

Qt Designer Manual

Qt 3.0

Copyright © 2001 Trolltech AS. All rights reserved.

TROLLTECH, Qt and the Trolltech logo are registered trademarks of Trolltech AS. Linux is a registered trademark of Linus Torvalds. UNIX is a registered trademark of X/Open Company Ltd. Mac is a registered trademark of Apple Computer Inc. MS Windows is a registered trademark of Microsoft Corporation. All other products named are trademarks of their respective owners.

The definitive Qt documentation is provided in HTML format supplied with Qt, and available online at <http://doc.trolltech.com>. This PDF file was generated automatically from the HTML source as a convenience to users, although PDF is not an official Qt documentation format.

Contents

Preface	3
Creating a Qt Application	6
Creating Main Windows with Actions, Toolbars and Menus	18
The Designer Approach	28
Subclassing and Dynamic Dialogs	33
Creating Custom Widgets	42
Creating Database Applications	52
Customizing and Integrating Qt Designer	66
Reference: Key Bindings	73
Reference: Menu Options	75
Reference: Toolbar Buttons	83
Reference: Dialogs	88
Reference: Wizards	116
Reference: Windows	126
Index	130

Preface

Introduction

This manual presents *Qt Designer*, a tool for designing and implementing user interfaces built with the Qt cross-platform application development framework. *Qt Designer* makes it easy to experiment with user interface design. At any time you can generate the code required to reproduce the user interface from the files *Qt Designer* produces, changing your design as often as you like. If you used the previous version you will find yourself immediately productive in the new version since the interface is fundamentally unchanged. But you will find new widgets and new and improved functionality which have been developed as a result of your feedback.

Qt Designer helps you build user interfaces with layout tools that move and scale your widgets (*controls* in Windows terminology) automatically at runtime. The resulting interfaces are both functional and attractive, comfortably suiting your users operating environments and preferences. *Qt Designer* supports Qt's signals and slots mechanism for type-safe communication between widgets. *Qt Designer* includes a code editor which you can use to embed your own custom slots inside the generated code. Those who prefer to separate generated code from hand crafted code can continue to use the subclassing approach pioneered in the first version of *Qt Designer*.

The manual introduces you to *Qt Designer* by leading you through the development of example applications. The first six chapters are tutorials, each designed to be as self-contained as possible. Every chapter, except the first, assumes that you are familiar with the material in chapter one which covers the basics of building a Qt application with *Qt Designer*. Here's a brief overview of the chapters:

- Chapter one, Creating a Qt Application, introduces *Qt Designer* and takes you step by step through the creation of a small but fully functional application. Along the way you will learn how to create a form and add widgets to it. In the course of this chapter you will use the form and property editors to customize the application, and learn how to lay out a form using the layout tools. You'll also learn how to use Qt's signals and slots mechanism and *Qt Designer's* built-in code editor to make the application functional. We will also explain how to use `qmake` to generate a Makefile so that you can compile and run the application.
- In chapter two, Creating Main Windows with Actions, Toolbars and Menus, we will create a simple text editor. Through writing this application you will learn how to create a main window with menus and toolbars. We will see how to use Qt's built-in functionality to handle common tasks (e.g. copy and paste handling), and how to create our own functionality for our own menu items and toolbar buttons.
- Chapter three, The Designer Approach, provides information on the *Qt Designer* approach to developing applications, and explains some of the rationale behind *Qt Designer*.
- Chapter four, Subclassing and Dynamic Dialogs, will show you how to subclass a form; this allows you to clearly separate the user interface from the underlying code that implements its functionality. Additional information on `qmake` and `uic` is included in this chapter. This chapter will also explain how you can dynamically load dialogs from `.ui` files into your application using **QWidgetFactory** and how to access the widgets and sub-widgets of these dialogs.
- Chapter five, Creating Custom Widgets, explains how you can create your own custom widgets. Both the simple method, that was introduced with the first version of *Qt Designer*, and the new more powerful method using plugins, are explained.
- Chapter six, Creating Database Applications introduces Qt's SQL classes and takes you through an example that demonstrates how to execute SQL queries and how to set up master-detail relationships, perform drilldown and handle foreign keys.

- Chapter seven, Customizing and Integrating Qt Designer, focuses on *Qt Designer* itself, showing you how to customize Designer, how to integrate Designer with Visual Studio and how to create Makefiles.

The remaining chapters provide reference material that explains *Qt Designer's* menu options, toolbars, key bindings and dialogs in detail.

What You Should Know

This manual assumes that you have some basic knowledge of C++ and the Qt application development framework. If you need to learn more about C++ or Qt there are a vast number of C++ books available, and a small but increasing number of Qt books. Qt comes with extensive online documentation and many example applications that you can try.

The Enterprise Edition of Qt includes the Qt SQL module. In Creating Database Applications we demonstrate how to build SQL applications with *Qt Designer*; this chapter requires some knowledge of SQL and relational databases.

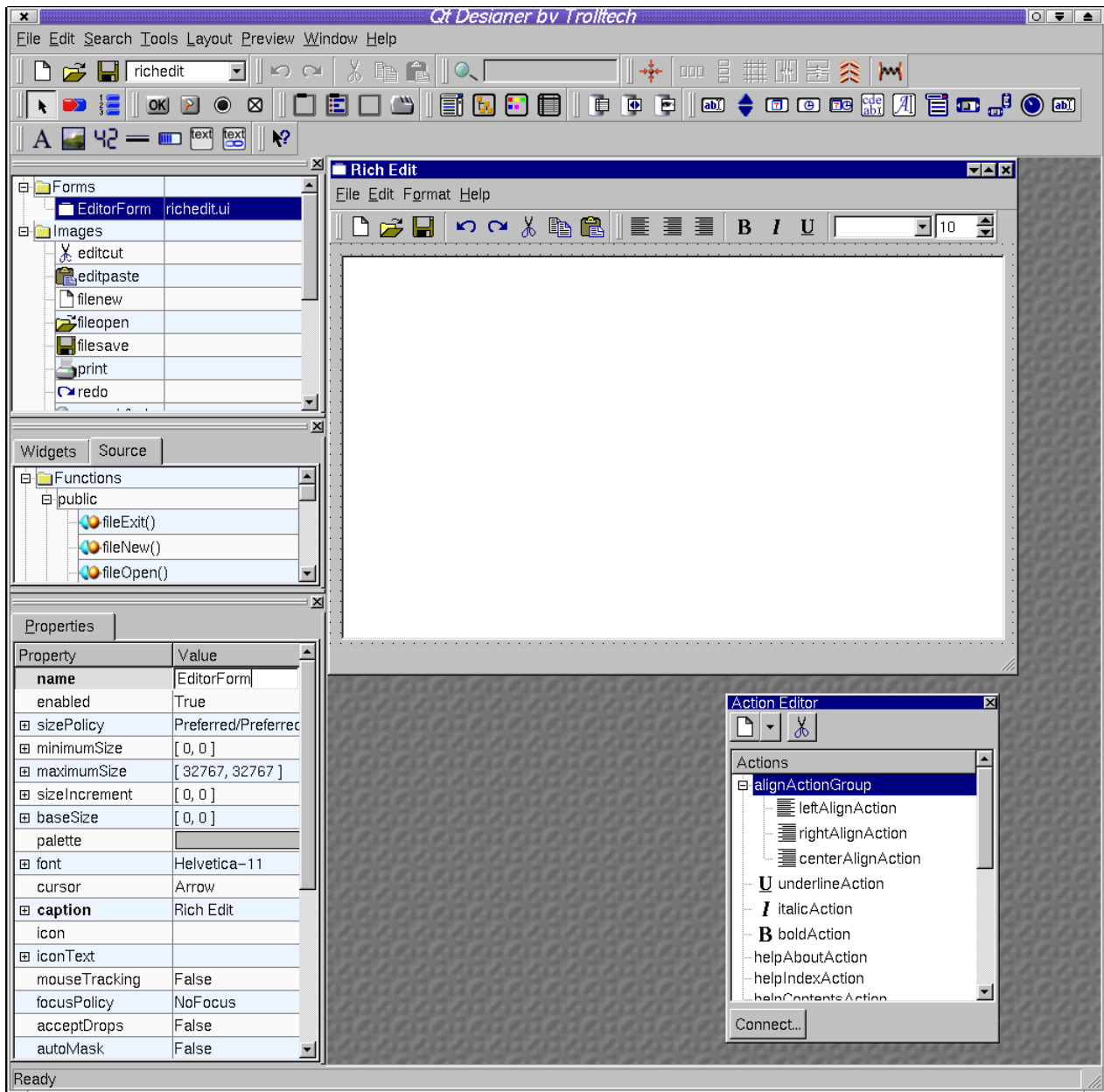
What's New in *Qt Designer*?

This version of *Qt Designer* has a great deal more functionality than its predecessor. The code for custom slots can be edited directly in *Qt Designer*; main windows with actions, toolbars and menus can be created; layouts that incorporate splitters can be used; plugins allow you to package any number of custom widgets and make them available to *Qt Designer*. Many other enhancements have been incorporated, from small improvements in the user interface to improved efficiency, for example the ability to share pixmaps across all the forms in an application.

This version of *Qt Designer* introduces project files which make it easy to switch between all the forms in an application, and to maintain a common set of database settings and images. Although subclassing is fully supported, writing code directly in *Qt Designer* offers a number of advantages which are covered in The Designer Approach chapter.

A new library `libqui` has also been introduced which allows you to load dialogs dynamically at runtime from *Qt Designer's* `.ui` files. This allows you to provide your application's users with considerable interface customizability without them needing to use C++.

Although the new version of *Qt Designer* introduces new approaches and techniques you can ignore these aspects and simply use it in exactly the same way as you used the version supplied with Qt 2.x, if you just want a simple but powerful single dialog visual design tool.



Qt Designer

Feedback

If you have any comments, suggestions, criticisms or even praise regarding this manual, please let us know at doc@trolltech.com. Bug reports on Qt or *Qt Designer* should be sent to qt-bugs@trolltech.com. You might also like to join the *qt-interest* mailing list, which *Qt Designer's* developers read and contribute to; see <http://www.trolltech.com> for further details.

Creating a Qt Application

Starting and Exiting Qt Designer

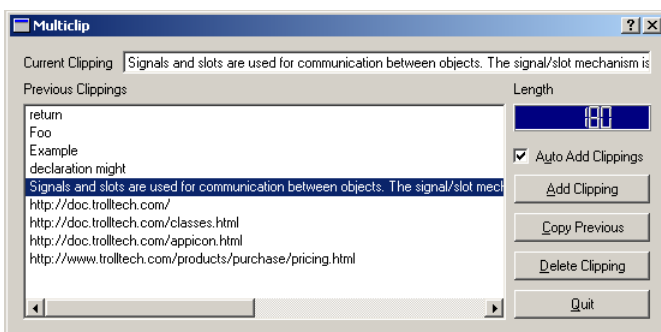
Qt Designer is controlled in the same way as any other modern desktop application. To start *Qt Designer* under Windows click the **Start** button and click **Programs|Qt X.x.x|Designer**. (X.x.x is the Qt version number, e.g. 3.0.0.) If you're running a Unix or Linux operating system you can either double click the *Qt Designer* icon or enter `designer &` in an xterm.

When you've finished using *Qt Designer* click **File|Exit**; you will be prompted to save any unsaved changes. Help is available by pressing **F1** or from the **Help** menu.

To get the most benefit from the tutorial chapters we recommend that you start *Qt Designer* now and create the example applications as you read. Most of the work involves using *Qt Designer's* menus, dialogs and editors, with only small amounts of code to type in.

When you start *Qt Designer*, by default, you will see a menu bar and various toolbars at the top. On the left hand side are three windows, the first is the Files window, the second is the Widgets and Source window (the Object Explorer) and the third is the Properties window. The Files window lists the files and images associated with the project; to open any form single click it in the Files list. The Widgets and Source window lists the current form's widgets and slots. The Properties window is used to view and change the properties of forms and widgets. We will cover the use of *Qt Designer's* windows, dialogs, menu options and toolbar buttons as we create example applications.

In this chapter we will build an application called 'multiclip' which allows you to store and retrieve multiple text clippings to and from the clipboard.



The Multiclip Application

Creating a New Project

Whenever you create a new application we *strongly* recommend that you create a project file and open the project rather than individual `.ui` files. Using a project has the advantage that all the forms you create for the project are available via a single mouse click rather than having to be loaded individually through file open dialogs. An additional benefit of using project files is that they allow you to store all your images in a single file rather than duplicate them in each form in which they appear. See The Designer Approach chapter's Project management

section for detailed information on the benefits of using project files.

Start *Qt Designer* if you haven't already. Click **File|New** to invoke the *New File* dialog. Click the 'C++ Project' icon, then click **OK** to invoke the *Project Settings* dialog. You need to give the project a name, and we recommend that you put each project in its own subdirectory. Click the ellipsis ... button to invoke the *Save As* dialog and navigate to where you want to put the new project. Click the **Create New Folder** toolbar button to create the 'multiclip' directory. Double click the 'multiclip' directory to make it the current directory. Enter a file name of 'multiclip.pro', and click the **Save** button. The 'Project File' field of the *Project Settings* dialog will have the path and name of your new project; click **OK** to create the project.

The name of the current project is shown in the **Files** toolbar which is the top left toolbar by default. Once we have a project we can add forms and begin to build our application. (See *Customizing Qt Designer* for information on changing *Qt Designer's* toolbars and windows to suit your preferences.)

