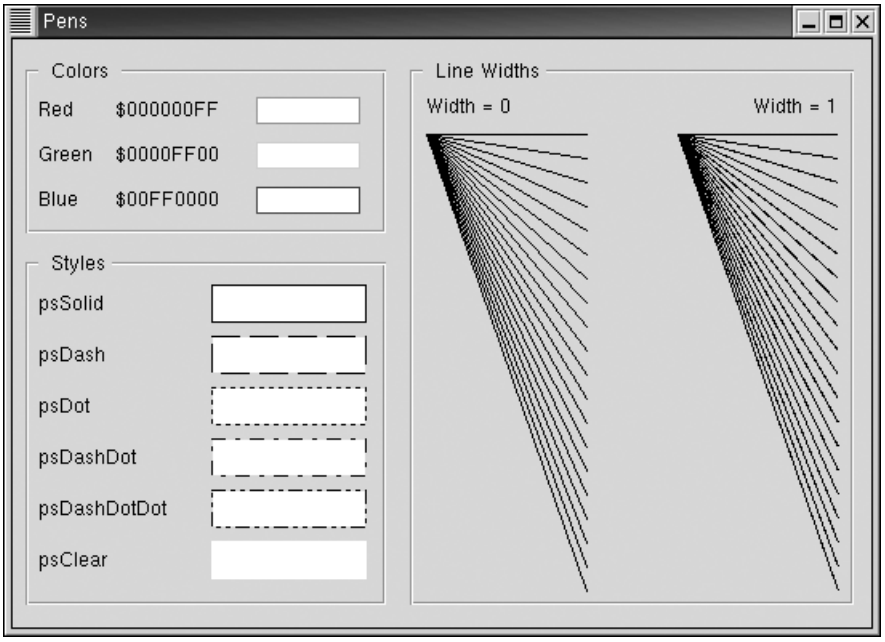


Figure 6-7. The Pens project

You can find the Qt documentation at `www.trolltech.com`.

The Pens project in the ch6/Canvas project group (Figure 6-7) illustrates color encoding, as well as the Width and Style properties. The Width of a pen is measured in pixels, except that a Width of 0 is special. The Qt documentation says a line width of 0 "draws a 1-pixel line very fast, but with lower precision than with a line width of 1. Setting the line width to 1 or more draws lines that are precise, but drawing is slower." This obviously won't affect horizontal or vertical lines, but it does affect diagonal lines (as you can see from the Pens project) and ellipses. I'm not sure that I would characterize the Width = 1 behavior as more "precise" so much as "heavier" —it seems to be drawing pairs of points near places where the Breshenham algorithm's error term is about to roll over—but I guess it doesn't really matter what you call it, so long as you know that the difference exists.

The Style property lets you draw dashed and dotted lines. This is of relatively little utility, except for drawing selection "marquees". Most of the time, you will use either psSolid or psClear. Note that when you draw with a psClear Pen, your drawings have no outline, and the Brush is used to paint the whole region; normally, the Brush is only used for the interior.

The Mode property controls how the Pen color interacts with the current screen color. This lets you "undraw" black lines by redrawing them with a pmXor Mode. Combining other colors is considerably more complicated: If you need to do this, you're on your own.

Brush

Any time you use a graphics primitive that draws a region with an interior, the Brush is involved. A Brush has a Color, which can be any named TColor or RGB integer, just like a Pen's Color. A Brush's Style controls how the Color is used to fill the region. A Brush can also have a bitmap, which overrides the Color; a Brush with a non-Nil Bitmap pays no attention to its Color or Style properties.

Note

Delphi programmers should be sure to note that the CLX's Brush.Bitmap is considerably more useful than the VCL's: Window's brushes only use the top-left 8x8 pixels of their bitmap, while Qt's brushes use the whole bitmap.

Kylix is not Delphi

The Ellipses program in the ch6/Canvas project group is an almost cool demonstration of bitmapped brushes. It loads all the .bmp, .png, or .jpg files in `/usr/share/pixmaps/backgrounds/tiles`—or any directories you list on the command line—and draws random ellipses, each filled with a random bitmap. As you can see if you run it, small, low-color Brush.Bitmap's are faster than big, high-color ones.

The Brushes program in the ch6/Canvas project group (Figure 6-8) is a more comprehensive demo of the interactions of various Brush Style's, Color's, and Bitmap's. There are several things worth noting in its code. In the OnPaint event handler,

```
// Solid Blue
Color := clBlue;
PaintRect(SolidBlue);

// Clear
Style := bsClear;
PaintRect(Clear);

// Bitmap
Bitmap := BrushBitmap.Picture.Bitmap;
PaintRect(Bitmapped);
```

note that a Style of bsClear supercedes the Color. The interior of the Clear rectangle is left untouched, despite the fact that Brush.Color is still clBlue.

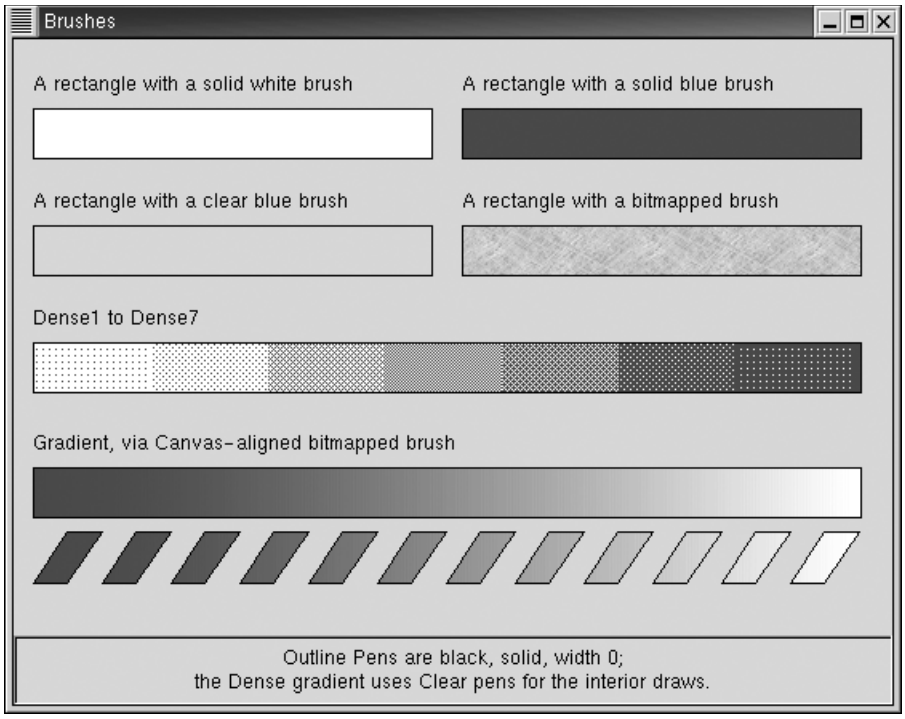


Figure 6-8. The Brushes project

Similarly, the Bitmap supercedes the Color and Style; the interior of the Bitmapped rectangle is filled with the 'paper' bitmap, despite the fact that Brush.Color is still clBlue and the Brush.Style is still bsClear.

In the DenseX code, note how the TBrushStyle's that are neither bsSolid nor bsClear blend the two behaviors. The patterned brushes draw some of the pixels in the interior in the current Brush.Color, while leaving others untouched. Thus, I draw the DenseSpectrum rectangle as a (coarse!) gradient from white to blue by drawing the interior of the rectangle in solid blue, then overlaying it with progressively less dense white.

```
// DenseSpectrum
Bitmap := Nil;
Style  := bsSolid;
Color  := clBlue;
Top    := DenseSpectrum.Top + DenseSpectrum.Height + 8;
Bottom := Top + RectHeight;
```

```
Width  := (Right - Left - 2) div 7;
Rectangle(Left, Top, Right, Bottom);

Pen.Style := psClear;
Color     := clWhite;
for Index := 0 to 6 do
begin
  Style := TBrushStyle(ord(bsDense1) + Index);
  Rectangle( Left + Width * Index + 1, Top + 1,
             Left + Width * (Index + 1) + 1, Bottom – 1 );
end;
```

Note also that the psClear Pen.Style for the interior rectangles draws no outline, and fills the whole rectangle with the brush. This is why the second Rectangle() call in the above snippet—the one that draws the white DenseX interiors—uses Top + and Bottom – 1. If it didn't, it would overwrite the rectangle's black frame.

Finally, perhaps the coolest part of this little demo program is the "Canvas-aligned bitmapped brush" at the bottom, which uses a bitmap I build in the form's OnCreate handler. Because the Brush's Bitmap is Canvas-aligned—*ie*, implicitly tiled over the whole Canvas, with its top-left corner at the top-left of the Canvas—I didn't have to reset the Brush between polygons to draw the slashed 'windows' 'onto' the bitmap: I just drew a dozen polygons, all using the same Brush.

```
for Index := 0 to 11 do
  Polygon( [ Point(Left + Width * Index + Width div 2, Top),
             Point(Left + Width * (Index + 1),         Top),
             Point(Left + Width * Index + Width div 2, Bottom),
             Point(Left + Width * Index,               Bottom) ]);
```

Of course, this Canvas-alignment is a bit of a double-edged sword. You might well **want** the top-left edge of the bitmap to be in the top-left corner of the area you're filling. For example, the Brushes project had a bug at first: the gradient color bar was drawing a solid blue bar at the far right. I was building a bitmap exactly as wide as the rectangle, and since it was tiled from the top-left of the Canvas, I was losing the leftmost pixels of the gradient, and getting them on the right.

For this simplistic demo, I just made the bitmap a bit wider, and left the leftmost pixels undefined. More generally, you might want to use a PaintBox to get a new Canvas, whose top-left is just where you want the top-left of the bitmap to be.

Alternatively, you can use the Qt function QPainter::setBrushOrigin to control where the top-left corner of the bitmap is placed. For example, the Brushes project actually builds two bitmaps, and calls

```
QPainter_setBrushOrigin(Canvas.Handle, Left, 0);
```

so that it doesn't have to have Left unused pixels in the second bitmap.

Font

When you draw text on a Canvas, you use the Canvas's current Font. A Font has four key properties—Name, Color, Size, and Style—and four minor properties—CharSet, Pitch, Height, and Weight. Unfortunately, the Font object doesn't include any methods or properties that can tell you if a given font is a nice, high-quality scalable font or some crufty old bitmapped X font that hasn't looked good since someone built it by hand at MIT in 1985—see the "TFont *vs* QFont" sidebar for more information on how to do this with Qt calls.

TFont *vs* QFont

Linux programmers are used to it, but Windows programmers are in for a real shock: Linux font handling is pretty poor. Under Windows, you can easily[5] tell if a font will scale smoothly. If it's a True Type font, it scales smoothly. If it's a bitmapped font, it scales poorly. You can easily tell if a font is a True Type font; you can choose to only 'see' True Type fonts. Not under Linux!

The Fonts project in the ch6/Canvas group illustrates some of the difficulties, as well as some lib/QGrabBag code that you can use to access the Qt QFontInfo and QFontDatabase classes.

When you set a TFont's properties, Qt tries to find the exact match for the Name, CharSet, Pitch, Size and so on. If it can't find an exact match, it will give you the best match it can come up with. This might not be a very good match! (This is true under Windows, too, but not only does Windows generally do a better matching job than X, most developers limit their exposure to this issue by specifying only the standard fonts that are found in all Windows versions. There are **no** standard fonts found in all Linux versions.)