Creating a New Form

Click **File|New** to invoke the *New File* dialog. Several default forms are supplied but we will use the default Dialog form, so just click **OK**. A new form called 'Form1' will appear. Note that the new form is listed in the Files list and the Properties window shows the form's default property settings.

Click the Value beside the *name* property and change the form's name to 'MulticlipForm'. Change the form's caption to 'Multiclip'. The properties are ordered in accordance with the inheritance hierarchy, and caption is roughly in the middle of the property editor. Save the form: click **File|Save**, enter the name 'multiclip.ui', then click the **Save** button.

Using the Property Editor

The Property Editor has two columns, the Property column which lists property names and the Value column which lists the values. Some property names have a plus sign '+' in a square to their left; this signifies that the property name is the collective name for a set of related properties. Click the form to make the Property Editor show the form's properties. Click the *sizePolicy* property's plus sign; you will see four properties appear indented below *sizePolicy*, *hSizeType*, *vSizeType*, *horizontalStretch* and *verticalStretch*. These properties are edited in the same way as any other properties.

Some properties have simple values, for example, the *name* property has a text value, the *width* property (within *minimumSize* for example) has a numeric value. To change a text value click the existing text and type in your new text. To change a numeric value click the value and either type in a new number, or use the spin buttons to increase or decrease the existing number until it reaches the value you want. Some properties have a fixed list of values, for example the *mouseTracking* property is boolean and can take the values True or False. The *cursor* property also has a fixed list of values. If you click the *cursor* property or the *mouseTracking* property the value will be shown in a drop down combobox; click the down arrow to see what values are available. Some properties have complex sets of values; for example the *font* property. If you click the font property an ellipsis button (...) will appear; click this button and a *Select Font* dialog will pop up which you can use to change any of the font settings. Other properties have ellipsis buttons which lead to different dialogs depending on what settings the property can have. For example, if you have a lot of text to enter for a *text* property you could click the ellipsis button to invoke the multi-line text editor dialog.

The names of properties which have changed are shown in bold. If you've changed a property but want to revert it to its default value click the property's value and then click the red 'X' button to the right of the value. Some properties have an *initial* value, e.g. 'TextEdit1', but no default value; if you revert a property that has an initial value but no default value (by clicking the red 'X') the value will become empty unless the property, e.g. name, is not allowed to be empty.

The property editor fully supports Undo and Redo (**Ctrl+Z** and **Ctrl+Y**, also available from the **Edit** menu).

Adding Widgets

The multiclip application consists of a text box to display the current clipboard text (if any), a list box showing the previous clippings, a length indicator, a checkbox and buttons. If you run the application and resize it all the widgets will scale properly.

The *Qt Designer* approach to laying out a form is to place the required widgets on the form in the approximate positions that they should occupy and then use the layout tools to size and position them correctly. We'll now add the multiclip form's widgets.

1. We'll start with the current clipping text box. Click the **Text Label** toolbar button and click towards the top left of the form. (If you hover the mouse over a toolbar button its name will appear in a tooltip.) We won't bother renaming the label since we'll never refer to it in code; but we need to change its text, so change its *text* property to 'Current Clipping'. (See the Using the Property Editor sidebar for an explanation of the property editor.)

Click the **Line Edit** toolbar button and click towards the top right of the form. Use the Property Editor to change the widget's name to 'currentLineEdit'.

2. Now we'll add another label and the list box. Click the **Text Label** toolbar button and click below the Current Clipping label. Change the *text* property to 'Previous Clippings'. Don't worry about positioning the widgets precisely, nor about the fact that they are the wrong size; the layout tools (covered in the next section) will take care of this.

Click the **List Box** toolbar button and click the form below the 'Previous Clippings' label. Change the list box's name to 'clippingsListBox'. By default *Qt Designer* creates list boxes with a single initial value of 'New Item'. We don't want this value (we'll be populating our list box in code later), so we need to remove the value. Right click the list box then click the **Edit** menu item on the popup menu to invoke the listbox's value editor dialog. Click **Delete Item** to delete the default item, then click **OK**. (See the Value Editors sidebar.)

3. We want to know the length of the current clipping so we'll add a label and an *LCD Number* widget.

Click the **Text Label** toolbar button and click below the *Line Edit*. Change its *text* property's value to 'Length'. Click the **LCD Number** toolbar button and click below the length label. Change the LCD Number's name to 'lengthLCDNumber'.

4. Multiclip can be made to detect clipboard changes and automatically add new clippings. We want the user to have control over whether this should happen or not so we will provide a check box which they can use to indicate their preference.

Click the **Check Box** toolbar button and click below the *LCD Number*. Change the checkbox's name to 'autoCheckBox' and its text to 'A&uto Add Clippings'. Note that the *accel* property automatically changes to **Alt+U** because the ampersand in the text signifies a keyboard shortcut.

5. The last widgets we require are the buttons. One way to add the same kind of widget multiple times is to add one, copy it, then paste repeatedly. We will use another approach.

Double click the **Push Button** toolbar button; now click below the checkbox to place a button. Click below the button we've just added to add a second button. Add a third and fourth button. Now click the **Pointer** toolbar button to switch off automatically adding the same widget. Change the first button's name to 'addPushButton' and its text to '&Add Clipping'. Change the second button's name to 'copyPushButton' and its text to '&Copy Previous'. Change the third button's name and text properties to 'deletePushButton' and '&Delete Clipping' respectively. Similarly change the fourth button's name and text to 'quitPushButton' and '&Quit'.

All our widgets have been placed on the form with their properties changed to suit our application's needs. In the next section we will use *Qt Designer's* layout tools to size and position the widgets correctly and in such a way that when the user resizes the form the widgets will scale properly.

Value Editors

Whilst the Property Editor is used to customize the generic properties of widgets, value editors are used to edit values held within instances of particular widgets. For example a **QLineEdit** can only contain a single line of text, but a **QListBox** can contain any number of items each of which may be a line of text, a pixmap, or both. To invoke a widget's value editor double click the widget. (Alternatively right click the widget and a popup menu will appear;

if the first menu item is 'Edit', you can click this to access the widget's value editor dialog.) Different widgets have different value editors. See the Value Editors chapter for more details.

Laying Out Widgets and Previewing

Introduction to Layouts

Layouts work by grouping together widgets and groups of widgets, horizontally, vertically or in a grid. Widgets that are laid out together horizontally or vertically can be grouped either with a Layout or with a Splitter; the only difference is that a user can manipulate a Splitter themselves.

If we want to lay out some widgets side by side we would select them and then click the **Lay Out Horizontally** toolbar button. If we want our widgets to be lined up one above the other we would use the **Lay Out Vertically** toolbar button. Once we've grouped some widgets together we can then lay out the groups in relation to each other, again using vertical, horizontal or grid layouts. Once we have a collection of laid out groups we then click on the form itself and lay out the groups within the form using one of the layout buttons.

Some widgets will grow to fill the available space, vertically or horizontally or both ways. Buttons and line edits will fill horizontal space for example, whereas a *ListView* will fill space in both directions. The easiest way to achieve the layout you want is to use *Qt Designer's* layout tools. When you apply a layout to some widgets in some situations the widgets may not lay out the way you want. If a widget does not fill up enough space try changing its *sizePolicy* to Expanding. If a widget takes up too much space one approach is to change its *sizePolicy*, and another approach is to use a *Spacer* to consume excess space.

Spacers have no visual appearance on the running form and are used purely to insert space between widgets or groups of widgets. Suppose you have a widget that takes up too much space. You could break the layout and resize the widget to make room for a spacer. Then you would insert the spacer and layout the spacer with the widgets and the spacer will consume the excess space. If the spacer doesn't take up the right amount of space you can change its *sizePolicy* for finer control.

The best way to learn about layouts and spacers is to try them out. Experimenting with layouts is easy. If you make any changes that you aren't happy with you can easily undo them by clicking **Edit|Undo** or by pressing **Ctrl+Z**. In the next section we'll lay out our multiclip example step-by-step.

Laying Out Widgets

Layouts provide a means of grouping widgets and groups of widgets together in horizontal and vertical pairs and in grids. If you use layouts your forms and the widgets they contain will scale automatically when the user resizes the window. This is better than using absolute sizes and positions since you don't have to write any code to achieve the scaling and your users can make the most of their screen size whether they have a laptop or a very large screen desktop machine. Layouts use standard sizes for margins and widget spacing which helps give your applications a consistent and proportional look without requiring any effort on your part. Layouts are also easier and faster to use than absolute positioning; you can just place your widgets on the form in approximate positions and leave the layout tools to size and scale the widgets correctly.

Selecting Widgets and Inserting Widgets

To select an individual widget, either click the widget itself or click its Name in the Object Explorer window. To select a group either click a fraction outside its red outline or click its Name in the Object Explorer window. To select multiple widgets or groups, click the form to deselect any selected widgets, then **Ctrl+Click** one widget or group then drag the rubber band so that it touches the other widgets or groups that you want to select. This technique is particularly useful for selecting widgets that are *inside* another widget. For example to select the radio buttons in a button group but not the button group itself you would click the form then **Ctrl+Click** one of the radio buttons and drag the rubber band to touch the other radio buttons.

If we want to insert a widget into a gap between widgets which are in a layout we can click the toolbar button for the new widget and then click in the gap. *Qt Designer* will ask us if we want to break the layout and if we click

Break Layout the layout will be broken and our widget inserted. We can then select the widgets and groups we want to lay out and lay them out again. The same effect can be achieved by clicking the group and either clicking the **Break Layout** toolbar button or pressing **Ctrl+B**.

The layout we want to achieve is to have the current clipping label and currentLineEdit side by side at the top of the form. We want the previous clippings label and the clippingsListBox to occupy the left hand side of the form with the remaining widgets in a column on the right. We want to divide left and right with a splitter and make the left hand side larger by default. We'll leave the sizing of the widgets to *Qt Designer*. The layout controls are in the **Layout** toolbar. (By default this is the fourth toolbar counting left to right.) We'll now lay out the widgets we've placed on the form.

1. Click the current clipping label and **Shift+Click** the currentLineEdit *Line Edit*. (**Shift+Click** means hold down the shift key whilst clicking; this will ensure that *Qt Designer* performs multiple selections.) Most of the layout toolbar buttons will now be available. Click the **Lay Out Horizontally** toolbar button. (If you hover the mouse over a toolbar button a tooltip giving the button's name will appear.) The two widgets will be moved together with a thin red line surrounding them. It doesn't matter that the widgets aren't the right size or in exactly the right place; as we progress with the layout *Qt Designer* will size and place them correctly.
2. Click the Previous Clippings label and **Shift+Click** the clippingsListBox. Click the **Lay Out Vertically** toolbar button.
3. We want the remaining widgets to be grouped together vertically. We could **Shift+Click** each one but instead click the form above the Length label, then drag until the Length label, the LCD Number, the check box and all the buttons are all touching the rubber band (a black outline rectangle) that appears when you drag. Release the mouse, and all the remaining widgets should be selected. If you missed any **Shift+Click** them. Now click the **Lay Out Vertically** toolbar button.

We now have three groups of widgets which must be laid out in relation to each other and then laid out in relation to the form itself.

1. **Shift+Clicking** is used to select individual widgets. To select a group we must click the form to deselect any selected widgets, then **Ctrl+Click** the group and drag so that the rubber band touches the groups we want to lay out and then release. With the buttons and other widgets already laid out and selected, **Ctrl+Click** the list box and drag the rubber band over the one of the buttons, then release. Both groups should now be selected. Click the **Lay Out Horizontally (in Splitter)** toolbar button.
2. We now have two groups, the top one with the Current Clipping label and the line edit and the group we've just created with the list box, buttons and other widgets. We now want to lay these out in relation to the form. Click the form and click the **Lay Out Vertically** toolbar button. The widgets will be resized to fill the entire form.

Unfortunately the Length label and the LCD Number take up far too much space, so we will have to revise the layout. With experience you will find that you do not need to rework layouts very often. We will insert a spacer which will use the extra space.

1. First we must make some room for the spacer. Click the LCD Number to select it. Now click the **Break Layout** toolbar button. Move the LCD Number up a little bit, there's no need to be exact we just want to create some space below it.
2. Now we'll add the spacer. Click the **Spacer** toolbar button, then click the form in the space you've created between the LCD Number and the check box. A popup menu with two options, Horizontal and Vertical, will appear; click Vertical. We choose vertical because we want the spacer to consume excess vertical space.
3. We need to regroup the buttons and other widgets in a vertical group. Drag the mouse from near the bottom right of the form so that the rubber band includes or touches the buttons, the check box, the spacer, the LCD Number and the Length label; then release. If you selected any other widgets by mistake, click the form and try the drag again. Click the **Lay Out Vertically** toolbar button.
4. We now have three groups as we had before, only this time with the small addition of the spacer. Select the list box and the buttons by clicking the form, dragging and releasing once the rubber band covers or touches both groups. Click **Lay Out Horizontally (in Splitter)** to regroup them with the splitter.

5. The last step is to lay out the form itself. Click the form and click **Lay Out Vertically**. The form should now be laid out correctly.

There are two small deficiencies in the layout that we have achieved. Firstly the list box and buttons take up an equal width whereas we'd rather have the list box take up about three quarters of the width. Secondly the Length label, the check box and the buttons extend right up to the splitter. They would look more attractive if there was a little bit of space separating them from the splitter.