However, the quality—or lack thereof—of the best-match font won't be reflected in the TFont. (This is true of Delphi, too.) TFont passes property

5. Well, actually, users can install font handlers for Postscript &c, and this does complicate matters. But you can still write code that only sees TrueType fonts.

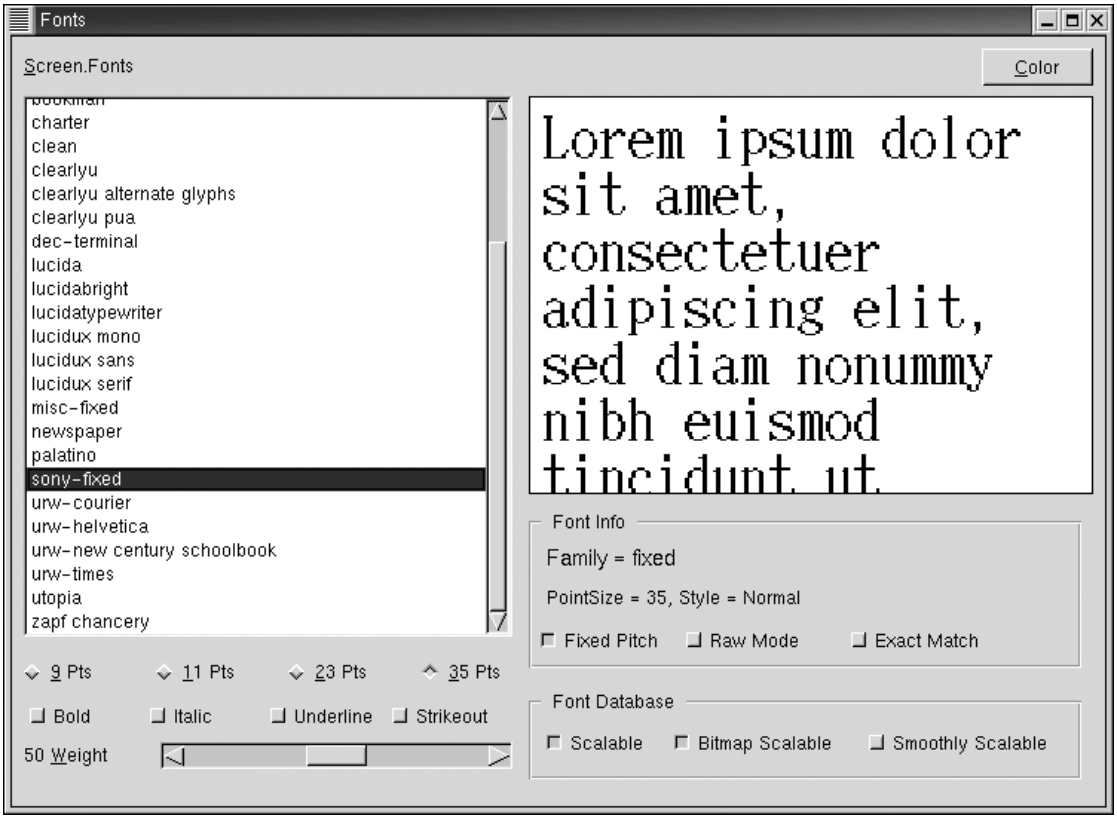


Figure 6-9. A poorly scaled bitmap font

read requests on to Qt, which simply returns the last value you set. To get the actual values, you need to use QFontInfo. The Fonts project does this: It allows you to specify only odd sizes, like 9, 11, 23, and 35 points to increase the chances that Qt won't be able to find an exact size match.

The QFontInfo class will not tell you how a given font family scales. You need to use QFontDatabase, which offers three methods: isScalable, isBitmapScalable, and isSmoothlyScalable. To complicate things, these functions do not apply to a whole font family, but to a font family, with a given combination of style (bold, italic) attributes, and a particular character set. The following abbreviated snippet from the Fonts project shows how to call these functions:

```
procedure TFontsFrm.FontChange(Sender: TObject);
var
  Info:  IFontInfo;
  Family, Encoding, Style:    WideString;
begin
  Info := FontInfo(LoremIpsum.Font);

  Family := LoremIpsum.Font.Name;
  Style  := FontDatabase.StyleString(LoremIpsum.Font);
  QFont_encodingName(@ Encoding, Info.CharSet);

  Scalable.Checked :=
    FontDatabase.IsScalable(Family, Style, Encoding);
  BitmapScalable.Checked :=
    FontDatabase.IsBitmapScalable(Family, Style, Encoding);
  SmoothlyScalable.Checked :=
    FontDatabase.IsSmoothlyScalable(Family, Style, Encoding);
end; // TFontsFrm.FontChange
```

The Family parameter should be the same as TFont.Name, a name that might appear in Screen.Fonts. Screen.Fonts is generated by QFontDatabase::families, which returns a name as "foundry-family" when a family exists in several foundries. (Hence names like 'adobe-courier' and 'urw-courier'.) QFontInfo::Name strips off the foundry-part: If you use a QFontInfo Name for a Family parameter, you will get the info for **a** family with that name, but not necessarily the one you want.

The Style parameter can be 'Normal', 'Bold', 'Italic', or 'Bold Italic'. Rather than build these strings myself—and risk possible future incompatibilities or locale issues—I prefer to use the QFontDatabase::styleString function to describe an existing TFont.

Similarly, I don't think it wise to hardcode locale names into my apps. QFont::encodingName returns a Qt-legible string version of the Qt locale enum I get from the QFontInfo::charSet.

Once I gather this information and call isScalable, isBitmapScalable, and isSmoothlyScalable, things are … still complicated. Some fonts (like 'avantgarde', on my system) that **do** scale smoothly are reported as not scaling smoothly by any of the three functions; others ('zapf chancery') are only reported as scaling smoothing with an 'Italic' style; while others ('adobe-courier') that manifestly **don't** scale smoothly are reported as doing so!

I urge you to run the Fonts project and draw your own conclusions, but I'd suggest this rule of thumb: Avoid fonts that are not smoothly scalable, and fonts that are bitmap scalable.

Key properties

TFont.Name selects a font or, more properly, a font family. Most font families include fonts for different sizes, styles, and character sets. If the font Name you select is on the user's system—*ie*, is in Screen.Fonts—that's the font you'll get. If the font doesn't exist, Qt will do its best to find a plausible font. See the Qt documentation (`www.trolltech.com`) for details.

TFont.Color can be any named TColor, or RGB value.

TFont.Size is a positive integer that is supposed to represent the font's height in "points", which is a traditional printer's measure: There are 72 points to an inch. Kylix passes Size requests straight through to Qt, which presumably[6] uses the same display size information it exposes in QPaintDeviceMetrics to convert heights in points to heights in pixel.

Note

When you read Font.Size, Kylix passes the request on to Qt, which returns the last Font.Size that you set. When you change the font family, if the font is not "smoothly scalable" Qt may not give you the exact font size that you asked for; it may choose an actual size that's as much as 20% bigger or smaller than what you asked for. Reading the Font.Size property, however, will give you what you ***asked for****, not what you actually* ***got****. To find out what Qt gave you, you need to use QFontInfo.*

TFont.Style property is a `set of TFontStyle`, which means it can be any combination of fsBold, fsItalic, fsUnderline, and fsStrikeOut. Note that because the set is a property, not a public field, you can't use the system procedures Exclude and Include with Font.Style. You have to write code like `with ThisFont do Style := Style + [fsBold]` or `with ThisFont do Style := Style - [fsItalic]`.

Minor properties

Naturally enough, TFont.CharSet selects the character set. Some font families will be designed to have a similar feel across character sets, so you might have a font named "Globalization" that has faces for Latin-1 as well as Greek,

6. Yes, I am in a state of sin: I'm writing a book about programming on an Open Source operating system, using an Open Source widget library—and I'm making inferences about how Qt does things, based on the public API, instead of reading the Source.

Russian, Japanese, Chinese, Korean, Thai, and so on. The default value, fcsDefaultCharSet, means that Kylix will ask Qt what CharSet is appropriate for the user's locale. This is usually fine, but may well cause problems in Latin-1 applications running on machines that default to an Asian language.

Note

English-only Delphi apps running on NT4 machines with a DBCS [Double Byte Character Set] locale have problems if they use DEFAULT_CHARSET, Delphi's equivalent of fcsDefaultCharSet. Their English text gets transcribed to the machine's default character set. To make sure that the text displays correctly even on machines that aren't set to an ANSI locale, you have to explicitly set the Font's CharSet to ANSI_CHARSET. I've done simple tests that don't show a similar problem–but I wasn't using any text that contained UTF-8 escape sequences (see Chapter 1).

TFont.Pitch allows you to specify whether the font is proportional (an 'i' is narrower than an 'M') or fixed pitch (all characters are the same width). This is significantly less useful than it might seem. Setting Pitch to fpFixed will **not** make a proportional font use fixed-width character cells as if it were a fixed-pitch font, nor will setting Pitch to fpVariable make a fixed-pitch font into a proportional one. Qt's font matching algorithm gives more weight to the Name and CharSet than to the Pitch, so changing the Pitch of a selected font will have no effect. The only time Pitch will make a difference is when the font family Name doesn't exist on the current system, and Qt has to select the best match based on CharSet, Pitch, Size, Weight, and whether or not fsItalic is in the Style set.

Kylix is not Delphi

TFont.Height is just like Size, but is measured in pixels, not points. Delphi programmers should note that Font.Height is always positive under Kylix, which does not use negative Height's to indicate the size *sans* internal leading.

TFont.Weight is a sort of generalized Bold attribute. Bold actually translates to a standard Weight, fwBold, while non-Bold is fwNormal. You can, at least in principle, get light or extra-bold fonts by setting Font.Weight—but few Linux fonts look any different with a Weight of 1 than with a Weight of fwNormal, or any different with a Weight of 100 than a Weight of fwBold. Note that adding and removing fsBold to and from the Style set **has no effect** if Weight is neither fwNormal nor fwBold. If the Weight may be non-standard, be sure to set Weight to fwNormal or fwBold before toggling the fsBold Style attribute.

Drawing operations

The drawing operations use the drawing tools to draw on a Canvas. Drawing operations can be roughly divided into geometric operations that use the Pen and Brush, text operations that use the current Font, and bitblts that transfer rectangular pieces of an image from one Canvas to another.

Geometric operations

The geometric operations are all really pretty straightforward. I've used some of them in example code already and, of course, they're well documented in the online help. In fact, I'll just reiterate the basic point that the geometric operations outline with the Pen and fill with the Brush, and talk a bit about the operations that take open array parameters.

Graphics primitives like PolyLine and Polygon take an `array of TPoint` parameter. Perhaps the most straightforward way to supply this array is the way I did it in the Brushes project: an open array of calls to the Point function.[7] As *per* Section 1, Kylix allocates space for the array on the stack, and passes offsets into it to each call to the Point function, which fills in each Point Result record. Kylix passes a pointer to the first Point, and an invisible Length parameter, to the PolyX function, and then deallocates the array when the function returns. All this is done automatically; it's incredibly convenient and reasonably efficient.