Expanding the list box half of the splitter would require us to add an `init()` function with the following code:

```
void MulticlipForm::init()
{
    QValueList sizes;
    sizes << 250 <setSizes( sizes );
}
```

We won't add this code now since we'll deal with the code when we come to implement the application's functionality later in the chapter.

We will create some space around the splitter by changing the margins of the layout groups that it joins together. To click a layout either click a fraction above the layout's top red line or click the layout's name in the Object Explorer (the Widgets and Source window). (See Object Explorer Window sidebar for an explanation of the Object Explorer window.) Click the layout that contains the list box, and change the `layoutMargin` property to 6, then press **Enter**. Click the layout that contains the buttons and other widgets, and change its `layoutMargin` to the same value in the same way.

The Object Explorer

View the Object Explorer (Widgets and Source) window by clicking **Window | Views | Object Explorer**. The Object Explorer has two tabs, the Widgets tab which shows the object hierarchy, and the Source tab which shows the source code you have added to the form. Clicking the name of a widget in the Widget tab will select the widget and show its properties in the Property Editor. It is easy to see and select widgets in the Object Explorer which is especially useful for forms that have many widgets or which use layouts.

In the original version of *Qt Designer* if you wanted to provide code for a form you had to subclass the form and put your code in the subclass. This version fully supports the subclassing approach, but now provides an alternative: placing your code directly into forms. Writing code in *Qt Designer* is not quite the same as subclassing, for example you cannot get direct access to the form's constructor or destructor. If you need code to be executed by the constructor create a slot called `void init()`; if it exists it will be called from the constructor. Similarly, if you need to be executed before destruction create a slot called `void destroy()`. You can also add your own class variables which will be put in the generated constructor's code, and you can add forward declarations and any includes you require. To add a variable or declaration right click the appropriate item, e.g. Class Variables, then click New then enter your text, e.g. `QString fileName`. If one or more items exist right clicking will popup a two item menu with New and Delete as options. To edit code just click the name of a function to invoke the code editor. Code editing and creating slots are covered later.

If you subclass the form you create your own `.cpp` files which can contain your own constructor, destructor, functions, slots, declarations and variables as your requirements dictate. (See Subclassing for more information.)

In the example we used *Qt Designer's* layout tools to lay out our widgets. We will use the layout tools again in the examples presented in later chapters. If you want to use absolute positioning, i.e. to place and size your widgets with exact pixel sizes you can easily do so. To place a widget click it and drag it to the desired position. To resize it, click it, and drag one of the sizing handles (these are small blue squares) until the size is right. To stop the widget from resizing when the window is resized change the `hSizePolicy` and `vSizePolicy` (these are properties within the `sizePolicy` property), to Fixed.

Previewing

Although *Qt Designer* presents an accurate view of our forms we often want to see what a form looks like when it is run. It is also useful to be able to test out some aspects of the form, for example how the form scales when resized

or how the splitters work in practice. If we're building cross-platform applications it is also useful to see how the form will look in different environments.

To see a preview either click **Preview|Preview Form** or press **Ctrl+T**. To leave preview mode close the window in the standard way for your environment. To view previews which show how the application will look on other platforms click the **Preview** menu and click one of the menu items that drop down.

Preview the multiclip form and try out the splitter and try resizing the form. In all probability you moved the splitter to the right to reduce the size of the buttons to make the form more attractive. The splitter seemed like a good idea but in practice we want the buttons and the other widgets on the right hand side to take up a fixed amount of space. *Qt Designer* makes changing layouts very easy, so we'll fix this straight away.

Click the splitter then click the **Break Layout** toolbar button; the splitter will be removed. Now click the form itself, near the bottom, and drag the rubber band so that it touches both the list box and some of the buttons, then release. The list box group and the buttons group are selected; click the **Lay Out Horizontally** toolbar button. Click the form then click the **Lay Out Vertically** toolbar button. The form is now laid out as we require. Preview the form (press **Ctrl+T**) and try resizing it.

It would be useful if you experimented further with layouts since they work visually and are best learnt through practice. To remove a layout click the **Break Layout** toolbar button; to apply a layout select the relevant widgets or groups and click a layout button. You can preview as often as you like and you can always undo any changes that you make.

Let's try an experiment, to see how the grid layout works. Click the list box, then press **Ctrl+B** (break layout). Click one of the buttons and press **Ctrl+B**. Click the form at the bottom and drag until all the widgets are touching or within the rubber band, (but excluding the Current Clipping label and the currentLineEdit line edit); then release. Press **Ctrl+G** (lay out in a grid). Click the form, then click **Ctrl+L** (lay out vertically). Our original design is back — but this time using a grid layout.

Changing the Tab Order

Keyboard users press the **Tab** key to move the focus from widget to widget as they use a form. The order in which the focus moves is called the tab order. Preview multiclip (press **Ctrl+T**) and try tabbing through the widgets. The tab order may not be what we want so we'll go into tab order mode and change it to the order we want.

When you click the **Tab Order** toolbar button a number in a blue circle will appear next to every widget that can accept keyboard focus. The numbers represent each widget's tab order, starting from 1. You change the tab order by clicking the widgets in the order you want to be the new tab order. If you make a mistake and need to start again, double click the widget you want to be first, then click the other widgets in the required order as before. When you've finished press **Esc** to leave tab order mode. If you made a mistake or preferred the previous tab order you can undo your changes by leaving tab order and undoing (press **Esc** then **Ctrl+Z**).

Click the **Tab Order** toolbar button, then click the current clipping *Line Edit* — even if it is already number one in the tab order. Next click the previous clipping *Listbox*, then the auto add clippings *CheckBox*. Click each button in turn from top (add clipping) to bottom (quit). Press **Esc** to finish tab order mode, then preview the form and try tabbing through the widgets.

Note that you can stop clicking if the tab order numbers for all the widgets is correct; just press **Esc** to leave tab order mode.

Connecting Signals and Slots

Qt provides the signals and slots mechanism for communicating between widgets. Signals are emitted by widgets when particular events occur. We can connect signals to slots, either pre-defined slots or those we create ourselves. In older toolkits this communication would be achieved using callbacks. (For a full explanation of Qt's signals and slots mechanism see the on-line Signals and Slots documentation.)

Connecting Predefined Signals and Slots

Some of an application's functionality can be obtained simply by connecting pre-defined signals and slots. In *Multiclip* there is only one pre-defined connection that we can use, but in the *richedit* application that we'll build in *Creating Main Windows with Actions, Toolbars and Menus* we will use many pre-defined signals and slots to get a lot of the functionality we need without having to write any code.

We will connect the `Quit` button's `clicked()` signal to the form's `accept()` slot. The `accept()` slot notifies the dialog's caller that the dialog is no longer required; since our dialog is our main window this will close the application. Preview the form (press **Ctrl+T**); click the **Quit** button. The button works visually but does nothing. Press **Esc** or close the preview window to leave the preview.

Click the **Connect Signals/Slots** toolbar button. Click the `Quit` button, drag to the form and release. The *Edit Connections* dialog will pop up. The top left hand list box lists the Signals that the widget we've clicked can emit. At the top right is a combobox which lists the form and its widgets; any of these are candidates for receiving signals. Since we released on the form rather than a widget the slots combobox shows the form's name, 'MulticlipForm'. Beneath the combobox is a list box which shows the slots available in the form or widget shown in the combobox. Note that only those slots that can be connected to the highlighted signal are shown. If you clicked a different signal, for example the `toggled()` signal, the list of available slots would change. Click the `clicked()` signal, then click the `accept()` slot. The connection will be shown in the Connections list box. Click **OK**.

We will make lots of signal/slot connections as we work through the examples, including connections to our own custom slots. Signal/slot connections (using pre-defined signals and slots) work in preview mode. Press **Ctrl+T** to preview the form; click the form's **Quit** button. The button now works correctly.

Creating and Connecting Custom Slots

In the first version of *Qt Designer* you could create the signatures of your custom slots and make the connections, but you could not implement your slots directly. Instead you had to subclass the form and code your slots in the subclass. The subclassing approach is still available, and makes sense in some situations. But now you can implement your slots directly in *Qt Designer*, so for many dialogs and windows subclassing is no longer necessary. (*Qt Designer* stores the slot implementations in a `.ui.h` file; see *The ui.h extension approach in The Designer Approach* chapter for more about these files.)

The *Multiclip* application requires four slots, one for each button, but only three need to be custom slots since we connected a signal to a pre-defined slot to make the `Quit` button functional. We need a slot for the `Add Clipping` button; this will add the current clipping to the list box. The `Copy Previous` button requires a slot which will copy the selected list box item to the current clipping line edit (and to the clipboard). The `Delete Clipping` button needs a slot to delete the current clipping and the current list box item. We will also need to write some initialization code so that when the application starts it will put the current clipboard text (if any) into the line edit. The code is written directly in *Qt Designer*; the snippets are taken from the generated `qt/tools/designer/examples/multiclip/multiclip.ui.h` file.

We'll need Qt's global clipboard object throughout the code which would mean calling `QApplication::clipboard()` or `qApp->clipboard()` in several places. Rather than perform all these function calls we'll keep a pointer to the clipboard in the form itself. Click the `Source` tab of the `Object Explorer`. (If the `Object Explorer` isn't visible click **Window|Views|Object Explorer**.) The `Source` tab shows us the functions in our form, the class variables, the forward declarations and the names of the include files we've asked for.

Right click the `Class Variables` item, then click **New** on the popup menu. (If there had been any existing variables the popup menu would also have a `Delete` option.) Type in `'QClipboard *cb;'` and press **Enter**. We will create an `init()` function in which we will assign this pointer to Qt's global clipboard object. We also need to declare the clipboard header file. Right click `Includes (in Declaration)`, then click **New**. Type in `'<qclipboard.h>'` and press **Enter**. Since we need to refer to the global application object, `qApp`, we need to add another include declaration. Right click `Includes (in Implementation)`, then click **New**. Type in `'<qapplication.h>'` and press **Enter**. The variable and declarations will be included in the code generated from *Qt Designer's* `.ui` file.

We will invoke *Qt Designer's* code editor and write the code.

We'll create the `init()` function first. One way of invoking the code editor is to click the `Source` tab, then click the

name of the function you want to work on. If you have no functions or wish to create a new function you need to use the Source tab. Right click the 'protected' item in the Source tab's Functions list, then left click **New** to launch the *Edit Slots* dialog. Change the slot's name from 'newSlot' to 'init()' then click **OK**. You can then click inside the editor window that appears to enter your code.

Note that you are not forced to use *Qt Designer's* code editor; so long as you add, delete and rename your slots all within *Qt Designer*, you can edit the implementation code for your slots using a separate external editor and *Qt Designer* will preserve the code you write.

```
void MulticlipForm::init()
{
    lengthLCDNumber->setBackgroundColor( darkBlue );
    currentLineEdit->setFocus();

    cb = QApplication->clipboard();
    connect( cb, SIGNAL( dataChanged() ), SLOT( dataChanged() ) );
    if ( cb->supportsSelection() )
        connect( cb, SIGNAL( selectionChanged() ), SLOT( selectionChanged() ) );

    dataChanged();
}
```

The first couple of lines change the LCD number's background color and make the form start with the focus in the line edit. We take a pointer to Qt's global clipboard and keep it in our class variable, `cb`. We connect the clipboard's `dataChanged()` signal to a slot called `dataChanged()`; we will create this slot ourselves shortly. If the clipboard supports selection (under the X Window system for example), we also connect the clipboard's `selectionChanged()` signal to a slot of the same name that we will create. Finally we call our `dataChanged()` slot to populate the line edit with the clipboard's text (if any) when the application begins.

Since we've referred to the `dataChanged()` and `selectionChanged()` slots we'll code them next, starting with `dataChanged()`.

```
void MulticlipForm::dataChanged()
{
    QString text;
    text = cb->text();
    clippingChanged( text );
    if ( autoCheckBox->isChecked() )
        addClipping();
}
```

We take a copy of the clipboard's text and call our own `clippingChanged()` slot with the text we've retrieved. If the user has checked the Auto Add Clippings checkbox we call our `addClipping()` slot to add the clipping to the list box.

The `selectionChanged()` slot is only applicable under the X Window System. Users of MS Windows can still include the code to ensure that the application works cross-platform.

```
void MulticlipForm::selectionChanged()
{
    cb->setSelectionMode( TRUE );
    dataChanged();
    cb->setSelectionMode( FALSE );
}
```

We tell the clipboard to use selection mode, we call our `dataChanged()` slot to retrieve any selected text, then set the clipboard back to its default mode.

In the `dataChanged()` slot we called another custom slot, `clippingChanged()`.

```
void MulticlipForm::clippingChanged( const QString & clipping )
{
    currentLineEdit->setText( clipping );
    lengthLCDNumber->display( (int)clipping.length() );
}
```

We set the line edit to whatever text is passed to the `clippingChanged()` slot and update the LCD number with the length of the new text.

The next slot we'll code will perform the Add Clipping function. This slot is called by our code internally (see the `dataChanged()` slot above), and when the user clicks the Add Clipping button. Since we want *Qt Designer* to be able to set up a connection to this slot instead of just typing it in the editor window we'll let *Qt Designer* create its skeleton for us. Click **Edit|Slots** to invoke the *Edit Slots* dialog. Click **New Slot** and replace the default name of 'new_slot()' with 'addClipping()'. There is no need to change the access specifier or return type. Now that we've created our slot we can implement it in the code editor where it has now appeared.