However, open arrays like this are also assignment compatible with both dynamic arrays and regular, static arrays. This means that you can pass static arrays and dynamic arrays to the PolyX routines that take open array parameters. You usually draw figures (in every OnPaint event) many more times than you change them (when you create, destroy, or move an element). Therefore, if a given figure is relatively expensive to compute (perhaps involving perspective transforms, or z-order decisions), you may want to store the vertices in a static or dynamic array. Then, your OnPaint handler could pass the drawing primitive the pre-calculated array of vertices, instead of recalculating them every time.

For example, in this snippet

7. I also use this approach in the FL3 project in the ch6/Canvas group. I won't talk much about this project—a Kylix port of a Delphi 1 port of a Turbo Pascal 4 project—but I will mention it again in the section on Printing. For a detailed walkthrough of FL3, please see my magazine article about it at http://www.midnightbeach.com/jon/pubs/3D_Fractal_Landscapes.html.

```
const
  Triangle: array[1..3] of TPoint = (
    (X: 10; Y: 20), (X: 100; Y: 100), (X: 30; Y: 200)
  );
  Arbitrary: array of TPoint = Nil;

procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Polygon(Triangle);
  if Assigned(Arbitrary) then
    Canvas.Polygon(Arbitrary);
end;
```

Canvas.Polygon() works just as well with static arrays like Triangle and dynamic arrays like Arbitrary as with an inline array: `Canvas.Polygon(Triangle)` is functionally equivalent to `Canvas.Polygon([Point(10, 20), Point(100, 100), Point(30, 200)])`, but clearer and a bit faster.

Tip

The Slice() function allows you to select a few elements from a larger array.

Text operations

Two basic routines draw text on a Canvas in the current Font: TextOut and TextRect. These routines differ in two ways: TextOut simply takes a position and a string, and draws the string at the position, without clipping or wrapping, and without much alignment control. You can use the TextAlign property to control whether the Y parameter is the top-line or the bottom-line, but that's about it. TextRect, on the other hand, allows you to specify a clipping rectangle and to use the full complement of Qt alignment flags, including word wrap and vertical and horizontal alignment.

Delphi programmers should note that where Kylix's TextOut is basically identical to Delphi's, Kylix's TextRect adds an optional TextFlags parameter that allows you to use any combination of the Qt TextAlign flags in Table 6-4.

Kylix is not Delphi

Table 6-4. Qt TextAlign flags

Qt.pas name	Interpretation	Notes
AlignmentFlags_AlignLeft	taLeftJustify	Combining with AlignRight or AlignCenter will give unpredictable results.
AlignmentFlags_AlignHCenter	taCenter	
AlignmentFlags_AlignRight	taRightJustify	
AlignmentFlags_AlignTop	tlTop	Combining with AlignVCenter or AlignBottom or will give unpredictable results.
AlignmentFlags_AlignBottom	tlCenter	
AlignmentFlags_AlignVCenter	tlBottom	
AlignmentFlags_AlignCenter	ord(AlignmentFlags_AlignLeft) or ord(AlignHCenter)	You can only have one horizontal and vertical flag; AlignCenter counts as one of each
AlignmentFlags_SingleLine	Treat all white-space as space; don't break or wordwrap.	
AlignmentFlags_DontClip	Ignore the bounds rectangle if necessary.	Can draw where you don't want it to.
AlignmentFlags_ExpandTabs	Honor ^I characters.	
AlignmentFlags_ShowPrefix	& underlines next character; && is an un-underlined &.	
AlignmentFlags_WordBreak	Wrap long lines at word boundaries.	Can be combined with ExpandTabs.
AlignmentFlags_DontPrint	Don't draw on the Canvas.	Not particularly useful in Kylix, as TextRect doesn't return the modified bounds rect. Use TextExtent instead.

Note

As the table indicates, these AlignmentFlags are defined in Qt.pas. GUI programs do not automatically use Qt the way they automatically use QGraphics, QControls, QForms, and QDialogs; if you want to pass AlignmentFlags to TextRect (or TextExtent), you'll need to manually add Qt to one of the unit's uses clauses.

Unfortunately, the combination of a verbose mechanical translation that turned Foo::Bar into Foo_Bar and the decision to treat C++ assigned enums as non-numeric ordinals condemns us to some long-winded and unreadable code when it comes to Qt constants like these. Where a C++ programmer would combine these flags as *eg* `AlignCenter | WordBreak | ExpandTab`, we're forced to use `ord(AlignmentFlags_AlignCenter) or ord(AlignmentFlags_WordBreak) or ord(AlignmentFlags_ExpandTab)`. Ugh.

Finally, it's not uncommon to need to know how many pixels a given piece of text will take to display. Perhaps we need to display some text using multiple fonts, or we want to trim the text to the space available, or to adjust various controls to make room for the text. TextWidth returns the pixel width of a given string, without any word wrapping, & underlining, or tab expansion. Similarly, TextHeight returns the pixel height of a given string, also without any word wrapping, & underlining, or tab expansion.

If you want to specify these alignment flags, or you want both height and width, you should use TextExtent. This is available in two overloaded versions: One is a function that returns a TSize—which is just like a TPoint except that the fields are CX and CY, instead of X and Y—while the other is a procedure that modifies a TRect parameter. Use whichever is more convenient. The function calls the procedure, but the speed difference is not significant and, of course, it's always possible that this may be different in future implementations.

Tip

*Both TextWidth and TextHeight call TextExtent internally. If you are using both values, you should call TextExtent directly. TextExtent converts your string to a WideString (if it isn't one already) and then calls Qt. This is relatively expensive: It's more worth paying attention to not doing this twice than worrying about one extra layer of Pascal function calls within CLX (*viz *TextExtent* vs *TextExtent).*

Kylix is not Delphi

As with TextRect, Delphi programmers should note that where Kylix's TextHeight and TextWidth are basically identical to Delphi's, Kylix's TextExtent adds an optional TextFlags parameter. The Text project of the ch6/Canvas project group illustrates this difference. TextHeight and TextWidth report the unwrapped size: The pixel width of the longest paragraph, and the height of five lines of text. By contrast, I use the form of TextExtent that allows me to pass in the same flags that I pass to TextRect, and I get the wrapped size: The pixel width of the longest wrapped line, and the height of all the wrapped lines.

Bitblts

The bitblt—or Bit Block Transfer—is the basic workhorse of GUI windowing systems. It moves a rectangular block of pixels (bits) from one portion of the screen to another, or from one off-screen bitmap to another, or between the screen and an off-screen bitmap. It's used for everything from drawing text to drawing pictures to moving windows. Paradoxically, because of its very ubiquity, Kylix programs rarely need to do many bitblt's themselves; it's built into some of the higher-level primitives and into key components like TImage and TImageList. Of course, "rarely" is not the same as "never", and so Kylix canvases include bitblt support, *via* the CopyRect procedure. (To copy a whole image—perhaps loaded from a file—onto a Canvas, you'd typically use the Draw (or StretchDraw) procedure. See the *TBitmap* section, below.)

CopyRect takes three parameters: a destination rectangle, a source Canvas, and a source rectangle. The Canvas that 'does' the CopyRect is the Canvas that's copied to. *Eg,*

```
ThisCanvas.CopyRect(ThisRect, ThatCanvas, ThatRect);
```

copies ThatRect from ThatCanvas to ThisRect on ThisCanvas. The use of a destination rectangle is actually a bit misleading: Only the Left and Top fields are used. That is, bitblts are neither clipped nor stretched when the destination rectangle is not the same size as the source rectangle. Under Kylix 1, the transfer will always be the size of the source rectangle; this may change in future releases, if Qt implements a StretchBlt function.[8]

8. If you really need StretchBlt abilities now, you can always synthesize them, *via* the following four step process: 1) Create a temporary TBitmap the size of the source rectangle. 2) Copy the source rectangle to the temporary bitmap. 3) StretchDraw the temporary bitmap to the destination rectangle. 4) Free the temporary bitmap.

You can use CopyRect to move an image (or part of an image) from any Canvas to any other. CopyRect can also copy from one part of a Canvas to another, as in

```
ThisCanvas.CopyRect(ThisRect, ThisCanvas, ThatRect);
```

When you do a 'self copy' like this, there is no requirement that ThisRect and ThatRect not intersect. That is, you can do bitblt's that overwrite (part of) the source. See the BitBlt project of the ch6/Canvas group for an example.

Normally, CopyRect simply replaces every pixel of the destination with the appropriate pixel of the source. In some cases—combining layers, and so on—this isn't what you want. The Canvas.CopyMode property allows you to specify one of sixteen different ways to combine the source and destination bits.

Kylix is not Delphi

Note

Delphi programmers should note that Kylix's CopyMode is different from Delphi's. In Delphi, CopyMode is a 32-bit integer that allows you to specify any possible Win32 "ternary raster ops". The Borland defined copy modes like cmSrcCopy are simply named constants. In Kylix, CopyMode is an enum; if you have existing code that uses custom copy modes, you'll have to rewrite it.

One common use for copy modes is transparent bitmaps. You use various copy modes to define a mask bitmap, a monochrome bitmap with 1's where the transparent bitmap has non-background pixels and 0's where the transparent bitmap has background pixels (or *vice versa*). You then use this mask to "cut out" this destination rectangle—*ie*, force every pixel that's going to be replaced to be black—then "drop in" a masked source rectangle. You can do this in Kylix, of course—but you shouldn't, as the TBitmap class has already done it for you. (See the *TBitmap* section, below.)

ClipRect

Normally, of course, you want to be able to draw on the whole Canvas. Sometimes, though, you don't. For example, one of my recent contracts needed to draw a "letterhead" at the top of a scrolling sheet of "paper". When the paper was scrolled down only a little, some of the letterhead showed and some didn't. Knowing how far the paper had scrolled, I could just draw the letterhead

at a negative Y position, and some would appear and some wouldn't. But, the customer also wanted a decorated margin all around the paper. Undoubtedly, I could have layered one TPaintBox on top of another, but the simplest solution was to just set Canvas.ClipRect to not overwrite the decorated margin. Then, I could just write the letterhead at the top of the "paper", and the system would take care of only drawing the parts that were actually visible.

Qt actually supports clip *regions*, which are pretty comparable to Windows regions. You can create simple regions with operations analogous to drawing rectangles, ellipses, and polygons, or you can convert a transparent bitmap into a region. You can combine simple regions into compound regions with union, intersection, subtraction, and xor operators. As under Windows, you can use these regions to clip, or for hit detection *via* the QRegion::contains() function. Also as under Windows, Kylix allows you to call these functions (Qt.pas contains QRegion_ bindings for all QRegion:: member functions) but does not provide Object Pascal wrappers.

TBitmap

Naturally enough, the TBitmap class represents bitmapped images. Several components—in particular TImage and the various classes like TSpeedButton and TBitBtn with a Glyph property—have TBitmap properties, and you can of course create your own instances of TBitmap at runtime. TBitmap includes methods to load from and save to files and streams, and it has a Canvas so you can create complex drawings off-screen. You can Draw or StretchDraw a whole bitmap onto any Canvas, and you can use CopyRect to extract pieces of a bitmap.

When you Assign() one bitmap to another, all that's copied is some descriptive information. The actual image information—which can be quite large—is reference counted by Qt, in much the same way as strings are by Kylix, so that copying takes very little time or memory.

Note

See Chapter 7 for a discussion of loading bitmaps from your executables' resource section.

File formats

To load a bitmap from a file image, you call its LoadFrom or LoadFromFile methods. To save a bitmap to file, you call its SaveTo or SaveToFile methods. (You can also load from and save to a *stream*; see Chapter 7 for more on Kylix's streams.) You can control the output format by setting the Format property before calling SaveTo.

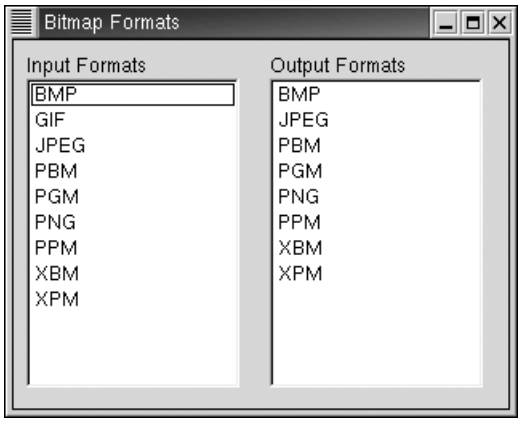


Figure 6-10. File formats

The Bitmap project in the ch6/Canvas project group illustrates how simple this can be.

```
if OpenDialog.Execute then
begin
  Image.Picture.Bitmap.LoadFromFile(OpenDialog.FileName);
  ClientHeight := Image.Top + Image.Height;
  ClientWidth  := Image.Width;
end;
```

OpenDialog.Execute pops up the open file dialog, and returns True if the user clicked "Open". The FileName property contains the complete path to the selected file. The TImage can hold a variety of different Picture types, so I use Image.Picture.Bitmap to specify that it's a bitmap file that I want to load. Kylix and Qt do all the work of checking and decoding the file format.

The Bitmap project also shows how to get the current list of supported formats. (See Figure 6-10.) This may change from release to release, as new

formats come along or various legal issues ruin old standards. Under Kylix 1, for example, you can read GIF files but you can't write them.

```
procedure TBitmapFormatsFrm.FormCreate(Sender: TObject);
var
  List:   QStringListH;
begin
  List := QStringList_create;
  try
    QImage_inputFormatList(List);
    CopyQStringListToTStrings(List, InputFormats.Items);

    QImage_outputFormatList(List);
    CopyQStringListToTStrings(List, OutputFormats.Items);
  finally
    QStringList_destroy(List);
  end;
end; // TBitmapFormatsFrm.FormCreate
```

LoadFromFile and SaveToFile figures out what file format to use from the file's extension. If you are loading from or saving to an anonymous stream, you need to explicitly set the Format property to one of the values in the input or output formats list.

While Qt offers control over image quality (*eg*, JPEG compression), Kylix doesn't wrap this. To control the space/quality tradeoff, you need to make a raw Qt call, as in this function from lib/QGrabBag:

```
type
  TQtQuality = -1..100;


function SaveBitmap(       Bitmap:   TBitmap;
                     const Filename: WideString;
                     const Format:   string;
                           Quality: TQtQuality = -1): boolean;
begin
  Result := QPixmap_save(Bitmap.Handle, @ Filename, PChar(Format), Quality);
  // Quality of -1 is default quality, the same as you get from Kylix.
  // 0 is low quality (small file); 100 is high quality (big file).
end; // SaveBitmap
```

Draw and StretchDraw

Draw and StretchDraw are Canvas methods that draw any TGraphic on a Canvas. TBitmap is the most important type of TGraphic; others include TIcon and TDrawing. With Draw, you specify an X, Y position and a graphic; the graphic is drawn full sized, with the top left at [X, Y]. With StretchDraw, you specify a rectangle and a graphic; the graphic is stretched or shrunk to fit into the rectangle. Note that since a TBitmap has a Canvas property of its own, you can Draw *This* bitmap onto *That* bitmap: `That.Canvas.Draw(X, Y, This)`.

Transparency

All TGraphics have a boolean Transparent property, and Draw and StretchDraw both honor transparency. When you load a bitmap from a file with transparency information, the bitmap will, naturally, be transparent. If you want to make a normal bitmap transparent, you have to specify the transparent color.

Bitmaps have three transparency related properties: Transparent, TransparentColor, and TransparentMode. TransparentMode has two different values: tmAuto and tmFixed. tmAuto is the default, and means that the color of the bottom-left pixel will be used as the transparent color. That is, if you set Transparent to True, the TransparentColor property will be set to the color of the bottom-left pixel, and all pixels of that color will be Transparent. If you set TransparentMode to tmFixed, you can set the TransparentColor to whatever you like. (Conversely, if you do set TransparentColor, TransparentMode will automatically change to tmFixed.) Specifying a TransparentColor allows you to have bitmaps where the bottom-left pixel is not transparent.

Note

Delphi programmers need to be aware that the CLX TBitmap is rather different from the VCL TBitmap. In particular, the CLX TBitmap has no MaskHandle property.

Kylix is not Delphi

Low-level manipulation

Most programs draw on bitmaps just as they draw on any other canvas, with TextOut, Rectangle, Polygon, and the like. But some of the time, you need low-level pixel-by-pixel access. For example, it would have been impossible to create the gradient bitmap in the Brushes example without setting pixels

one by one. Similarly, applying any sort of graphics filter to a bitmap requires reading and writing individual pixels. Since there can be hundreds of thousands of pixels in even a relatively small image, it's important that pixel-by-pixel access is very fast.

TBitmap provides both a PixelFormat property that allows you to get and set the bit-depth of each individual pixel, and a ScanLine property which gives you access to each row of the image as an array of pixels. The PixelFormat is of type

```
TPixelFormat = (pf1bit, pf8bit, pf16bit, pf32bit, pfCustom);
```

but probably most programs that manipulate pixels force it to pf32bit, as an array of 32-bit integers is a lot easier to work with than a "high color" array of 16-bit integers, a palletized array of 8-bit bytes, or (especially!) a monochrome bit-stream. (You're not likely to see pfCustom except with uninitialized bitmaps; pfCustom basically means 'unexpected result from QImage::depth()'.)

The ScanLine property is an array property that returns an untyped `pointer` to the first pixel in the row. It's your responsibility to cast it to something like a `^ array[word] of LongWord` as in this relatively long bit from the Brushes project:

```
type
  TInterpolateRec = record
                    Start, Delta, Steps: integer;
                    end;

function InterpolateRec(Start, Stop, Steps: integer): TInterpolateRec;
begin
  Result.Start := Start;
  Result.Delta := Stop - Start;
  Result.Steps := Steps;
end; // InterpolateRec

function Interpolate( const InterpolateRec: TInterpolateRec;
                           Step:            integer ): integer;
begin
  with InterpolateRec do
    Result := Start + Delta * Step div Steps;
end; // Interpolate

const
  StartColor = clBlue;
  StopColor  = clWhite;
```

```
procedure TBrushesFrm.FormCreate(Sender: TObject);
type
  ArrayOf32Bit = array[word] of LongWord;
var
  cbLeft, cbRight: integer;
  Colors, Index:   integer;
  R, G, B:         TInterpolateRec;
  ScanLine:        ^ ArrayOf32Bit;
begin
  ColorBar := TBitmap.Create;

  cbLeft   := 0;
  cbRight  := Bitmapped.Left + 250 - 1;

  Colors := cbRight - DenseSpectrum.Left;
  R := InterpolateRec( StartColor and $FF,
                       StopColor  and $FF,);
  G := InterpolateRec( StartColor and $FF00 shr 8,
                       StopColor  and $FF00 shr 8, Colors);
  B := InterpolateRec( StartColor and $FF0000 shr 16,
                       StopColor  and $FF0000 shr 16, Colors);

  ColorBar.Height      := 1;
  ColorBar.Width       := cbRight - cbLeft;
  ColorBar.PixelFormat := pf32bit;
  ScanLine := ColorBar.ScanLine[0];
  for Index := 0 to Colors do
    ScanLine^[Index + DenseSpectrum.Left + 1] :=
      pf32RGB( Interpolate(R, Index),
               Interpolate(G, Index),
               Interpolate(B, Index) );
end; // TBrushesFrm.FormCreate
```

The first line of FormCreate creates the bitmap and saves a reference to it in a private field of the form object. (The OnDestroy event handler frees it.) I calculate the bitmap Width from the position of various labels … after all, this is just a quick demo. Real code would probably bury those calculations in a single, maintainable method. The bitmap is only 1 high, and I force the PixelFormat to pf32bit. The Interpolate() and InterpolateRec() functions use a straightforward mul/div idiom to calculate each pixel's RGB components.

*ScanLine is **not** an array of TColor values, even in pf32bit mode.*

The real work is done in the for loop, which sets each pixel in turn. Note that it uses pf32RGB, not the TColor returning RGB function I showed earlier. pf32bit pixels use a different encoding than TColor does; their R and B com-

ponents are switched. Thus, the low (or first) byte is the Blue component, the second byte is the Green component, and the third byte is the Red component—or xxRRGGBB in 'pseudo-hex'.[9]

Pixmaps

Kylix is not Delphi

Delphi programmers are familiar with the VCL's TBitmap.HandleFormat, which allows you to switch between fast, device-dependent bitmaps [DDB], and editable device-independent bitmaps [DIB]. The CLX's TBitmap doesn't work that way. Each bitmap exists both as an Image, which corresponds more or less to a DIB, and a Pixmap, which corresponds more or less to a DDB. The Image is what gets loaded and saved, and the Pixmap is what gets drawn to the screen. Kylix keeps the two in synch, and you rarely have to pay attention to the implementation details.

However, you do need to keep the implementation in mind when you are working with the ScanLine property. You can use a ScanLine pointer to read and write the Image side of the bitmap. Before a changed image can be displayed, any changes have to be reflected on the Pixmap side. Unfortunately, there's no way to tell Qt 'I just changed this rectangular region of this Image; please migrate those changes to this Pixmap'—you have to regenerate the whole Pixmap.

The ScanLine property's read function—QGraphics.TBitmap.GetScanLine—handles this by calling FreePixmap before returning a pointer to the first pixel. This means that the next time you Draw the bitmap or do anything else that requires a Pixmap, Kylix will automatically regenerate one. For simple applications like the Brushes project, this poses no real concerns. I built a bitmap pixel by pixel, and it worked just fine when I assigned it to a Brush.

However, if you are doing some long operation on a large bitmap and wish to show your progress, line by line, you have to pay attention to this issue. Each time you get the ScanLine of a new row, Kylix calls FreePixmap. If you haven't done anything that required a QPixmap since the last read of the ScanLine property, this has no real effect. However, if you've gone and done a CopyRect of the updated line to the screen, you've created a new Pixmap. This took a while, which means your line-by-line processing is limited by the need to keep recreating (and freeing) Pixmaps of the whole image.

---

9. The high, or 'xx' byte represents an optional "alpha channel". If you've enabled the alpha channel, this allows you to specify the transparency of each pixel, which obviously can be very useful for layering images on top of each other, as programs like PhotoShop do, or for fading from one slide to the next. If you want to experiment with alpha channel effects, see the Qt documentation (`www.trolltech.com`) for QImage::setAlphaBuffer.

The best solution is to maintain a one-row working bitmap, and manipulate its ScanLine[0], then Draw or CopyRect it to the large bitmap and to the screen. Remember, you'll need to FreePixmap at the start of each row, either by reading ScanLine[0] again or by explicitly calling FreePixmap.

Printing

Printing under Kylix is reasonably straightforward. Use the `QPrinters` unit to get access to the Printer function. This returns a reference to a global TPrinter object that lets you select printers from the list of printers attached to the system. It also provides information about the printers, and provides a Canvas you draw on just as you'd draw on any other Canvas. The FL3 project in the ch6/Canvas group provides a simple example.