The Add Clipping button is used to copy the clipping from the Current Clipping line edit into the list box. We also update the length number.

```
void MulticlipForm::addClipping()
{
    QString text = currentLineEdit->text();
    if ( ! text.isEmpty() ) {
        lengthLCDNumber->display( (int)text.length() );
        int i = 0;
        for ( ; i < count(); i++ ) {
            if ( clippingsListBox->text( i ) == text ) {
                i = -1; // Do not add duplicates
                break;
            }
        }
        if ( i != -1 )
            clippingsListBox->insertItem( text, 0 );
    }
}
```

If there is some text we change the LCD's value to the length of the text. We then iterate over all the items in the list box to see if we have the same text already. If the text is not already in the list box we insert it.

To make the Add Clipping button functional we need to connect the button's `clicked()` signal to our `addClipping()` slot. Click the **Connect Signals/Slots** toolbar button. Click the Add Clipping button, drag to the form and release. (Make sure you drag to the form rather than another widget — the form will have a thin pink border during the drag. If you make a mistake simply change the name in the Slots combobox.) The *Edit Connections* dialog will appear. Click the `clicked()` signal and our `addClipping()` slot. Click **OK** to confirm the connection.

The Copy Previous button is used to copy the selected clipping from the list box into the line edit. The clipping is also placed on the clipboard. The procedure is the same as for the Add Clipping button: first we create the slot, then we implement it and finally we connect to it:

1. Create the slot.

Click the **Edit|Slots** menu item to invoke the *Edit Slots* dialog. Click **New Slot** and replace the default 'new_slot()' name with 'copyPrevious()'. Click **OK**.

2. Implement the slot.

```
void MulticlipForm::copyPrevious()
{
    if ( clippingsListBox->currentItem() != -1 ) {
        cb->setText( clippingsListBox->currentText() );
    }
}
```

```

        if ( cb->supportsSelection() ) {
            cb->setSelectionMode( TRUE );
            cb->setText( clippingsListBox->currentText() );
            cb->setSelectionMode( FALSE );
        }
    }
}

```

The code for Copy Previous checks to see if there is a selected item in the list box. If there is the item is copied to the line edit. If we are using a system that supports selection we have to repeat the copy, this time with selection mode set. We don't explicitly update the clipboard. When the line edit's text is changed it emits a `dataChanged()` signal which our `dataChanged()` slot receives. Our `dataChanged()` slot updates the clipboard.

3. Connect to the slot.

Click the **Connect Signals/Slots** toolbar button. Click the Copy Previous button, drag to the form and release. The *Edit Connections* dialog will pop up. Click the `clicked()` signal and the `copyPrevious()` slot. Click **OK**.

We take the same approach to the Delete Clipping button.

1. Click **Edit|Slots** to invoke the *Edit Slots* dialog. Click **New Slot** and replace the default name with 'delete-Clipping()'. Click **OK**.
2. The Delete button must delete the current item in the list box and clear the line edit.

```

void MulticlipForm::deleteClipping()
{
    clippingChanged( "" );
    clippingsListBox->removeItem( clippingsListBox->currentItem() );
}

```

We call our own `clippingChanged()` slot with an empty string and use the list box's `removeItem()` function to remove the current item.

3. Connect the Delete Clipping button's `clicked()` signal to our `deleteClipping()` slot. (Press **F3** — which is the same as clicking the **Connect Signals/Slots** toolbar button. Click the Delete Clipping button and drag to the form; release. The *Edit Connections* dialog will appear. Click the `clicked()` signal and the `deleteClipping()` slot. Click **OK**.)

Compiling and Building an Application

So far we have written about 99% of a Qt application entirely in *Qt Designer*. To make the application compile and run we must create a `main.cpp` file from which we can call our form.

The simplest way to create a new source file is by clicking **File|New** to invoke the 'New File' dialog, then click 'C++ Source' or 'C++ Header' as appropriate, then click **OK**. A new empty source window will appear. Click **File|Save** to invoke the *Save As* dialog, enter 'main.cpp', then click **Save**.

Enter the following code in the `main.cpp` C++ editor window:

```

#include <qapplication.h>
#include "multiclip.h"

int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

    MulticlipForm clippingForm;
    app.setMainWidget( &clippingForm );
}

```



```
        clippingForm.show();  
  
        return app.exec();  
    }
```

The program creates a **QApplication** object and an instance of our `MulticlipForm`, sets the form to be the main widget and shows the form. The `app.exec()` call starts off the event loop.

Now start up a console (or `xterm`), change directory to the `multiclip` application and run `qmake`. A Makefile compatible with your system will be generated:

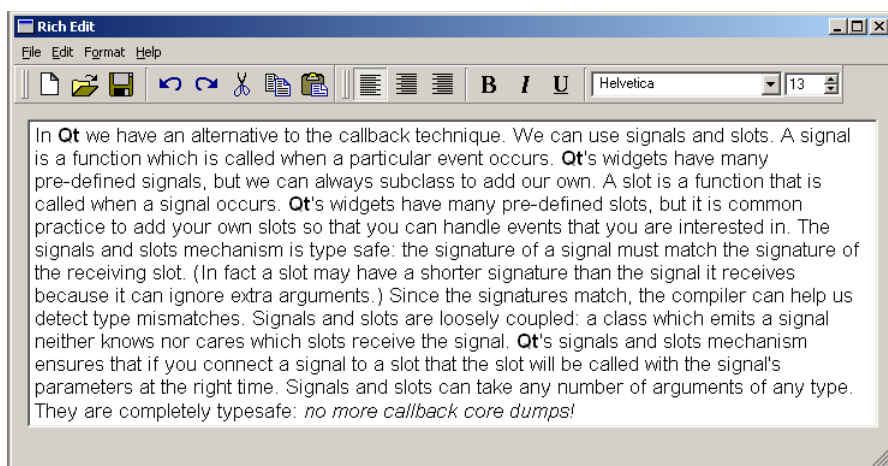
```
qmake -o Makefile multiclip.pro
```

You can now make the application, e.g. by running `make` or `nmake`. Try compiling and running `multiclip`. There are many improvements you could make and experimenting with both the layout and the code will help you learn more about Qt and *Qt Designer*.

This chapter has introduced you to creating cross-platform applications with *Qt Designer*. We've created a form, populated it with widgets and laid the widgets out neatly and scalably. We've used Qt's signals and slots mechanism to make the application functional and generated the Makefile. These techniques for adding widgets to a form and laying them out with the layout tools; and for creating, coding and connecting slots will be used time and again as you create applications with *Qt Designer*. The following chapters will present further examples and explore more techniques for using *Qt Designer*.

Creating Main Windows with Actions, Toolbars and Menus

In this chapter we will explain how to create an application's main window and how to add actions (explained shortly), menus and toolbars. We will also demonstrate how some common actions, like cut and paste in a **QTextEdit**, can be performed automatically simply by making the appropriate signals and slots connections. We will build the *richedit* application to illustrate the necessary techniques.



The Richedit Application

We begin by creating a project file. Start *Qt Designer* or if it is already running, close any existing projects and files. Click **File|New** to invoke the *New File* dialog. Click the 'C++ Project' icon, then click **OK** to invoke the *Project Settings* dialog. Click the ellipsis button to invoke the *Save As* dialog and navigate to where you want to put the new project. Use the **Create New Folder** toolbar button to create the 'richedit' directory if it doesn't exist. Make sure you're in the 'richedit' directory (double click it) and enter a file name of 'richedit.pro'. Click the **Save** button. The 'Project File' field of the *Project Settings* dialog will have the path and name of your new project; click **OK** to create the project.

If you're unfamiliar with Actions and Action Groups the sidebar provides the necessary introduction; otherwise skip ahead to "Designing the Main Window".

Actions and Action Groups

An *action* is an operation that the user initiates through the user interface, for example, saving a file or changing some text's font weight to bold.

We often want the user to be able to perform an action using a variety of means. For example, to save a file we might want the user to be able to press **Ctrl+S**, or to click the **Save** toolbar button or to click the **File|Save** menu option. Although the means of invoking the action are all different, the underlying operation is the same and we don't want to duplicate the code that performs the operation. In Qt we can create an action (a **QAction** object) which will call the appropriate function when the action is invoked. We can assign an accelerator, (e.g. **Ctrl+S**), to an action. We can also add an action to a menu item and to a toolbar button.

If the action has an on/off state, e.g. bold is on or off, when the user changes the state, for example by clicking

a toolbar button, the state of everything associated with the action, e.g. menu items and toolbar buttons, are updated.

Some actions should operate together like radio buttons. For example, if we have left align, center align and right align actions, only one should be 'on' at any one time. An *action group* (a **QActionGroup** object) is used to group a set of actions together. If the action group's `exclusive` property is `TRUE` then only one of the actions in the group can be on at any one time. If the user changes the state of an action in an action group where `exclusive` is `TRUE`, everything associated with the actions in the action group, e.g. menu items and toolbar buttons, are updated.

Qt Designer can create actions and action groups visually, can assign accelerators to them, and can associate them with menu items and toolbar buttons.

Designing the Main Window

We will use the main window wizard to build a main window. The wizard allows us to create actions and a menu bar and a toolbar through which the user can invoke the actions. We will also create our own actions, menus and toolbar. We will add some widgets to the toolbar and add a main widget to the main window. Finally we will connect signals to slots to take advantage of Qt's default functionality and minimize our coding.

Creating the Main Window

Click **File|New** to invoke the *New File* dialog, click *Mainwindow*, then click **OK**. A new **QMainWindow** form will be created and the *Mainwindow Wizard* will pop up.

1. The *Choose available menus and toolbars* wizard page appears first. It presents three categories of default actions, File Actions, Edit Actions and Help Actions. For each category you can choose to have *Qt Designer* create menu items, toolbar buttons and signal/slots connections for the relevant actions. You can always add or delete actions, menu items, toolbar buttons and connections later.

We will accept the defaults for File Actions, i.e. have menu items, toolbar buttons and the relevant connections created. But for the Edit Actions we don't want any connections created since we'll be connecting the actions directly to the **QTextEdit** we'll create later, so uncheck the Edit Action's *Create Slots and Connections* checkbox. We won't have any Help Actions on the toolbar so uncheck the Help Action's *Toolbar* checkbox. Click **Next** to move on to the next wizard page.

2. The *Setup Toolbar* wizard page is used to populate a toolbar with actions from each of the default action categories. The *Category* combobox is used to select which set of actions you wish to work on. The *Actions* list box lists the actions available for the current category. The *Toolbar* listbox lists the toolbar buttons you want to create. The blue left and right arrow buttons are used to move actions into or out of the *Toolbar* list box. The blue up and down arrow buttons are used to move actions up and down within the *Toolbar* list box. Note that the '<Separator>' item in the *Actions* list box may be moved to the *Toolbar* list box as often as required and will cause a separator to appear in the finished toolbar.

Copy the *New*, *Open* and *Save* Actions to the *Toolbar* list box. Copy a *<Separator>* to the *Toolbar* list box. Change the *Category* to *Edit* and copy the *Undo*, *Redo*, *Cut* *Copy* and *Paste* actions to the *Toolbar* list box. Click **Next** and then click **Finish**.

If you preview the form (**Ctrl+T**) the *File* and *Edit* menus will be available and you'll be able to drag the toolbar either into an independent window of its own, or to dock it to the left, right, bottom or top of the window. The menus and toolbars are not functional yet but we will rectify this as we progress.

Creating and Deleting Actions and Action Groups

Creating Actions

Our application requires more menu items and toolbar buttons than the the defaults we created with the main window wizard. But before we create the menu items and toolbar buttons we will create the actions that they'll

invoke. The Action Editor appears automatically when you create a main window. You can also access it through the Window menu (click **Window|Action Editor**).

For the richedit application we need to create actions for indicating bold, italic and underlined font attributes, and to set text alignment.

Right click in the *Action Editor* where the actions are listed, then left click New Action. This will create a new action called 'Action' at the top of the list of actions. The Property Editor will change to show the default settings for the new action. We'll now go through all the properties we need to change.

1. Change the *name* of the action to 'boldAction'.
2. Since bold can only be on or off change the *toggleAction* property to True.
3. The *iconSet* property is used to set an icon which will appear to the left of the action's name in any menu you associate the action with. The *iconSet* is also used for the toolbar button if you associate the action with a toolbar. Click the ellipsis button (...) to invoke the *Choose an Image* dialog. The ellipsis button appears when you click in the Value part of the Properties list by a *pixmap* or *iconSet* property. The pixmap we require is not in the default collection. Click the **Add** button and a file dialog will appear. The icons we require are in the Qt examples directory, `qt/examples/textedit/`. Navigate to the `textedit` directory and select the `textbold.xpm` file. Click the new `textbold` icon in the pixmap collection then click **OK**.
4. Change the *text* property to 'bold'. This automatically changes the *menuText*, *toolTip* and *statusTip* properties.
5. Change the menu text to '&Bold'. If we choose to associate this action with a menu item then this property is used; otherwise it is ignored.
6. Change the *accel* property to 'CTRL+B'. This will associate the **Ctrl+B** keyboard accelerator with this action.

Note that at this point the only way to invoke this action is to use the keyboard accelerator (**Ctrl+B**), because we have not yet associated the action with a menu item or with a toolbar button.