Printer.Printers is a TStringList of available printers that you can use to let users select a printer. Set Printer.PrinterIndex to select a printer from this list; by default, you print to the system default printer, or Printer.PrinterIndex = -1.

You have to call Printer.BeginDoc before you start using Printer.Canvas, and you have to call Printer.EndDoc before the page will print. Printer.PageHeight and Printer.PageWidth allow you to scale your drawings to fit the page.

Kylix is not Delphi

Delphi programmers will note the absence of TPrintDialog and TPrinterSetupDialog from the Dialogs tab of Kylix's Component Palette. Qt does provide a dialog that pretty much combines the two, but it's a tad … quirky. (Presumably that's why Borland didn't wrap it for Kylix 1.) To bring up the Qt print dialog, use

```
if (Printer.PrintAdapter as TQPrintAdapter).ExecuteSetup then {print};
```

QPainter

If you look at QGraphics.pas, you can see that TCanvas is actually a fairly thin wrapper for Qt's QPainter. Most Canvas operations are implemented as QPainter operations, using only a few lines of code. This means that dealing with Qt via the Canvas layer isn't all that much slower than dealing with Qt directly. It also means that you can call any QPainter functions that Borland doesn't wrap.

For example, QPainter includes a rich set of coordinate transformations that make it easy to rotate, scale, and translate drawings. If you want to draw rotated text, just about all you need to do is to call QPainter_rotate, as in this OnPaint event handler from the QPainter project (Figure 6-10) in the ch6/Canvas project group.

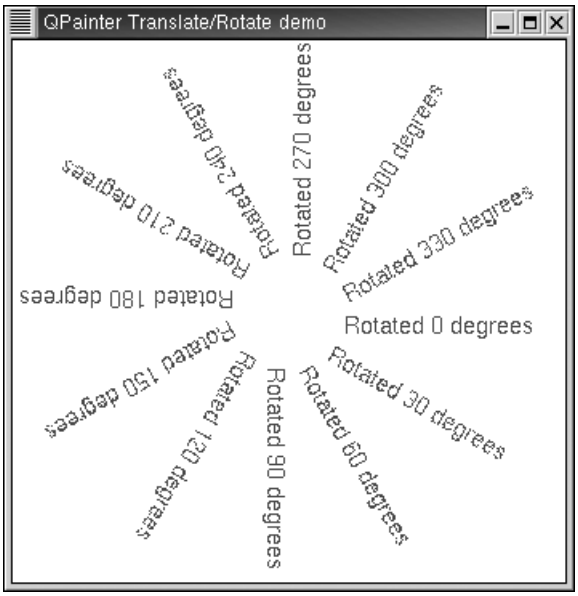


Figure 6-11. Rotated text

```
procedure TQPainterFrm.PaintRotatedText(Sender: TObject);
const
  Steps = 12;
var
  Index: integer;
begin
  Canvas.FillRect(ClientRect);
  Canvas.Font.Color := clBlue;

  // Move 0,0 to the center of the form
  QPainter_translate(Canvas.Handle, Width div 2, Height div 2);
  for Index := 0 to Steps - 1 do
  begin
    Canvas.TextOut( 35, 0,
                    Format('Rotated %0.f degrees', [Index * 360 / Steps]));
    // Rotate clockwise
    QPainter_rotate(Canvas.Handle, 360 / Steps);
  end;
end; // TQPainterFrm.PaintRotatedText
```

There are five things worth noting, here. First, you have to pass Canvas.Handle as the first parameter to the QPainter_ calls that expect a QPainterH. Second, the call to QPainter_translate() moves the coordinate system's *origin* from the top left of the form to the point we want to rotate around, the center of the form. This makes it much easier to place our rotated text, as rotation affects not just the way the text is drawn but the very meaning of the coordinates we pass to TextOut. (Try commenting out the call to QPainter_translate() and see if you can get all twelve strings to draw on the form.) Third, we can freely mix "raw" QPainter_ calls with Canvas operations like TextOut. Fourth, QPainter_rotate() takes an angle measured in **degrees** (not radians!) and rotates **clockwise**. Fifth, we don't need to call QPainter_save and QPainter_restore to protect other drawing operations from our coordinate transformations. Kylix brackets OnPaint event handlers with Canvas.Start and Canvas.Stop operations, which in turn call QPainter_begin and QPainter_end, which implicitly save and restore the coordinate system. You can, of course, use QPainter_save and QPainter_restore to push and pop coordinates during complex drawings, but you don't need to 'put things back as they were'.

The QPainter project in the ch6/Canvas project group has an action list that calls DrawRotatedText when you press CTRL+D. Outside of the different color, DrawRotatedText isn't very different from PaintRotatedText:

```
procedure TQPainterFrm.DrawRotatedText(Sender: TObject);
const
  Steps = 12;
var
  Index: integer;
begin
  Canvas.FillRect(ClientRect);
  Canvas.Font.Color := clRed;
  // Outside of a Paint handler, bracket QPainter_ calls with a Start/Stop
  Canvas.Start;
  try
    // Move 0,0 to the center of the form
    QPainter_translate(Canvas.Handle, Width div 2, Height div 2);
    for Index := 0 to Steps - 1 do
    begin
      Canvas.TextOut( 35, 0,
                    Format('Rotated %0.f degrees', [Index * 360 / Steps]) );
      // Rotate clockwise
      QPainter_rotate(Canvas.Handle, 360 / Steps);
    end;
  finally
    Canvas.Stop;
  end;
end; // TQPainterFrm.DrawRotatedText
```

All QPainter drawing operations have to be bracketed by a QPainter_begin and a QPainter_end. Kylix wraps this in Canvas.Start and Canvas.Stop, and calls Start and Stop around every Canvas drawing operation. In the OnPaint event handler, I didn't have to explicitly call Canvas.Start and Canvas.Stop, because Kylix wraps those around the event handler. But Kylix **doesn't** wrap most event handlers with Canvas.Start and Canvas.Stop because that would be inefficient, and because it has no way of knowing which Canvas you might draw on. So, when we call raw QPainter routines outside of an OnPaint event (or if we are modifying a QPainter from within another QPainter's OnPaint handler) we have to call Canvas.Start and Canvas.Stop. If you comment out the Canvas.Start and Canvas.Stop lines in DrawRotatedText, the text will not be rotated or translated, and all twelve strings will be drawn on top of each other in the top-left of the form.

Note that Canvas.Start and Canvas.Stop are implemented using a "start count" approach that means that you can safely nest Start/Stop pairs. Thus, the Start/Stop within Canvas.TextOut doesn't affect the Start/Stop around the translation and rotation. Similarly, the QPainter project would work just fine if you set the form's OnPaint handler to DrawRotatedText, which explicitly calls Canvas.Start and Canvas.Stop, instead of PaintRotatedText, which doesn't.

Note

It's a good idea to call Canvas.Start and Canvas.Stop around ***every*** *operation that calls QPainter_ directly, even if it's in an OnPaint event handler. The incremental cost is very modest, and you can then use the same code in paint and other event handlers. (Not to mention that people reading your code–which might be you in five years–don't have to wonder why you call Start/Stop* here *but not* there*.) I deliberately don't call Start/Stop in PaintRotatedText because I want to contrast PaintRotatedText and DrawRotatedText–PaintRotatedText is hardly an example of Best Practices!*

Finally, it's probably obvious that I've only scratched the surface of what you can do with QPainter coordinate transformations. The Qt documentation includes an example of how easy coordinate transformations make drawing a clock. Similarly, translation and scaling would simplify any line or bar charting.

Shearing does translation along one axis by an amount that increases linearly with another axis; it looks as if your drawing was composed of layers that slide over each other. It's a standard coordinate transform—Windows and Java offer shearing, too—but I suspect that this is because it's mathematically trivial, not because it's incredibly useful. You can use shearing to synthesize an (ugly) italic font from an upright font (as in the CTRL+S action of the QPainter project), but shearing's neither a perspective transform nor a z-axis rotation transform.

I've included an Object Pascal wrapping of the coordinate transformation parts of the QPainter API in lib/QGrabBag. (See the ITransform interface, and the Transform() function.) You can use the raw QPainter_ calls if you like, but I think you'll find that my thin object layer makes for clearer code—look at the difference between the rotated and sheared text procedures, where code like

```
QPainter_translate(Canvas.Handle, Width div 2, Height div 2);
```

gets replaced by

```
QPainter.Translate(Width div 2, Height div 2);
```

Forms

So far in this chapter, I've talked a lot about controls and canvases, but not much about the forms they sit on. This section is all about forms.

Opening and closing forms

When you or your users run a Kylix application, the Main Form (as defined on the Forms tab of the Project Options dialog) is shown. If you want any other forms to show, you have to make that happen.

Forms can be shown either modally or modelessly. Modal forms take precedence over other forms; when a modal form is open, none of the other forms in the application can be activated (brought to the front and given keyboard focus) until the modal form is closed. When a modeless form is active, you can activate other modeless forms. Modal forms are useful for

About boxes, yes/no popups, and some configuration dialogs, while modeless forms are more suitable for the various parts of an application that you may need at any time but not all the time. (The various windows in the Kylix IDE—the main window, Code Editor, Object Inspector, Project Manager, Alignment Palette, and the various debug windows are all modeless forms.)

The same form can be modal or modeless at different times. What controls modality is how you make the form visible. When you call ShowModal, the form is opened modally, and control doesn't return to the calling code until the form is closed. When you call Show, or set Visible to True, the form is opened modelessly; control returns to the calling code as soon as the new form is visible. Modal forms are a bit easier to deal with, as the same routine can open them and proceed to deal with any information entered on them. The Execute procedure of the standard dialogs is basically a wrapper for ShowModal that returns True when the OK or Open buttons are pressed.

If you ever need to know whether a particular form is modal or modeless, you can examine its FormState: if `fsModal in FormState`, then the form is modal. If not, it's modeless.

Modal forms can be closed by calling their Close method, by the user clicking on the frame's close button, or by setting their ModalResult property to a non-zero value. (TButton's and TBitBtn's can be configured to set their form's ModalResult automatically.) ShowModal is a function, which returns the ModalResult property of the form. Thus, you can write either

```
ThisForm.ShowModal;
if ThisForm.ModalResult = mrOK then {whatever};
```

or

```
if ThisForm.ShowModal = mrOK then {whatever};
```

Modeless forms can be closed by calling their Close method, by calling their Hide method, by setting Visible to False, or by the user clicking on the frame's close button. The visual effect of all these is the same, but Hide doesn't invoke the OnCloseQuery and OnClose event handlers. The OnCloseQuery event lets you prevent a form from being closed (perhaps it contains invalid required data) and the OnClose event lets you control *how* the form is closed.

Note that there is a difference between closing a form and freeing the form object! Normally, when a form is closed, that's all that happens. The GUI window goes away, but the object stays there, with all its data still valid. You can call the object's methods to extract data from it; you can bring it up again.

The form's OnClose event handler lets you control what happens when the form is closed, no matter how it's closed. The default is to just close the form, but you can set the `var Action: TCloseAction` parameter to either

prevent the form from closing, minimize instead of closing, or close and then free the form object. Note that you will normally use this last, caFree close action only with modeless forms—code following a ShowModal often extracts some data from the form object, and it can't do this if the object has been freed—but Kylix has no mechanism to keep you from setting `CloseAction := caFree` in a modal form's OnClose event handler.