We need to add two more actions, italic and underline. For each one right click the *Action Editor* and click New Action. Then repeat the steps listed above to change each action's properties:

- For the italic action change its *name* to 'italicAction' and ensure its *toggleAction* property is True. The *iconSet* is in the `textedit` directory and is called `textitalic.xpm`; add its pixmap in the same way that we added the bold pixmap. (For example, click the ellipsis (...), click **Add**, navigate to the `textedit` directory and click the `textitalic.xpm` pixmap. Then click the `textitalic` pixmap in the pixmap collection and click **OK**). Change the action's *text* to 'italic', its *menuText* to '&Italic' and its *accel* to 'CTRL+I'.
- For the underline action change its *name* to 'underlineAction' and set its *toggleAction* property to True. The *iconSet* is in the same directory and is called `textunder.xpm`; add it in the same way as the previous pixmaps. Change its *text* to 'underline', its *menuText* to '&Underline' and its *accel* to 'CTRL+U'.

Creating Action Groups

It is perfectly possible to have bold, italic and underline all active at once. But for alignment, i.e. left align, right align and centered, it only makes sense for one of them to be active at any one time. Because we need the alignment actions to operate in sync with one another we must create an Action Group which will automatically manage the behaviour of the actions it contains in the way we require.

Right click the *Action Editor* then left click New Action Group. Change the action group's *name* in the Property Editor to 'alignActionGroup', and change its *text* to 'align'. The '*exclusive*' property's default is True. This ensures that only one action within the action group can be 'on' at any one time which is the behaviour we require.

We create the actions within the action group in almost the same way as before. The only difference is that we must right click the `alignActionGroup` (rather than an Action) and then left click New Action to create a new action *within* the action group. We will create three new actions within the `alignActionGroup`:

1. Create a new action within the `alignActionGroup` action group. Change the *name* of this action to 'left-AlignAction' and ensure its *toggleAction* property is True. Use the *iconSet* in the `textedit` directory called

`textleft.xpm`, adding the pixmap to the pixmap collection in the same way as we added the `textbold` pixmap earlier. Change its `text` to 'left', its `menuText` to '&Left' and its `accel` to 'CTRL+L'.

2. Create another new action within `alignActionGroup`. Change this action's `name` to 'rightAlignAction' and set its `toggleAction` property to `True`. Set its `iconSet` to `textright.xpm` using the pixmap collection as before. Change its `text` to 'right', its `menuText` to '&Right' and its `accel` to 'CTRL+R'.
3. Create a third action within `alignActionGroup`. Change its `name` to 'centerAlignAction' and make its `toggleAction` property `True`. Change its `iconSet` to `textcenter.xpm`. Change its `text` to 'center' and its `menuText` to '&Center'. We won't set an accelerator for this action.

Note that the `toolTip` and `statusTip` properties were inherited from the action group; you may wish to change these to be specific to the actions concerned.

Deleting Actions and Action Groups

We have some actions that we don't intend providing for this first release, for example, the `editFindAction` and the `filePrintAction`. Click `editFindAction` and then click the **Delete Action** toolbar button. Delete the `filePrintAction` in the same way. Action Groups (including any actions they contain) can also be deleted with the **Delete Action** toolbar button.

Creating and Populating a Toolbar

All the actions we require are now in place and we are ready to create a new toolbar and add some of our actions to it. Right click on the right hand side of the toolbar area, or on the form, then left click **Add Toolbar**. The new toolbar is empty and is visible only by its `toolbar handle`. (Toolbar handles are usually represented as a gray area containing either two thick vertical lines or with many small pits).

We'll add the new actions we've just created. Drag the `alignActionGroup`, (not any action it contains), to the new toolbar and drop it on the toolbar handle to the right of the vertical lines. The three alignment actions will be added to the toolbar. The bold, italic and underline actions do not belong to an action group, and must be dragged to the toolbar individually. Drag the bold action to the toolbar: when the mouse is over the toolbar a thick red line will appear indicating the position at which the toolbar button will be inserted; drop the bold action on the toolbar. Drag the italic and underline actions to the toolbar and drop them next to the bold button. Separate the alignment toolbar buttons from the font related buttons by right clicking the centered toolbar button and clicking **Insert Separator**.

Toolbar buttons and separators (usually represented as indented vertical gray lines), can be dragged and dropped into new positions in the toolbar at any time. Separators can be inserted by right clicking a toolbar button and clicking **Insert Separator**. Toolbar buttons and separators can be deleted by right clicking them and clicking **Delete Item**. Toolbars can be deleted by right clicking their toolbar handle and clicking **Delete Toolbar**.

If you preview the application you'll find that both the original and new toolbar can be dragged to different docking points or dragged out of the application as independent tool windows.

Adding Widgets to the Toolbar

We want our users to be able to choose the font and font size from the toolbar. To achieve this we'll create a font combobox and a font size spinbox and place them in the formatting toolbar we created in the previous section.

Click *Qt Designer's* **ComboBox** toolbar button and click the last (right-most) toolbar button in the application's new toolbar. Change the combobox's name to 'fontComboBox'. Click the **SpinBox** toolbar button and click the combobox we've just added to insert the spinbox next to it. Change the spinbox's `minValue` property to 6, its `value` property to 10 and its `suffix` to a space followed by 'pt'. Insert a separator to the left of the combobox.

Although you can put any widget into a toolbar we recommend that widgets which can be associated with an action should *not* be added to the toolbar directly. For these widgets, i.e. menu items, toolbar buttons and lists of items, you should create an action (drop down action for a list of items), associate the action with the widget, and add

the action to the toolbar. Widgets that can sensibly be inserted directly into a toolbar are *ComboBoxes*, *SpinBoxes* and *Line Edits*.

Creating Menus

We'll now add the actions we added to the new toolbar to a new menu and modify the existing menus slightly.

Right click our application's menu bar and click Add Menu Item. A new menu item called 'Menu' will appear. Right click this menu item and click Rename Item. Change its name to 'F&ormat'. Click the menu item and a red bar will appear beneath it — this is the empty menu. Drag the `alignActionGroup` to the Format menu item and drop the action group on the menu. (The menu's red bar will appear and a thick red line will be drawn where the new menu items will appear — drop when the red line is in the position you require.) Now if you click the Format menu item the three alignment actions will be displayed. Just like the toolbar we must add the bold, italic and underline actions individually. Drag the bold action to the Format menu and move the mouse so that the red line is positioned at the bottom of the menu, then drop the action. Repeat this process for the italic and underline actions.

We'll now deal with the separators in the menus. Firstly we'll add a separator in the Format menu and then we'll remove some redundant separators from the other menus. Click the Format menu and right click the bold item; click Insert Separator. Click the File menu and right click one of the separators above the Exit action; click Delete Item. Click the Edit menu, right click the separator at the very bottom of the menu and click Delete Item. Delete Item can be used to delete separators, menu items and menus.

Click the Format menu and drag it to the left of the Help menu, then drop the Format menu. (A thick red bar will appear to indicate the insertion position.) Both menus and menu items may be dragged and dropped to different positions in the same way.

Preview the application and try clicking the alignment and font style toolbar buttons and menu items. Qt will automatically keep the state of the menu items and the toolbar buttons synchronized.

Adding and Connecting the Main Widget

Our application is a rich text editor, but so far there has been nowhere for the user to edit text. We'll add a **QTextEdit** widget and use Qt's signals and slots mechanism to minimize the code we have to write to make it functional.

Click *Qt Designer's* Richtext Editor (**Text Edit**) toolbar button and click in the center of the form. Click the form, then click the **Lay Out Horizontally** toolbar button. We're now ready to make the connections we need; but first we will do some renaming to make things easier to understand. Click on the *Text Edit* widget and change its *name* property to 'textEdit'. Change the *textFormat* property to 'RichText'. Change the *name* of the form to 'EditorForm' and its caption to 'Rich Edit'.

The **QTextEdit** widget contains built-in functionality for cut and paste and various other editing functions. By connecting the appropriate signals to our textEdit we can take advantage of this functionality without the need to write any code.

Connecting Actions to Change Font Attributes

Click the underlineAction in the *Action Editor*, then click the Connect button. The *Edit Connections* dialog will appear. Click the `toggled()` signal. Since we wish to connect this signal to the text edit widget drop down the Slots combobox and click textEdit. The text edit's slots that can respond to a toggled signal will appear in the right hand list box. Click the `setUnderline()` slot, then click **OK**.

Connect up the bold and italic actions in the same way. (For example, click the bold action and click Connect. Click the `toggled()` signal, change the Slots combobox item to textEdit and click the `setBold()` slot. Click **OK**.) If you preview the form you'll find that you can enter text and that choosing bold, italic or underline will work.

Connecting Actions to Implement Cut, Copy, Paste, Undo and Redo

The cut, copy, paste, undo and redo actions are connected in the same way as the font attributes. For example, click the `editPasteAction` action and click **Connect**. Click the `activated()` signal, change the Slots combobox item to `textEdit` and click the `paste()` slot. Click **OK** to save the connection. Connect the cut, copy, undo and redo actions in the same way. (For example, click the `editCopyAction` action, click **Connect**, click the `activated()` signal, change the Slots combobox item to `textEdit`, click the `copy()` slot then click **OK**.) Then the cut, copy, paste, undo and redo actions will all work in preview mode.

Connecting for Text Alignment

We need to create a slot to receive signals from alignment actions and set the `textEdit` widget's alignment state accordingly. One approach would be to connect each individual alignment action to our slot, but because the alignment actions are in a group we will connect the `alignActionGroup` to our slot and determine which alignment the user chose from the `QAction` pointer that is passed.

Create a new slot with the signature `changeAlignment(QAction *align)`. (Click **Edit|Slots**, click **New Slot**, enter the slot's signature and click **OK**.) Click `alignActionGroup` in the *Action Editor*, then click **Connect**. Connect the `selected(QAction*)` signal to our change alignment slot, then click **OK**.

We'll have to write the code to set the alignment ourselves; we'll cover this in *Aligning Text*.

Connecting for Font Names and Sizes

We'll start by dealing with font size since it's easiest. Click the **Connect Signals/Slots** toolbar button then click the spinbox and drag to the text edit widget; release on the text edit. Click the `valueChanged(int)` signal and on the `textEdit`'s `setPointSize(int)` slot. Click **OK** and font sizes are done. (Since font sizes are handled purely through built-in signals and slots they work in preview mode.)

Connect the `fontComboBox`'s `activated()` signal to the `textEdit`'s `setFamily()` slot. This connection will handle updating the `textEdit`'s font family with the user's choice of font. Note that when you invoke the *Edit Connections* dialog the first signal that is highlighted is `activated(int)`. Since the `setFamily()` slot takes a `QString` argument it does *not* appear in the list of slots. Only those slots which are compatible with the highlighted signal are shown, in this case, slots which take no argument or which take an integer argument. Click the `activated(const QString&)` signal and the list of slots will change to those which take no argument or which take a `QString` argument; the list will now include `setFamily()` since this takes a `QString` argument. We will have to populate the combobox with the font names for the user to choose from in code. (See the `init()` function in *Changing Fonts*.) It's a good idea to connect the `fontComboBox`'s `activate()` signal to the `textEdit`'s `setFocus()` slot; this will ensure that after the user has changed font the focus will return to the text.

The `richedit` application is nearly complete. We will have to write code to handle text alignment, font family and file loading and saving. We will also write the code for application exit to deal correctly with any unsaved changes.

Converting the Design into an Executable Application

We've built the user interface through *Qt Designer* and connected those slots that provided sufficient default functionality. The last steps are to code the slots that require customization and then to create `main.cpp` so that we can compile and build our application.

Implementing the Main Window's Functionality

When the user starts the `richedit` application we want the focus to be in the `textEdit` widget so we need to create an `init()` function with one line of code to achieve this. (All the code snippets are from `qt/tools/designer/examples/richedit/richedit.ui.h`.)

```
void EditorForm::init()
{
    textEdit->setFocus();
}
```

We'll add more to this function later.

New Files and Loading and Saving Existing Files

The code for these tasks is straightforward. When the user clicks **File|New** we check to see if there are unsaved changes in the existing text and give them the opportunity to save, continue without saving or cancel the operation. When the user opts to open an existing file or exit the application we perform the same check and offer them the same choices.

```
void EditorForm::fileNew()
{
    if ( saveAndContinue( "New" ) )
        textEdit->clear();
}
```

The fileNew() function clears the text and the filename.

```
void EditorForm::fileOpen()
{
    if ( saveAndContinue( "Open" ) ) {
        QString fn( QFileDialog::getOpenFileName(
            QString::null,
            "Rich Text Files (*.htm*)", this ) );
        if ( !fn.isEmpty() ) {
            fileName = fn;
            QFile file( fileName );
            if ( file.open( IO_ReadOnly ) ) {
                QTextStream ts( &file );
                textEdit->setText( ts.read() );
            }
        }
    }
}
```

The fileOpen() function asks the user to choose a file using QFileDialog::getOpenFileName(). If they choose a file we set the fileName member to its name, open it and read its contents directly into the text edit via a text stream.

```
void EditorForm::fileSave()
{
    if ( fileName.isEmpty() ) {
        fileSaveAs();
    } else {
        QFile f( fileName );
        if ( f.open( IO_WriteOnly ) ) {
            QTextStream ts( &f );
            ts <text();
            textEdit->setModified( FALSE );
        }
    }
}
```


If there is no current file name we call `fileSaveAs()` which will prompt for a file name and if a file name is given calls `fileSave()`. If we have a file name we open a file and write the text from the text edit into the file via a text stream. We also set the text edit's modified property to `FALSE`.

```
void EditorForm::fileSaveAs()
{
    QString fn = QFileDialog::getSaveFileName(
        "", "Rich Text Files (*.htm*)", this );

    if ( !fn.isEmpty() ) {
        fileName = fn;
        fileSave();
    }
}
```

The `fileSaveAs` function prompts the user for a file name and if they give a file name, saves the text to the file by calling `fileSave()`.

```
void EditorForm::fileExit()
{
    if ( saveAndContinue( "Exit" ) )
        QApplication->exit();
}
```

When we exit the application we must perform the same check for unsaved changes as we've done in the preceding functions, so we've included the `fileExit()` function's code here.

```
int EditorForm::saveAndContinue(const QString & action)
{
    int continueAction = 1;

    if ( textEdit->isModified() ) {
        switch( QMessageBox::information(
            this, "Rich Edit",
            "The document contains unsaved changes.\n"
            "Do you want to save the changes?",
            "&Save", "&Don't Save", "&Cancel " + action,
            0, // Enter == button 0
            2 ) ) { // Escape == button 2
        case 0: // Save; continue
            fileSave();
            break;
        case 1: // Do not save; continue
            break;
        case 2: // Cancel
            continueAction = 0;
            break;
        }
    }

    return continueAction;
}
```

The `saveAndContinue()` function is included for completeness.

Aligning Text

```
void EditorForm::changeAlignment(QAction * align)
```

```

{
    if ( align == leftAlignAction )
        textEdit->setAlignment( Qt::AlignLeft );
    else if ( align == rightAlignAction )
        textEdit->setAlignment( Qt::AlignRight );
    else if ( align == centerAlignAction )
        textEdit->setAlignment( Qt::AlignCenter );
}

```

We compare the chosen alignment action's pointer to the the pointers stored in the form and if we get a match set the appropriate alignment in the textEdit widget.

Changing Fonts

We've already connected the fontComboBox's activated() signal to the textEdit's setFamily() slot so we just have to populate the combobox with the font names when we call init().

```

void EditorForm::init()
{
    textEdit->setFocus();

    QFontDatabase fonts;
    fontComboBox->insertStringList( fonts.families() );
    QString font = textEdit->family();
    font = font.lower();
    for ( int i = 0 ; i < count(); i++ ) {
        if ( font == fontComboBox->text( i ) ) {
            fontComboBox->setCurrentItem( i );
            break;
        }
    }
}

```

The first line sets the focus as we've already mentioned. We then create a **QFontDatabase** object and insert its list of font families into the fontComboBox. Finally we set the fontComboBox's current item to the textEdit's current font.

Making the Application Run

With all the connections and code in place we are now ready to make our application run. Click on the Source tab of the Object Hierarchy window and click on the Includes (in Implementation) item. We need to include the files that our source code depends on. Right click the Includes item and click New. Type in <qapplication.h> for fileExit()'s exit() call. In the same way add <qmessagebox.h> for saveAndContinue()'s message box, <qfiledialog.h> for the fileOpen() and fileSaveAs() functions, and <qfontdatabase.h> for the **QFontDatabase** class in init().

We referred to a member variable, fileName, in our source code so we must add it to the form. Click the Source tab, right click the Class Variables item, click New from the pop up menu, then enter 'QString fileName';

The simplest way to create a new source file is by clicking **File | New** to invoke the 'New File' dialog, then click 'C++ Source' or 'C++ Header' as appropriate, then click **OK**. A new empty source window will appear. Click **File | Save** to invoke the *Save As* dialog, enter 'main.cpp', then click **Save**. Enter the following code in the main.cpp C++ editor window:

```

#include <qapplication.h>
#include "richedit.h"

```

```
int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

    EditorForm richeditForm;
    app.setMainWidget( &richeditForm );
    richeditForm.show();

    return app.exec();
}
```

All that's left to do is to generate the Makefile, compile and run. The Makefile is created with `qmake`: `qmake -o Makefile richedit.pro`.

The `richedit` application demonstrates how easy it is to create a Qt application's main window with menus and dockable toolbars. A great deal of functionality was obtained by connecting the appropriate built-in signals and slots. The remaining functionality was achieved by connecting built-in signals to our own custom slots. We could continue developing the application, for example updating the `fontComboBox`, the font size spinbox and the actions with the font attributes as the user moves the cursor through their text. But our objective has been to demonstrate the creation of a main window with actions, menus and toolbars so we must stop at this point and leave further development and experimentation to you.

The Designer Approach

Introduction

In Qt 2.x, *Qt Designer* was a visual form designer for editing files in the `.ui` file format. *Qt Designer*'s primary goal was to turn the most tedious part of GUI programming — dialog design — into a pleasant experience. From an architectural point of view, *Qt Designer* in 2.x is a fairly simple program. It reads and writes `.ui` files. Each `.ui` file contains an XML description of a single dialog form. A second utility — the user interface compiler `uic` — is used during the build process of an application to generate C++ code from those XML descriptions.

For Qt 3.0 our ambitions for *Qt Designer* have grown beyond single dialog editing. In addition to many new design features like the ability to creating main windows and actions, the new version introduces:

- *project management* for the user interface part of your application;
- *code in forms* *Qt Designer* provides a code editor so that you can code your slots directly; the code is stored in `.ui.h` files and eliminates the need for sub-classing (although you can still subclass if you prefer);
- *dynamic form loading* allows you to load `.ui` files at runtime which provides great scope for design customisation separate from the underlying code.

The purpose of this chapter is to explain the motivation for making these changes, describe the new concepts involved and show how these features work internally.

Qt Designer is and remains a visual design tool: it is not a complete integrated development environment. Our policy is to make GUI development as easy and powerful as possible without locking our users into any particular tool: *Qt Designer* makes it easy to create and modify GUI designs, but you can still achieve the same results directly in code using a plain text editor if you prefer.

To make working more convenient, *Qt Designer* now includes a C++ editor (as a plugin). If you want to create or edit a form, use *Qt Designer*. If you want edit code for that form, you can use the C++ editor in *Qt Designer* as well. This built-in editor has certain benefits stemming from its tight integration with the visual form design process that we will explain later. However, if you prefer using the editor you're used to, `vim`, `emacs`, `notepad`, Microsoft Visual Studio, etc. you can still do so.

Project management

Reading and writing single, non-connected `.ui` files is conceptually simple and worked fairly well in Qt 2.x. However, it lacked certain features that made us introduce project management for the GUI part of an application in *Qt Designer*. The main benefits of project management are:

- Grouping forms that belong together.
- Sharing images between different forms.
- Sharing database information between different forms.

The following sections explain these benefits in more detail, and why project management is required to achieve them.

Grouping forms

Grouping forms means that *Qt Designer* maintains a list of the `.ui` files that belong to the same project. This makes it easy to switch between forms with a single mouse click.

Sharing images in a image collection

In Qt 2.x's *Qt Designer* each form included the images it required and no images were shared. This led to duplication when several forms needed to use the same images. Furthermore the images were stored in the XML `.ui` files which made them large.

As a workaround, we introduced a pixmap-loading function that you could define in *Qt Designer*. It then was your responsibility to provide the implementation of this function in your application code. The big disadvantage of this approach was that you couldn't see the images during the design process in *Qt Designer*. This not only makes designing a form less visually interesting, but also has a noticeable impact on geometry management.

In the Qt 3.0 version of *Qt Designer* we've introduced the concept of a project image collection. If you use a project you can add images to the project's image collection, and these images can be shared and used by any of the forms you include in the project. The images are stored as PNGs (portable network graphics) in a subdirectory, `images/`, inside the project's directory. Whenever you modify the image collection, *Qt Designer* creates a source file which contains both the image data in binary format and a function to instantiate the images. The images are accessible by all forms in the project and the data is shared.

A further benefit of using an image collection is that the images are added to the default `QMimeSourceFactory`. This way they are accessible from rich-text labels, What's This? context help and even tooltips through standard HTML image tags. The *source* argument of the image tag is simply the image's name in the image collection. This also works during the design process in *Qt Designer*.

Sharing database settings

Qt 3.0 introduces a brand new database module, the Qt SQL module. *Qt Designer* is fully integrated with the SQL module and can show live data from the databases that you connect to.

When you've opened or created a project you can set up its database connections using the *Edit Database Connections* dialog (invoked by the **Project|Database Connections** menu option). The connections you make are stored in a `.db` file. When you reload a project you can reconnect by going to the *Edit Database Connections* dialog, clicking a connection in the list and clicking the **Connect** button.

In most non-trivial database applications you will want to access the database from more than one form. This is why the `.db` file is part of a project, not just part of a single form.

.pro files

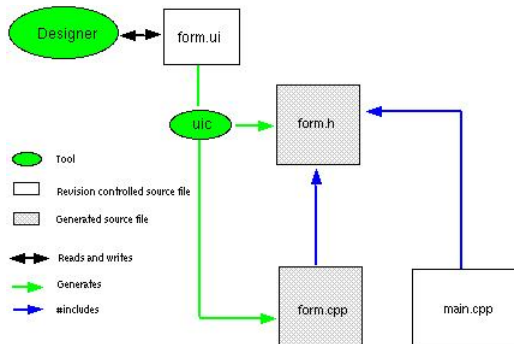
Qt Designer needs to store information on projects, for example, the list of forms, the image collection and information about available databases and how to access them. The majority of Qt users already use a project file format to create cross-platform makefiles: `tmake` (and with Qt 3.0 `qmake`) project `.pro` files. These files already contain the list of forms, `.ui` files, used in the project for `uic`.

We've extended the sections in the `.pro` file to include the extra information that *Qt Designer* needs to manage projects. For example, when you add a form to your project in *Qt Designer*, it is automatically added to the `FORMS` section of the project file, and thus `qmake` will generate the required build rules without any further work. Similarly, the images are added to the `IMAGES` section and thus gets automatically compiled into executable.

We don't force you to use `qmake`; if you prefer another build system, for example `automake/autoconf` or `jam`, you can still continue to use it. Look upon the `.pro` file as a file that describes the GUI part of your application. All you need to do — as previously — is add the `.ui` files and the images collection to your own Makefiles.

Extending the functionality of a form

First let us look at a small figure that shows the relationship between `.ui` files, generated code and application code:



Qt Designer reads and writes `.ui` files, e.g. `form.ui`. The user interface compiler, `uic`, creates both a header file, e.g. `form.h`, and an implementation file, e.g. `form.cpp`, from the `.ui` file. The application code in `main.cpp` `#includes` `form.h`. Typically `main.cpp` is used to instantiate the `QApplication` object and start off the event loop.

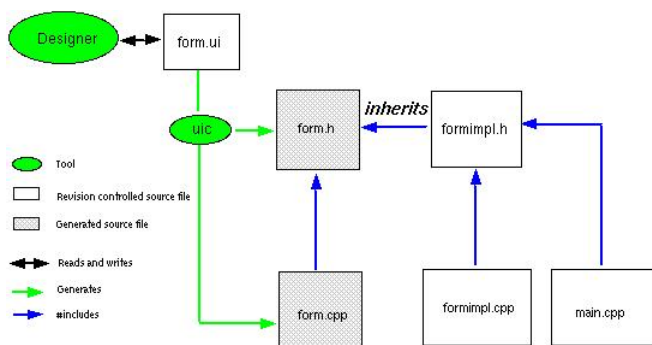
While this approach is simple, it isn't sufficient for more complex dialogs. Complex dialogs tend to have quite a lot of logic attached to the form's widgets, more logic than can usually be expressed with predefined signals and slots. One way of handling this extra logic is to write a controller class in the application code that adds functionality to the form. This is possible because `uic` generated classes expose a form's controls and their signals to the public space. The big disadvantage of this method is that it's not exactly Qt-style. If you were not using *Qt Designer*, you would almost always add the logic to the form itself, where it belongs.

This is why the capability of adding custom slots and member variables to a form was added to *Qt Designer* early on. The big additional benefit with this approach is that you can use *Qt Designer* to connect signals to those custom slots, in the same elegant graphical way that is used to connect signals to predefined slots. The `uic` then adds an empty stub for each custom slot to the generated `form.cpp` implementation file.

The big question now is how to add custom implementation code to those custom slots. Adding code to the generated `form.cpp` is not an option, as this file gets recreated by the `uic` whenever the form changes — and we don't want a combination of generated and handwritten code. There are two possible solutions, which we'll cover next.

The subclassing approach

A very clean way to implement custom slots for generated forms is via C++ inheritance as shown in the next figure:



Here the user wrote an additional class **FormImpl**, which is split into the header file `formimpl.h` and the implementation file `formimpl.cpp`. The header file includes the `uic` generated `form.h` and reimplements all the custom slots. This is possible because `uic` generated custom slots are virtual. In addition to implementing custom slots, this approach gives the user a way to do extra initialization work in the constructor of the subclass, and extra cleanups

in the destructor.

Because of these benefits and its flexibility, this approach became the primary way of using *Qt Designer* in Qt 2.x.

Note: To keep the namespace clean, most users did not follow the Form and FormImpl naming scheme shown in the figure, but instead named their *Qt Designer* forms FormBase and their subclasses Form. This made a lot of sense, because they always subclassed and were using those subclasses in application code.