It's common for infrequently used popups to be created and destroyed in the routine that shows them modally. This sort of code often uses the idiom I used in the Bitmaps project in the ch6/Canvas project group:

```
class procedure TBitmapFormatsFrm.Popup;
begin
  with TBitmapFormatsFrm.Create(Nil) do
  try
    ShowModal;
  finally
    Release;
  end;
end; // TBitmapFormatsFrm.Popup
```

Note the way that the `with TBitmapFormatsFrm.Create(Nil) do` allowed me to create a form without declaring a form variable. This is a mildly controversial practice. The vocal minority who find with statements confusing are rendered apoplectic by this "anonymous object". Most find it perfectly clear, and appreciate the avoidance of boilerplate code. For what it's worth, Borland uses this "with Create" idiom throughout the CLX and VCL.

Note

More importantly, note how I call `Release`*, not* `Free`*. You should always call Release to free a form object, not Free. Release waits for all event handlers on the form or its components to finish executing before freeing the form object. Using Free with a form may lead to intermittent SIGSEGV's.*

When you Release an open form, it will be closed before it's freed. This will trigger the OnClose event. Be careful never to use an OnCloseEvent handler that sets `Action := caFree` in a form that will be explicitly Release()d: freeing an already free form is just as much of an error as re-freeing any another sort of object.

Finally, notice how TBitmapFormatsFrm.Popup is a class procedure of the form itself. This keeps the knowledge of what's involved in popping up the form (which, admittedly, isn't much in this case) local to the form object—and saves code, if Popup is called from two or more places.

Form variables

Whenever you create a form, Kylix creates a form variable for it. A form named Foo has a form object type named TFoo and a form variable of type TFoo named Foo. By default, all forms in a project are created at load time, and their form variables are set to point to them. Only the main form is made visible, but all the others are ready to Show or ShowModal. Thus, you can bring up the form named Foo from the main form (or any other form) by including its unit in a uses clause in the other form's unit, and coding `Foo.Show` or `Foo.ShowModal`.

This is wonderfully convenient for small, simple apps, but it can make for excessively long load times for larger apps. Large apps can also end up wasting a lot of system resources on windows that are only brought up infrequently. Thus, you can control the list of auto-create forms on the Project Option dialog, and you can choose in Environment Options to change the default new form behavior to manual create. If you select manual creation, only the first form in an application is auto-created.

When a form besides the main form comes and goes a lot, it can make sense to create it the first time it's needed, and then leave it around. Thus, you'll sometimes see code like

```
if not Assigned(Frequent) then
  Frequent := TFrequent.Create(Application);
Frequent.ShowModal;
```

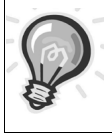
Similarly, one commonly needs to assure that there is no more than one instance of a form (or other object) at a time. This "singleton pattern" is usually implemented by replacing the public form variable with a form function that refers to a hidden (implementation section) form variable:

```
var
  FSingleton: TSingleton;

function Singleton: TSingleton;
begin
  if not Assigned(FSingleton) then
    FSingleton := TSingleton.Create(Application);
  Result := FSingleton;
end; // Singleton
```

However, manually created forms often don't use the form variable at all. For example, most About boxes are created using the anonymous object idiom I just used in TBitmapFrm.FormatsBtnClick. Similarly, many modeless forms are created and shown in a single operation as `TModeless.Create(Application).Show`, and rely on a caFree CloseAction to Release their memory when done.

Tip



If you don't use the form variable, you should go ahead and delete it. This will save four whole bytes of global data space and (much more importantly) reduce possible confusion.

One time where you need to be especially careful with form variables is when you have multiple copies of a form. (Kylix's "New Code Window" is an example; you can have as many different tabbed code editors open as you can keep track of.) Just as with any other object class with multiple instances, each form copy shares code but also has its own form object. Most code in event handlers and such will refer to Self, and so will have no problem with multiple instances, but you do want to be sure that you don't do something stupid like assign each new instance to the form variable as you create it. Generally, you would delete the form variable, and store references to each form in a variable length list (see Chapter 7).

Inter-system differences

Kylix is not Delphi

Windows is a much more uniform environment than Linux. Under Delphi, when you maximize a window, it will not cover the system tray, and in fact you have to go through some contortions to get a true full-screen window. Under Kylix, your form's maximize behavior will depend on the user's window manager. That is, on some systems, windows will maximize as on Windows, and not cover system windows like panels and task bars. On other systems, windows will maximize to the full-screen size.

This is just one more of those things, like BorderStyle and BorderIcons, that you can't count on.

One thing that you **can** count on is that Screen.PixelsPerInch (see Chapter 7) will vary from system to system. (According to the Qt documentation, this is "usually" a function of dot count divided by monitor size—but I found that my PixelsPerInch changed when I upgraded from Redhat 6.1 to Redhat 7.0 and installed a different set of X fonts.) This matters, because your design time PixelsPerInch is saved in the form resource if the form's Scaled property is True. At load time, Kylix will attempt to resize your Scaled forms so that they look about as big on the users' monitor as they did on yours. Thus, if you design a 300 pixels wide form on your 75 PixelsPerInch monitor, it will come up as 384 pixels wide on a 96 PixelsPerInch monitor.

The controls usually scale just fine. However, fonts never scale as smoothly. The closest match to the scaled font size may actually be bigger or smaller than the scaled size, so the text may be proportionately bigger or smaller than it was at design time. This may not matter for very simple forms. You can just leave enough white space that a 20% jump in relative font size (see the *Font* section, above) won't break things. That's a lot of white space, though, and is usually not possible on more complex forms.

Most people set Scaled to False on all but the simplest dialogs. This means that their forms look smaller on high dot count monitors than on low dot count monitors, but users are used to that, and it's generally better than having either tiny text or huge, clipped text.

Form events

Like any other component, forms have events. Some are completely generic, like the OnMouseX events. Others are generic but behave slightly differently on forms than other controls, like the OnKeyX events; others are unique to forms, like OnCreate. This sub-section is about the events that are unique to forms, with a brief description of keyboard events at the end.

A form's OnCreate and OnDestroy events are analogous to its constructor and destructor,[10] and essentially complementary. You can put setup and teardown code in either the constructor and destructor or in the OnCreate and OnDestroy events, but you generally choose either one or the other, not both. Most people use the OnCreate and OnDestroy event handlers, not the constructor and destructor, simply because it's easy to jump to them from the Object Inspector. About the only time you really will use a form's constructor is when you need something to happen **before** the `inherited Create`. This is not a very common event.

The OnCreate event is a good place to populate various components, like the ch6/Fonts project's `ScreenFonts.Items := Screen.Fonts`. The OnCreate event is also a good place to Create any non-component objects, like string lists or semaphores. Just as with class constructors and destructors, I try to always create an OnDestroy event at the same time I create an OnCreate event, and always add a Free line to OnDestroy as I add a Create line to OnCreate. This small bit of self-discipline goes a long way toward preventing memory leaks.

The OnShow and OnHide events are called when a form is opened (whether by calling Show or ShowModal or by setting Visible to True) and closed (whether by calling Hide or Close or by setting Visible to False). If a form is freed as soon it's closed, it doesn't much matter whether you put setup code in OnCreate or in OnShow. However, if a form is closed and re-opened, OnCreate will be called once, while OnShow will be called several times. It thus makes sense to restrict OnCreate to initialization (creation of non-components, and population of components that won't change) while OnShow should be used only for re-initialization (resetting any state that might change while the form is open).

When you show a form, OnActivate is called after OnShow. It's then called whenever control is transferred to this form from another form in the application. That is, imagine you have two modeless forms open, *This* and *That*, with *This* being active, the one with the keyboard focus and the highlighted caption bar. When you click on *That*, it becomes active: *This*'s OnDeactivate event fires, then *That*'s OnActivate event fires.

10. For that matter, OnCreate is called from TCustomForm.Create, while OnDestroy is called from TCustomForm.Destroy.

Note

Kylix distinguishes between a transfer of activation within the application, and a transfer of activation from another application. A form's OnActivate is only fired on an intra-application transfer; on inter-application transfers, the Application.OnActivate event is fired, instead. (See the Application *section of Chapter 7.)*

Finally, I've already mentioned the OnClose and OnCloseQuery events, which are called whenever the form is being closed. The OnCloseQuery event allows you to abort the close operation; the OnClose event allows you to control how the form is closed. If neither interferes with the closing process, they are followed by an OnDeactivate event, an OnHide event, and then (if CloseAction was set to caFree) by an OnDestroy event.

KeyPreview

Normally, keyboard events go to the form's ActiveControl. If you want to see keystrokes in a form-wide event handler, you need to set the form's KeyPreview property to true.

Under Delphi, this works no matter what BorderStyle you have selected. However, Qt has this strange notion that borderless windows are not interactive, and so don't get the keyboard focus. Fortunately, you can override this simply by including the line

Kylix is not Delphi

```
QWidget_setActiveWindow(Handle);
```

in the form's OnShow handler. (See the date picker component in Chapter 9 for an example.) Note that you only need to do this if BorderStyle = fbsNone! I suspect that it can't hurt to call QWidget::setActiveWindow in other cases, but I think of this as a hack and a kluge that limits portability, and prefer to use it only where necessary.

Enter as tab

One common reason to use KeyPreview is to treat the Enter key like the Tab key, and use it to move to the next active control. This is a common requirement in heavy-duty data entry applications, where the operators will be using the application day in and day out, and every keystroke counts. Particularly

when using the number pad, it can be a lot faster to press Enter than to reach over and press Tab. This code from the EnterAsTab project of the ch6/Forms project group will do the trick:

```
procedure TEnterAsTabFrm.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  // Only works if KeyPreview is True!
  if (Key = Key_Return) or (Key = Key_Enter) then
  begin
    Key := 0;
    SelectNext(ActiveControl, not (ssShift in Shift), True);
  end;
end; // TEnterAsTabFrm.FormKeyDown
```

Note the way it sets Key to 0, to suppress further handling. The OnKeyUp event will still be called when the key is released, but the OnKeyPress event will **not** be called with a Key = ^M.

Forms are objects

One unfortunate aspect of both Kylix and Delphi is that controls are part of the `published` interface of the form object. This means that they're visible to code in any unit that uses the form's unit. You'll sometimes see people write code like `ThisForm.Edit1.Text := ThatForm.Edit1.Text`.

Don't do this!

Remember that forms are objects, and **maintain encapsulation**. Pretend that the controls aren't really visible, and read and write any data *via* public properties and methods. This is good for the same reason that encapsulation is good: It limits the effects of changes. If you change the way a particular datum is presented on the form, you only have to change one access function, not all the code throughout the system that relied on object internals.

For example, if both ThisForm and ThatForm had a

```
public
  property CurrentFilename: string read  GetCurrentFilename
                                   write SetCurrentFilename;
```

which was implemented *via* the private routines

```
function TThisForm.GetCurrentFilename: string;
begin
  Result := Edit1.Text;
end; // TThisForm.GetCurrentFilename

procedure TThisForm.SetCurrentFilename(const Value: string);
begin
  Edit1.Text := Value;
end; // TThisForm.SetCurrentFilename
```

code like `ThisForm.Edit1.Text := ThatForm.Edit1.Text` could be replaced by `ThisForm.CurrentFilename := ThatForm.CurrentFilename`. Even if the "Edit1" control had a more descriptive name, the code that reads the form property is simpler and clearer than the code that reads a property of a control on the form. And, of course, if TThisForm needed any changes to Edit1 to be mirrored *here* or *there*, it's better to do this mirroring in one place—the SetCurrentFilename method—than all throughout the application.