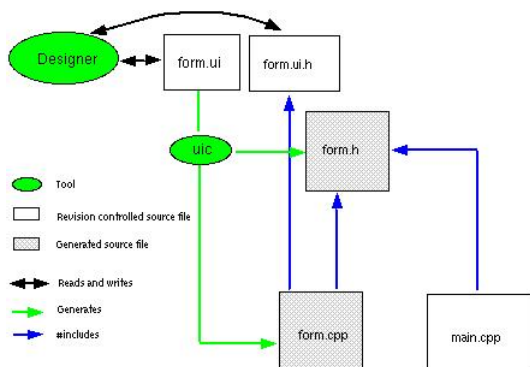
The ui.h extension approach

Despite its flexibility and cleanness, the subclassing approach has some disadvantages:

- Subclassing is not natural and easy for everybody. Newcomers to object-oriented techniques may feel uneasy about being *forced* to subclass for such a simple and natural thing like the implementation of a custom slot.
- Inheriting generated classes is an additional possible source of programming mistakes, especially if the number of reimplemented functions is high and the signatures change often during the design process. To make the development process smoother, `uic` generates empty stubs for custom slots rather than pure virtual functions. While this approach keeps the code compiling and running, programmers can find themselves in a situation where they miss a runtime warning message and lose time before they find a small spelling error in their subclass.
- In larger projects with hundreds of forms, the additional subclasses can make a noticeable difference in terms of compilation speed and code size.

There may be more disadvantages, but these were reason enough for us to investigate alternative solutions. For Qt 3.0, we came up with a new concept, the *ui.h extension*.

This is how it works:



In addition to the `.ui` file, `form.ui`, *Qt Designer* reads and writes another associated file `form.ui.h`. This `.ui.h` file is an *ordinary C++ source file* that contains *implementations* of custom slots. The file gets included from the generated form implementation file `form.cpp` and thus can be totally ignored by other user code. The reason we use a `.h` extension for the `.ui.h` file even though it contains C++ code is because it is always *included*, and because it is easier to integrate into the build process with a `.h` extension.

The `form.ui.h` file has a special position among all other files. It is a *shared* source file that gets written and read by both the user and *Qt Designer*. As such it is an ordinary revision controlled source file and not generated by `uic`. *Qt Designer's* responsibility is to keep the file in sync with the custom slot definitions of the associated form:

1. Whenever the users adds a new slots to the form, *Qt Designer* adds a stub to the `.ui.h` file.
2. Whenever the user changes a custom slot's signature, *Qt Designer* updates the corresponding implementation.
3. Whenever the user removes a custom slot, *Qt Designer* removes it from the `.ui.h` file.

This way integrity is guaranteed, there is no more need for subclassing and no more danger of forgotten or misspelled slots in subclasses.

You can edit `.ui.h` files either directly in *Qt Designer* with the built-in C++ editor plugin, or with whatever editor you prefer. You should only put slot implementations in the `.ui.h` file and you should *always* add, delete or rename slots *within Qt Designer*. You can edit the implementations of the slots either within *Qt Designer* or using your own editor; if you use your own editor *Qt Designer* will keep your changes.

Construction and destruction

The `ui.h` extension approach has one disadvantage compared to subclassing. The `ui.h` file only contains custom slot implementations, but the objects are still entirely constructed and destructed inside the generated `form.cpp` code. This leaves the user without the possibility of doing further form initializations or cleanups that you normally would do within the constructor and destructor functions of a C++ class.

To work around this limitation, we created the `init/destroy` convention. If you add a slot `Form::init()` to your form, this slot will be called automatically at the end of the generated form constructor. Similarly, if you add a slot `Form::destroy()` to your form, the slot will automatically be invoked by the destructor before any form controls get deleted. (These slots should return `void`.) If you prefer to use your own editor you must still create these functions in *Qt Designer*; once created you can then write your implementation code either using *Qt Designer's* C++ editor plugin or using your own editor.

Loading forms dynamically

We extracted the part of *Qt Designer* that is responsible for loading and previewing a form into a library of its own, `libqui`. A new class **QWidgetFactory** makes it possible to load `.ui` files at runtime and instantiate forms from them.

This dynamic approach keeps the GUI design and the code separate and is useful in environments where the GUI may have to change more often than the underlying application logic. Ultimately, you can provide users of your application the ability to modify the graphical user interface without the need for a complete C++ development environment.

Since the `.ui` file is not compiled it cannot include any C++ code, (e.g. custom slot implementations). We provide a way of adding those implementations via a controlling `QObject` subclass that you pass as receiver to the widget factory.

This concept and its usage is explained in detail in the Subclassing and Dynamic Dialogs chapter.

Subclassing and Dynamic Dialogs

This chapter describes two different approaches that you can take to creating forms with *Qt Designer*. Subclassing is used to extend the functionality of a form by creating your own class based upon a form you create in *Qt Designer*. Dynamic dialogs are `.ui` files which can be executed by a Qt application; this keeps the GUI design and the code separate and is useful in environments where the GUI may have to change more often than the underlying application logic.

Subclassing

We'll start with a general description of how to subclass a form and follow with a short example. Note that subclassing has some disadvantages compared with putting your code into a form directly; see *Extending the functionality of a form in The Designer Approach* chapter for details.

Generating Source Code from *Qt Designer* `.ui` Files

Qt Designer reads and writes `qmake .pro` (project) files which are used to record the files used to build the application and from which Makefiles are generated. *Qt Designer* also reads and writes `.ui` (user interface) files. These are XML files that record the widgets, layouts, source code and settings you've used for a form. Every `.ui` file is converted by the `uic` (user interface compiler) into a C++ `.h` file and a C++ `.cpp` file. These C++ files are then read by `moc` (meta object compiler), and finally compiled by your compiler into a working application.

If you create applications wholly within *Qt Designer* you only need to create a `main.cpp`.

If you create the `main.cpp` file within *Qt Designer*, it will automatically be added to your project file by *Qt Designer*. If you create the `main.cpp` file outside of *Qt Designer* you must add it to the project file manually by adding the following line at the end of your project's `.pro` file:

```
SOURCES += main.cpp
```

You can then use `qmake` to generate the Makefile. (For example `qmake -o Makefile myproject.pro`.) Running `make` (Linux, Unix or Borland compilers), or `nmake` (Visual C++), will then call `uic`, `moc` and your compiler as necessary to build your application.

If you use *Qt Designer* to create your main window and dialogs, but also add other C++ files, or if you subclass any of your forms you will need to add these files to the `.pro` file so that they are compiled with the rest of your application's source files. Each `.h` file that you create separately from *Qt Designer* should be added to the `HEADERS` line, and each `.cpp` file should be added to the `SOURCES` line, just as we've done for `main.cpp`. If you get undefined reference errors it is worth checking that you've added the names of all your header and implementation files to the `.pro` file.

Subclassing a Form

When subclassing a form it is helpful to use a naming convention to help us identify which files are generated from *Qt Designer's* `.ui` files and which are hand coded.

Suppose, for example, that we are developing a dialog and writing the code directly in *Qt Designer*. We might call our dialog 'OptionsForm' and the .ui file, optionsform.ui. The automatically generated files will be optionsform.h and optionsform.cpp.

If we were developing another dialog, but this time one that we intended to subclass, we want to make it easy to distinguish between the automatically generated files and our hand coded files. For example, we might call our dialog 'SettingsFormBase' and the .ui file settingsformbase.ui. The automatically generated files would then be called settingsformbase.h and settingsformbase.cpp. We would then call our subclass 'SettingsForm' and code it in the files settingsform.h and settingsform.cpp.

Any subclass of a form should include the `Q_OBJECT` macro so that slots and signals will work correctly. Once you've created your subclass be sure to add the .h and the .cpp files to the .pro project file. For example we would add the following lines for our subclassed 'SettingsForm' at the end of the .pro file:

```
HEADERS += settingsform.h
SOURCES += settingsform.cpp
```

The simplest way to create a new source file is by clicking **File|New** to invoke the 'New File' dialog, then click 'C++ Source' or 'C++ Header' as appropriate, then click **OK**. A new empty source window will appear. You don't need to manually edit the .pro file since *Qt Designer* will add them for you automatically.

Qt Designer will have added

```
FORMS = settingsformbase.ui
```

to the project file. The settingsformbase.h and settingsformbase.cpp files will be generated from the .ui file automatically.

A Subclassing Example

We will write a small example dialog to show the use of subclassing in practice. The dialog will present a choice of customer credit ratings with an option of choosing a 'special' rating for which a specific amount must be given. We'll implement the functionality in a subclass. We'll start by creating the base form and connecting its signals and slots, then we'll create the subclass and a simple main.cpp so that we can test it.

Designing the Form

We'll begin by creating a new project. Click **File|New**, then click the 'C++ Project' icon to invoke the *Project Settings* dialog. Click the ellipsis button to invoke the *Save As* dialog; navigate to the project's directory (creating it if necessary). Make sure you're in the project's directory, then enter a project name of 'credit.pro'. Click the **Save** button to return to the *Project Settings* dialog, then click **OK**. Now we'll add a form to the project. Click **File|New** to invoke the *New File* dialog. The default form is Dialog which is what we want; click **OK**. Resize the form to make it smaller; it should be about 2 inches (5 cm) square. Change the form's name to 'CreditFormBase' and the caption to 'Credit Rating'. Save the form as creditformbase.ui.

We'll now add the widgets we need.

1. Click the **Button Group** toolbar button, then click near the top left of the form. Resize the button group so that it takes up approximately half the form. Change the button group's *name* to 'creditButtonGroup' and its *title* property to 'Credit Rating'.
2. We'll now add some radio buttons. *Double* click the **Radio Button** toolbar button. Click towards the top of the Credit Rating button group and a radio button will appear. Click below this button, to create a second radio button, then click below the second button to create a third. Now we will switch off the effect of the *double* click by clicking the **Pointer** (arrow) toolbar button. The pointer will now behave normally, i.e. clicking the form will no longer create more radio buttons. Change the first radio button's *name* to 'stdRadioButton' and

its text to '&Standard'. Change its *checked* property to True. Change the second button's name to 'noneRadioButton' and its text to '&None'. Change the third radio button's properties to 'specialRadioButton' and 'Sp&ecial' respectively.

3. If the user chooses the special credit rating they must specify an amount. Click the **SpinBox** toolbar button and click the form just below the button group. Change the spin box's *name* to 'amountSpinBox'. Change its *prefix* to '\$' (note the space), its *maxValue* to '100000' and its *lineStep* to '10000'. Change its *enabled* property to False.
4. Click the **Push Button** toolbar button and click the form below the spin box. Change the button's *name* to 'okPushButton', its *text* to 'OK' and its *default* property to 'True'. Add a second button to the right of the first. Change the second button's *name* to 'cancelPushButton' and its *text* to 'Cancel'.

We'll now lay out the widgets and connect up the slots we need.

1. Click the credit rating group box then press **Ctrl+L** (lay out vertically).
2. Click the form so that the button group is no longer selected. **Ctrl+Click** the OK button and drag the rubber band to touch the Cancel button, then release. Press **Ctrl+H**.
3. Click the form, then press **Ctrl+L**.

The widgets will be laid out vertically, each one stretching to fill up the maximum space both vertically and horizontally. The buttons look rather large since they've expanded to take up the full width of the form. It might look more attractive to make the buttons smaller using spacers. Click the OK button, then press **Ctrl+B** (break layout). Resize both buttons to make them narrower leaving space on either side of them. Click the **Spacer** toolbar button then click to the left of the OK button; click Horizontal from the pop up spacer menu. Copy this spacer and place the copy between the two buttons. Copy the spacer again and place the copy to the right of the Cancel button. (For the second and third spacers, click on the first spacer, press **Ctrl+C** then **Ctrl+V**. Drag the new spacer to the desired position.) **Ctrl+Click** the left most spacer and drag the rubber band so that it touches the buttons and the spacers, then release. Press **Ctrl+H**. Click the form then press **Ctrl+L**.

We'll now connect the signals and slots. Press **F3** (connect signals/slots), then click the OK button. Drag to the form and release. In the *Edit Connections* dialog that pops up connect the `clicked()` signal to the `accept()` slot. (Click the `clicked()` signal, click the `accept()` slot, then click **OK**.) Connect the Cancel button to the `reject()` slot using the same technique.

We want the amount spin box to be enabled only if the special radio button is checked. Press **F3** (connect signals/slots), then click the special radio button. Drag to the spin box and release. In the *Edit Connections* dialog that pops up click the `toggled()` signal and the `setEnabled()` slot.

If the user checks the standard or none radio buttons we want to set the amount accordingly. Press **F3**, then click the credit rating button group. Drag to the form and release. Click the `clicked()` signal. We want to connect this signal to our own custom slot, but we haven't created one yet. Click the **Edit Slots** button and the Edit Slots dialog will pop up. Click **New Slot** and change the Slot's name to 'setAmount()'. Click **OK**. This new slot is now available in the list of slots. Click the `setAmount()` slot then click **OK**.

We'll subclass the form to set the amount in the spin box depending on which radio button is checked. Save the form as 'creditformbase.ui' (press **Ctrl+S**).

Creating the Test Harness

Although we intend our dialog to be used within an application it is useful to create a test harness so that we can develop and test it stand-alone. Right click the 'Source Files' entry in the Files window then click **Add new source file to project**. This will invoke the *Save As* dialog; enter 'main.cpp' and click **Save**. In the editor window that pops up, enter the following code:

```
#include
#include "creditformbase.h"
```

```
int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

    CreditFormBase creditForm;
    app.setMainWidget( &creditForm );
    creditForm.show();

    return app.exec();
}
```

Note that we're including `creditformbase.h` and instantiating a `CreditFormBase` object; once we've written our subclass we'll replace the header with our subclass, `creditform.h`, and instantiate a `CreditForm`.