Class methods

How far do you take encapsulation? Sure, public properties are an abstraction of the data in the controls, a contract with the rest of the system that's less likely to change than the specific controls on the form, but doesn't reading and/or writing a handful of properties constitute embedding a lot of knowledge about the object into the rest of the system?

Frankly, I go back and forth on this. I do find that when a particular dialog has a narrow purpose—perhaps getting a Yes/No answer, or a name and password—that it makes a lot of sense to provide public access to the form through a class procedure or a class function which is responsible for both creating and popping up the dialog, and which returns the results either as a function result or *via* var parameters.

Note

A simple example of this is the `class procedure TBitmapFormatsFrm.Popup` *(in the ch6/Bitmaps project) that I listed in the section on* Opening and closing forms*, above.*

This means that the code that invokes this dialog knows only the one or two functions that pop it up, not the handful of properties that configure it and the handful of properties that represent the results. The calling code is smaller and clearer and, if the dialog itself changes, the narrower interface is less likely to get out of synch than a handful of inter-related properties.

Internal modularity

Forms are funny objects. Where other objects are tightly focused, modeling a single entity or presenting a single service, forms have diffuse responsibilities. They're a visual representation of often heterogeneous data, and they need to be able to read and write data models; they have UI methods, and often they have various bits of UI state data. This is unavoidable. What belongs together on a form is what makes the application easiest to use. Just as object modeling can be difficult because what initially seems a natural unit may actually be an ensemble of related concepts, so the clean modular lines of a good model may not have much correlation with what belongs together in an intuitive user interface.

Your first response to this issue should be to maintain a distinction between a *data model* and a *view* of the model. The data model is the set of objects that represent the real world phenomena that you're modeling; your user interface is a view of that model. Don't write event handlers that modify your data model directly; call methods, or read and write properties of the model's objects. Maintaining this distinction yields the same benefit that encapsulation always does: if the internals of the model change, you only have to change the access points, not part of *this* dialog and part of *that* dialog.

Maintaining a model/view distinction still leaves the view with a lot of unrelated UI code and state data. The event handlers on *this* tab aren't going to fire when *that* tab is visible; the UI state data for *this* tab is never going to have any correlation with the UI state data for *that* tab. The best response to this muddle seems to me to be to remember that in some ways object oriented languages just provided syntax to make easy what was already good practice: Modularize. Break your code into subsystems, and don't let the left hand know what the right hand is doing. Don't give *this* functional group knowledge of how *that* functional group works: Create a private method for it to call.

This is a hard subject to write about, because you really can't illustrate complexity with simple examples. So, I'm mostly going to wave my hands about and hope you understand. The way the vfind projects in Section 4 decompose routines into private methods is a partial example, as is the PrivateProperties project in the ch6/Forms project group. This is a rather useless little demo program with a couple of unexceptional private methods:

```
function TPrivatePropertiesFrm.GetPanelText(N: integer): string;
begin
  Result := StatusBar.Panels[N].Text;
end; // TPrivatePropertiesFrm.GetPanelText

procedure TPrivatePropertiesFrm.SetPanelText(N: integer;
  const Text: string);
begin
  StatusBar.Panels[N].Text := Text;
end; // TPrivatePropertiesFrm.SetPanelText
```

What's interesting about these methods is these three property declarations in the private section:

```
property Filename:  string index 0 read GetPanelText write SetPanelText ;
property FileLines: string index 1 read GetPanelText write SetPanelText ;
property Caret:     string index 2 read GetPanelText write SetPanelText ;
```

These declarations give names to the various status bar panels. As the form state changes, routines like

```
procedure TPrivatePropertiesFrm.ShowCaret;
begin
  with Memo.CaretPos do
    Caret := Format('%d:%d', [Line + 1, Col + 1]);
end; // TPrivatePropertiesFrm.ShowCaret
```

can update the status bar without 'knowing' that that's what they're doing. If I need to change the panel assignments, I just have to change the index constants in the property declarations.

I find that I use this approach whenever I have a paneled status bar. You can also use this indexed property approach to give names to the various parts of any other indexed component: the header row of a string grid, the columns of a list view, and so on. More generally, this is an example of hiding the details of how one part of the form works from the other parts, thus increasing legibility and conserving flexibility.

Interfaces and forms

Forms can implement interfaces, just like any other object can. By default, however, they implement them **without reference counting**. This is done to avoid the problems of mixing object and interface references that I discussed

in Chapter 3. If you create an interface reference to a form that implements an interface, the form will not Free itself when the interface reference goes away.

I think this is a sensible default behavior, and of course we're still free to add reference counting when that's the behavior we want. For an illustration of some of the differences between normal forms and reference counted forms, see the InterfacesAndForms project of the ch6/Forms project group. (See Figure 6-12.)

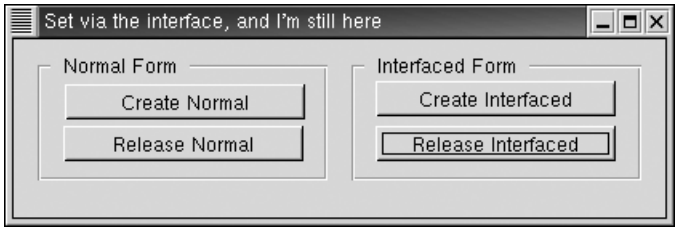


Figure 6-12. The InterfacesAndForms project

Interfaced forms in Delphi

"Old" Delphi programmers may be familiar with the problem of adding interfaces to forms; forms (all components, for that matter) implement the three methods of IUnknown (which is called IInterface in Kylix), but the implementation relies on a VCLComObject property that's normally Nil. If you add an interface to a form without supplying a VCLComObject or overriding this implementation, your applications will generate Access Violations (SIGSEGV, in Linux-speak) as soon as you try to execute `Form as IUnknown`.

Supplying a VCLComObject is not totally trivial, so most people just cribbed some code from TInterfacedObject to create a TInterfacedForm. You could then have reference-counted forms, or (carefully!) mix object and interface references.

Familiar, yes? But out of date. Since Delphi 4, controls without a VCLComObject implement IUnknown without reference counting. That is, the reference count is always –1,[11] and neither _AddRef nor _Release ever change it—nor does _Release ever call Free. (Kylix, of course, does this the same way.) This means that there are no problems mixing object and

11. Just as it is for string constants.

interface references for TComponents. You can pass a direct descendant of TForm (that implements one or more interfaces) to any routine that expects one of those interfaces as a parameter, without having to manually _AddRef to avoid a disaster when the procedure returns. If you **want** a reference-counted form, you can still use TInterfacedObject-based code—but you no longer have to.

The main form is just a normal TForm that implements the ISetCaption interface—`class(TForm, ISetCaption)`–yet it can call `(Self as ISetCaption).SetCaption` without courting disaster by not calling _AddRef first. The main form isn't reference counted.

The "normal" form isn't reference counted, either. Each time you click on the Create Normal button, you create a new form. Assigning a new value to the Normal interface variable **does** dereference the old value, but this doesn't free the object. Similarly, clicking on the Release Normal button has no visible effect.

```
procedure TForm1.CreateNormalBtnClick(Sender: TObject);
begin
  Normal := TFormWithInterface.Create(Application) as ISetCaption;
  Normal.SetCaption('A normal form, with an interface');
end; // TForm1.CreateNormalBtnClick

procedure TForm1.ReleaseNormalBtnClick(Sender: TObject);
begin
  Normal := Nil;
end; // TForm1.ReleaseNormalBtnClick
```

The code behind the "Interfaced" buttons is virtually identical[12] but since the "interfaced" form **is** reference counted, the buttons act very differently. Each time you click on the Create Interfaced button, you get a new form that replaces any existing Interfaced form; when you click on the Release Interfaced button, the form goes away.

12. Yes, that does mean "I don't think it's worth killing trees to print it."

Splash screens

Many applications have splash screens. These are a window that comes up while the application is initializing, often with a cool graphic, that go away by themselves after a few seconds. These pose a modest difficulty for Kylix applications.

The first auto-create form is the Main Form. When the main form closes, the application closes. The process is terminated, and any other open windows are closed. If we make our splash screen the main form, we can hide it (without closing it) when we're done, but then we've lost the main form semantics; when we close the true main form, the application itself doesn't close. We could probably fiddle around with the Application object (see below) but this is unappealing. Such fiddling has a way of being dependent on the internal workings of a particular version. We don't want to have to rewrite our splash screen code every time Borland redoes some framework code. Similarly, we could have the 'real' main form close the splash screen when it closes, but while this is better, it's still a bit fragile (what if the main form changes?) and it requires that the main form know about the splash form. The simplest solution would keep all the 'magic' in the splash screen unit itself.

The SplashScreen project in the ch6/Forms project group illustrates just such a simple solution that you can drop into your own projects. It's based on an ancestral splash form in lib/GenericSplashscreen.pas, which supplies almost all the actual splash screen logic. All you need do to create your own splash screens is to inherit from TGenericSplashFrm and add the look you want, and then add a couple of lines at the bottom of your new splash screen's unit like

```
initialization
  TSplashFrm.Create(Application).Show;
```

Unit initialization blocks run before the main program block in the program (.dpr) file, so this is functionally equivalent to editing the .dpr file (Project ➤ View Source) file, and adding the Create().Show line before the normal `Application.Initialize` first line

```
begin
  TSplashFrm.Create(Application).Show;
  Application.Initialize;
```

as some Delphi splash screen code you'll find on the Net suggests. However, not only do most people prefer to leave the .dpr file alone as much as possible (some of the automatic editing that Kylix does to it can get confused if you've

changed it yourself), I think it also makes a lot of Software Engineering sense to keep the special splash screen code as self-contained as possible. Why spread it through three files when two will do?

Calling `TSplashFrm.Create` instead of `Application.CreateForm(TSplashFrm, SplashFrm)` means only that the first form created with CreateForm—the main form—will **be** the main form. That is, the only difference between an auto-create form, created in the .dpr file *via* `Application.CreateForm(TFormName, FormName)` and a form manually created *via* `FormName := TFormName.Create(Application)` is that CreateForm is responsible for setting Application.MainForm.

The GenericSplashscreen unit is responsible for making sure that the main form doesn't come up until the splash screen times out. It's pretty short, so I'll include the whole file here:

Listing 6-1. GenericSplashscreen.pas

```
unit GenericSplashscreen;

interface

uses
  SysUtils, Types, Classes, Variants,
  QGraphics, QControls, QForms, QDialogs, QTypes,
  QExtCtrls, QStdCtrls;

type
  TGenericSplashFrm = class(TForm)
    Timer: TTimer;
    procedure TimerTimer(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
  public
  end;

implementation

{$R *.xfm}

procedure TGenericSplashFrm.TimerTimer(Sender: TObject);
begin
  Timer.Enabled := False; // allow form to close
  Close;
end; // GenericSplashFrm.TimerTimer
```

```
procedure TGenericSplashFrm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  CanClose := not Timer.Enabled;
end; // GenericSplashFrm.FormCloseQuery

procedure TGenericSplashFrm.FormClose( Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
  Application.MainForm.Show;
end; // GenericSplashFrm.FormClose

initialization
  Application.ShowMainform := False;
end.
```

The first thing to note in GenericSplashscreen.pas is actually the penultimate line of the file, in the initialization block:

```
  Application.ShowMainform := False;
```

Setting ShowMainForm to False means that Application.Run won't automatically Show Application.MainForm, the way it normally does. It will still be created as normal—and it can do any initialization it needs to do—and it will still be the MainForm, but it won't appear on the screen until the splash screen times out and calls `Application.MainForm.Show`.

That's the only real 'magic' here. The Timer starts running when the splash screen shows, just like any other TTimer. At the end of the Timer's Interval, it fires off its OnTimer event, which sets Enabled to False and Close's the form. Setting Enabled to False turns the timer into a one-shot event; it won't fire again until Enabled is set True. In this case, that doesn't matter, because Enabled is being used only to avoid having to declare a special TimerTicked flag for the OnCloseQuery event. I use Timer.Enabled to answer the close query. (This doesn't matter with borderless splash screens like the one in the ch6/SplashScreen project, but you could easily have a bordered splash screen if you wanted to.)

The OnClose event handler sets the close action to caFree, and calls Application.MainForm.Show, which then comes up as normal. When you close the main form, the application shuts down as it should.

Asynchronous processing

Kylix is not Delphi

Delphi programmers—and Windows programmers in particular—are used to the Windows message-driven model. Just about all events are ultimately traceable to a message that Delphi receives and massages for us; we control the behavior of controls like the edit and rich edit controls by sending messages, and we post messages to ourselves when we want to do asynchronous processing. If we have a lengthy operation that we want to start after OnShow and the first OnPaint, we PostMessage a custom message, and handle it with a `message` procedure. The message goes to the end of the queue, and gets processed once the app is up and running. Similarly, we commonly post update messages from thread code, for the foreground thread to display the results of a threaded calculation.

There's no PostMessage in Kylix.

Fortunately, Qt does have a message loop, and Qt does let us create custom messages. With just a little work, we can do the same sort of asynchronous, message-based processing in a Kylix app that we can do under Windows. This work is embedded in the lib/QMessages unit that the Asynch project in the ch6/Forms project group uses. My QMessages unit defines some functions to send and post messages to a form's Qt event loop, and a form that receives these messages. To use the QMessages framework, just create a new form that inherits from my TMessageForm.

QMessages declares procedures to post pointers, integers, strings, and interfaces, as well as routines that directly emulate the Windows SendMessage/PostMessage API, which allows you to send a wParam and an lParam. They all work similarly, so I'll just walk through a basic PostPointer and how it gets passed to its `message` handler, and leave you to read the code more closely if you like.

Caution

As under Windows, SendMessage dispatches the event, and returns the result code. The event is handled within the thread that calls SendMessage. Do not call SendMessage from a non-GUI thread! Use the various Post procedures to pass data from background threads to the GUI thread. You can always be sure that posting data will result in a `message` *handler being called asynchronously in the GUI thread.*

Qt's QCustomEvent allows you pass a numeric type code and a 32-bit pointer to the event loop. The PostQtPointer function constructs a QCustomEvent, then posts it (indirectly) to a form's event queue.

```
function PostQtPointer( Handle: QObjectH;
                        Msg: SimpleMessages; Data: pointer): boolean;
var
  CustomEvent: QCustomEventH;
begin
  Result := True;
  CustomEvent := nil;
  try
    CustomEvent :=
      QCustomEvent_create( QEventType(Cardinal(QEventType_User) + Msg),
                           Data );

    MessageQueue.Post(Handle, CustomEvent);
  except
    if Assigned(CustomEvent) then
      QCustomEvent_destroy(CustomEvent);

    Result := False;
  end;
end; // PostQtPointer
```

Now, it turns out that QApplication::postEvent is not thread-safe. The simplest way to make it thread-safe and retain the benefits of letting background threads run at full speed (*ie*, not wait for TThread.Synchronize to return) is to add another thread. MessageQueue.Post uses a critical section to add the Handle/Event pair to a queue in a thread safe way, and then sends a signal (*via* a TSimpleEvent) to the new thread. This new message queue thread spends most of its time waiting on the TSimpleEvent. When the event is signaled, the message queue thread uses Synchronize to call a method that posts each message in the queue. Since the synchronized method runs in the GUI thread, there's no danger of its calling QApplication::postEvent while Qt itself is. The only real consequence of this new message queue and its thread is Synchronize runs very slowly when integrated debugging (Tools ➤ Debugger Options) is turned on; applications that use QMessages will run much faster from the command line (or when integrated debugging is turned off) than when they run within Kylix's debugger.

PostQtPointer is a hidden (implementation section) function. You call it indirectly through the TMessageForm.PostPointer function or one of its siblings.

```
function TMessageForm.PostPointer(Msg: SimpleMessages; Data: pointer):
   Boolean;
begin
```

```
  Result := PostQtPointer(Handle, Msg, Data);
end; // TMessageForm.PostPointer
```

Now, under Windows, message handling is a more fundamental part of GUI programming than under Qt. Where Windows controls send messages to applications whenever anything happens, Qt controls use a *slot* and *signal* approach (see Section 3) that means that they don't send messages: they call the application directly. Because messages are less fundamental under Qt, the messages that we send ourselves with QApplication::postEvent don't automatically get dispatched to a `message` handler the way that messages that we send ourselves with PostMessage() do under Windows. Rather, we have to override the form's EventFilter function, and catch the type code there.

TMessageForm does this, and puts the type code and 32-bit Data into a record that it then hands to Dispatch. As *per* the Borland documentation, Dispatch then looks for a `message` handler with an identifier that matches the type code, and passes the record to it. Here's a somewhat abbreviated version:

```
function TMessageForm.EventFilter(Sender: QObjectH;
  Event: QEventH): Boolean;
var
  PointerMessage: TPointerMessage;
  EventType:      QEventType;
begin
  Result := True;
  EventType := QEvent_type(Event);
  case EventType of
   QEventType_CMPostSimpleLow..QEventType_CMPostSimpleHigh:
    begin
      PointerMessage.Msg  := Cardinal(EventType) - Cardinal(QEventType_User);
      PointerMessage.Data := QCustomEvent_data(QCustomEventH(Event));
      Dispatch(PointerMessage);
    end;
    else Result := inherited EventFilter(Sender, Event);
  end;
end; // TMessageForm.EventFilter
```

This EventFilter function means that forms that descend from TMessageForm only have to declare a `message` method, and call PostPointer() to pass a pointer to that method asynchronously. That is, the PostPointer call will return, and the calling code can do whatever else it has to do, and the message handler will be called as soon as any messages ahead of it in the queue are processed. Importantly, the message handler will be called in the

GUI thread, not in whichever non-GUI thread may have called PostPointer(), which means that the message handler can safely make GUI calls. (Threaded code can't do this; see Chapter 7.)

The following extracts from the ch6/Asynch project may make this a bit clearer

```
const
  qt_Pointer   = qtm_Simple + 3;

type
  TAsynchFrm = class(TMessageForm)
    Status: TLabel;
    PostPointerBtn: TButton;
    procedure PostPointerBtnClick(Sender: TObject);
  private
    procedure qtPointer(   var Msg: TPointerMessage);   message qt_Pointer;
  end;

procedure TAsynchFrm.PostPointerBtnClick(Sender: TObject);
begin
  inherited;
  PostPointer(qt_Pointer, pointer(Self));
end; // TAsynchFrm.PostPointerBtnClick

procedure TAsynchFrm.qtPointer(var Msg: TPointerMessage);
begin
  Status.Caption := 'Posted pointer was $' + IntToHex(integer(Msg.Data), 8);
end; // TAsynchFrm.qtPointer
```

Caution

`Message` *methods require only that their single parameter be a* `var` *parameter. They don't do any type checking.* `You` *are responsible for making sure that you post a pointer to a method that expects a TPointerMessage, or an integer to a method that expects a TIntegerMessage.*

Posting reference counted objects

Strings, interfaces, and dynamic arrays are actually just pointers to a reference counted data block, so we can Post them, too. This is a bit trickier than passing data that's not reference counted, because the data is dereferenced when the Post function returns, and without a bit of hackery it may not still be around when the message handler gets fired. To make sure that the data doesn't get freed, we have to increment the reference count before we post a pointer to the event queue.

With interfaces, of course, we can just use AddRef. Borland doesn't supply anything equivalent for strings, but the String_AddRef and String_Release functions in my lib/GrabBag are pretty straightforward. (See the sidebar for details.)

String_AddRef and String_Release

```
function String_AddRef(const S: string): pointer;
// increments ref count, returns pointer(S)
var
  Reference: string;
begin
  Result := pointer(S);      // Typically used where storing a raw pointer
  Reference := S;            // Increment reference count, unless < 0
  pointer(Reference) := Nil; // DON'T deref Reference on exit
end; // String_AddRef
```

The first line just assigns the address of the first character (if any—the string may be '') to the Result. This is just a convenience feature, since the most typical use of this function will be to pass a string *via* an untyped pointer, as we're doing here; to a callback function's Data parameter; or to a TCustomViewItem's Data parameter. (For an example, see the TMessageForm.PostString procedure in lib/QMessages.)

The real work of the function is done in the next two lines. Copying the string to a local variable increments the reference count, if any. (Empty strings aren't reference counted, obviously, nor are string constants.) By casting the local to pointer, we bypass the reference counting mechanism; by setting the pointer to Nil, we assure that the string will not be dereferenced when the function exits and the string goes out of scope.

```
procedure String_Release(const S: string);
var
  Reference: string;
begin
  pointer(Reference) := pointer(S); // Assign without changing reference
                                    // count; reference will be derefed on
                                     // exit, unless ref count < 0
end; // String_Release
```

Again, by assigning to `pointer(Reference)` and not simply to `Reference`, we bypass the reference counting mechanism. The effect is to set the local string to the passed in string. When the procedure exits, the string goes out of scope, and the string is dereferenced.

Note

String constants are stored in the application's code pages with a reference count of –1. This is a special value that means that the string constant doesn't live on the heap as other strings do. String constants are not reference counted the way string values are: The reference count is never incremented, however many copies are made; nor is the reference count ever decremented when a copy goes out of scope. Because the string constant doesn't live on the heap it doesn't ever need to be freed.

It's not possible to write code that works with any dynamic array, so QMessages doesn't offer any PostArray support.

The code to Post an interface is

```
function TMessageForm.PostInterface(Msg: SimpleMessages;
  const Data: IInterface): Boolean;
begin
  Data._AddRef; // So it doesn't get destroyed before EventFilter sees it
  Result := PostQtPointer(Handle, Msg, pointer(Data));
end;
```

and the corresponding bit from the TMessageForm.EventFilter case statement is

```
QEventType_CMPostInterfaceLow..QEventType_CMPostInterfaceHigh:
begin
  InterfaceMessage.Msg  := Cardinal(EventType) - Cardinal(QEventType_User);
  pointer(InterfaceMessage.Data) := QCustomEvent_data(QCustomEventH(Event));
  Dispatch(InterfaceMessage);
end;
```

Explicitly adding a reference means that the strings and interfaces 'know' that there's a reference to them in the event queue. QMessages defines separate message code ranges for strings, interfaces, and "simple" 32-bit data; this lets the EventFilter know if a given 32-bit value needs to be assigned to a string or interface message record. As in the String_AddRef and String_Release routines, these assignments are *via* pointer casts that don't touch the reference count. You can think of the EventFilter code as redeeming the pledge made by the explicit _AddRef in the Post call, or you can think of it as the Post call making an assignment to a record in the EventFilter.