We can now generate the application with `qmake`, e.g. `qmake -o Makefile credit.pro`, make it and run it. The form should run fine, but doesn't yet have the behaviour we require.

Creating the Subclass

We need to create a header and an implementation file for our subclass. The code for our subclass is minimal. The header file is `qt/tools/designer/examples/credit/creditform.h`:

```
#include "creditformbase.h"

class CreditForm : public CreditFormBase
{
    Q_OBJECT
public:
    CreditForm( QWidget* parent = 0, const char* name = 0,
               bool modal = FALSE, WFlags fl = 0 );
    ~CreditForm();
public slots:
    void setAmount();
};
```

We've declared the slot, `setAmount()`, that we created in *Qt Designer*. The `Q_OBJECT` macro is included because it is essential for classes that use signals and slots.

The implementation in `qt/tools/designer/examples/credit/creditform.cpp` is simple:

```
#include <radiobutton.h>
#include <qspinbox.h>
#include "creditform.h"

CreditForm::CreditForm( QWidget* parent, const char* name,
                       bool modal, WFlags fl )
    : CreditFormBase( parent, name, modal, fl )
{
    setAmount();
}

CreditForm::~CreditForm() { /* NOOP */ }

void CreditForm::setAmount()
{
    if ( stdRadioButton->isChecked() )
        amountSpinBox->setValue( amountSpinBox->maxValue() / 2 );
}
```

```

        else if ( noneRadioButton->isChecked() )
            amountSpinBox->setValue( amountSpinBox->minValue() );
    }

```

We call `setAmount()` in the constructor to ensure that the correct amount is shown when the form starts based on whichever radio button we checked in *Qt Designer*. In `setAmount()` we set the amount if the standard or none radio button is checked. If the user has checked the special radio button they are free to change the amount themselves.

To be able to test our subclass we change `main.cpp` to include `creditform.h` rather than `creditformbase.h` and change the instantiation of the `creditForm` object:

```

#include <qapplication.h>
#include "creditform.h"

int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

    CreditForm creditForm;
    app.setMainWidget( &creditForm );
    creditForm.show();

    return app.exec();
}

```

If you created the `creditform.h` and `creditform.cpp` files in *Qt Designer*, they are already in the project file, but if you created them manually you must also update the project file by adding these two new lines at the end:

```

HEADERS += creditform.h
SOURCES += creditform.cpp

```

To test the form rerun `qmake` to regenerate the Makefile, then `make` and run.

The subclassing example we've used is simple, but this reflects subclassing forms in Qt: it is easy to do.

Creating Dynamic Dialogs from .ui Files

Qt programs are capable of loading *Qt Designer* `.ui` files and instantiating the forms represented by the `.ui` files. Since the `.ui` file is not compiled it cannot include any C++ code, (e.g. slot implementations). In this section we will explain how to load a dynamic dialog and how to create a class that can be used to implement the dynamic dialog's custom slots.

We will use the credit form that we created in the subclassing section as our example form. We will start by simply instantiating and running the form and then we'll cover how to implement custom slots.

We'll create a `main.cpp` file to use as a test harness, and manually create a project file.

Creating the Project File

The project file `qt/tools/designer/examples/receiver1/receiver.pro` looks like this:

```

TEMPLATE    = app
CONFIG      += qt warn_on_release
TARGET      = receiver
SOURCES     += main.cpp

```

```

unix:LIBS += -lqui
win32:LIBS += $(QTDIR)/lib/qui.lib
FORMS      = mainform.ui
LANGUAGE   = C++
INCLUDEPATH += $(QTDIR)/tools/designer/uilib

```

We do *not* include the `creditformbase.ui` file since this file will be read at runtime, as we'll see shortly. We must include the `qresource` library since the functionality we require is not part of the standard Qt library.

Creating main.cpp

The `main.cpp` is quite standard. It will invoke the form we're going to create in *Qt Designer* as its main form. This form will then load and execute the dynamic dialog.

```

#include <qapplication.h>
#include "mainform.h"

int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

    MainForm *mainForm = new MainForm;
    app.setMainWidget( mainForm );
    mainForm->show();

    return app.exec();
}

```

We create a new instance of our `MainForm` class, set it to be the main widget, show it and enter the event loop in the `app.exec()` call.

Creating the Main Form

Designing the Form

1. Open the `receiver.pro` project file in *Qt Designer*. We'll create a dialog as our main window which we'll use to invoke the dynamic dialog. Press **Ctrl+N** to launch the *New File* dialog and click **OK** to get the default which is a dialog. Change the dialog's name to 'MainForm' and its caption to 'Main Form'. Add two buttons, one called 'creditPushButton' with the text '&Credit Dialog', and the other called 'quitPushButton' with the text '&Quit'. (For each button click the **Push Button** toolbar button, then click the form. Change the properties in the property window to those we've just described.)
2. We will now add a couple of labels so that we can show the settings the user chose in the dynamic dialog. Click the **Text Label** toolbar button, then click the form below the Credit Dialog button. Change the label's *text* to 'Credit Rating'. Add another text label below the Quit button. Change its *name* to 'ratingTextLabel' and its *text* to 'Unrated'.
3. We'll now lay out the widgets. Click the form then press **Ctrl+G** (lay out in a grid).
4. We'll now handle the signals and slots connections. Press **F3** (connect signals/slots). Click the Credit Dialog button, drag to the form and release. Click the `clicked()` signal. We'll need to implement a custom slot. Click **Edit Slots** to invoke the *Edit Slots* dialog. Click **New Slot** and type in the Slot name 'creditDialog()'. Click **OK**. The new slot is now in the list of slots; click the `creditDialog()` slot to make the connection then click **OK**. Connect the Quit button's `clicked()` signal to the dialog's `accept()` function. (Press **F3**. Click the Quit button and drag to the form; release. Click the `clicked()` signal and the `accept()` slot, then click **OK**.)

Save the form and call it `mainform.ui`. (Press **Ctrl+S** and enter the filename.) In the next section we'll write the code for loading and launching the dynamic dialog directly in *Qt Designer*.

Loading and Executing a Dynamic Dialog

We'll now add the code to invoke the credit dialog. Before we can do this we need to add the widget factory's header file to the form. Click the Source tab in the Object Hierarchy. Right click Included (in Implementation), then click **New**. Type in '<qwidgetfactory.h>', then press **Enter**. Because we will need to access the spin box in the dynamic dialog we must add its header file. Right click Included (in Implementation), then click **New**. Type in '<qspinbox.h>', then press **Enter**.

In our main form we created a slot called `creditDialog()`. We will implement this slot directly in *Qt Designer* and use it to load and execute the dynamic dialog. The code is taken from `qt/tools/designer/examples/receiver1/mainform.ui.h` which contains the C++ implementation of `mainform.ui`'s slots.

```
void MainForm::creditDialog()
{
    QDialog *creditForm = (QDialog *)
        QWidgetFactory::create( "../credit/creditformbase.ui" );
    // Set up the dynamic dialog here

    if ( creditForm->exec() ) {
        // The user accepted, act accordingly
        QSpinBox *amount = (QSpinBox *) creditForm->child( "amountSpinBox", "QSpinBox" );
        if ( amount )
            ratingTextLabel->setText( amount->text() );
    }
    delete creditForm;
}
```

The `create()` function is a static **QWidgetFactory** function. It loads the specified `.ui` file and returns a pointer to the toplevel **QWidget** created from the `.ui` file. We have cast the pointer to **QDialog** since we know that the `creditformbase.ui` file defines a **QDialog**. After creating the dialog we `exec()` it. If the user clicked **OK** the dialog returns `Accepted` and we enter the body of the `if` statement. We want to know the amount of credit that the user selected. We call the `child()` function on the dialog passing it the name of the widget we're interested in. The `child()` function returns a pointer to the widget with the name we passed, or returns 0 if no widget of that name was found. In the example we call `child()` to get a pointer to the 'amountSpinBox'. If the pointer we get back is not 0 we set the rating text to the amount in the dialog's spin box. At the end we delete the dynamic dialog. Deleting the dialog ensures that we free up its resources as soon as it is no longer required.

We used the `child()` to gain access to a widget within the dynamic dialog, passing it the name of the widget we were interested in. In some situations we might not know what a widget is called. We can access the first widget of a specified class by calling `child()` with a null widget name and a classname, e.g. `child(0, "QPushButton")`. This will return a pointer to the first **QPushButton** it finds (or 0 if there isn't one). If you want pointers to all the widgets of a given class you can call the `QObject::queryList()` function, passing it the name of the class. It returns a **QObjectList** pointer which points to every object in the dialog that is derived from the given class. See the online **QObject** documentation for further details.

Implementing Slots for Dynamic Dialogs

There is one outstanding issue that we haven't addressed: the dynamic dialog does not have the behaviour of the original credit dialog because we have not implemented the `setAmount()` slot. We can implement slots for dynamic dialogs by creating a **QObject** subclass. We then create an instance of this subclass and pass a pointer to it to the `QWidgetFactory::create()` function which will connect the dynamic dialog's signals to the slots implemented in our subclass.

We need to create a **QObject** subclass and change our `creditDialog()` to create an instance of our subclass that can be passed to the `QWidgetFactory::create()` function. Here is the modified `creditDialog()` function from the `qt/tools/designer/examples/receiver2/mainform.ui.h` file that contains the code for `mainform.ui`'s slots:

```

void MainForm::creditDialog()
{
    Receiver *receiver = new Receiver;
    QDialog *creditForm = (QDialog *)
    QWidgetFactory::create( "../credit/creditformbase.ui", receiver );
    receiver->setParent( creditForm );

    // Set up the dynamic dialog here

    if ( creditForm->exec() ) {
        // The user accepted, act accordingly
        QSpinBox *amount = (QSpinBox *) creditForm->child( "amountSpinBox", "QSpinBox" );
        if ( amount )
            ratingTextLabel->setText( amount->text() );
    }

    delete receiver;
    delete creditForm;
}

```

We create a new instance of our 'Receiver' subclass. (We'll write the code for this class shortly.) We then create the **QDialog** using `QWidgetFactory::create()`. This call differs from our previous example because we pass in the subclass object so that the `create()` function can set up the signals/slots connections automatically for us. Since our slot must access the widgets in the dynamic form we pass a pointer to the form to the receiver object through our `setParent()` function. The remainder of the function is the same as before except that we delete our receiver object.

We'll now look at the implementation of our 'Receiver' subclass. The code is taken from `qt/tools/designer/examples/receiver2/receiver.h` and the corresponding `receiver.cpp` file. We'll start with the header file.

```

#include <qobject.h>
#include <qdialog.h>

class Receiver : public QObject
{
    Q_OBJECT
public:
    void setParent( QDialog *parent );
public slots:
    void setAmount();
private:
    QDialog *p;
};

```

Our class must be a **QObject** subclass and because we're using signals and slots it must include the `Q_OBJECT` macro. We declare a function and the `setAmount()` slot that we wish to implement as well as a private **QDialog** pointer.

We'll discuss the implementation of each function in `receiver.cpp` separately.

```

void Receiver::setParent( QDialog *parent )
{
    p = parent;
    setAmount();
}

```

The `setParent()` function assigns a pointer to the dynamic dialog to our private pointer. We could not do this in a constructor call because we have to construct our Receiver object before we call `QWidgetFactory::create()`, since

we must pass the Receiver object to the `create()` function. Once we've called `create()` we then have a pointer to the dynamic dialog which we can then pass via `setParent()` to our Receiver class. In the subclass version of this example we called `setAmount()` in the constructor; but we cannot do that here because the implementation of `setAmount()` depends on knowledge of the dynamic dialog which is not available at construction time. Because of this we call `setAmount()` in the `setParent()` function.

```
void Receiver::setAmount()
{
    QSpinBox *amount =
        (QSpinBox *) p->child( "amountSpinBox", "QSpinBox" );

    QRadioButton *radio =
        (QRadioButton *) p->child( "stdRadioButton", "QRadioButton" );
    if ( radio && radio->isChecked() ) {
        if ( amount )
            amount->setValue( amount->maxValue() / 2 );
        return;
    }

    radio =
        (QRadioButton *) p->child( "noneRadioButton", "QRadioButton" );
    if ( radio && radio->isChecked() )
        if ( amount )
            amount->setValue( amount->minValue() );
}
```

Since we may be updating the amount spin box we need to get a pointer to it. We call `child()` on the pointer `p` which points to the dynamic dialog assigned in the `setParent()` call. We cast the resulting pointer to the correct type so that we can call any functions relevant to that type. In the example we call `child()` to get a pointer to the amount spin box, and then call `child()` again to get a pointer to the 'stdRadioButton'. If we get a pointer to the radio button and the button is checked we set the amount providing we have a pointer to the amount spin box. If this radio button was checked we're finished so we return. If the 'stdRadioButton' isn't checked we get a pointer to the 'noneRadioButton' and set the amount if this button is checked. We do nothing if the 'specialRadioButton' is checked because the user is free to enter a value of their choice.

Compiling vs Dynamically Loading Dialogs

The differences between using a 'compiled in' .ui file and a dynamically loaded .ui file are these:

- Dynamic dialogs cannot have any C++ code in the .ui file; any custom slots must be implemented via a **QObject** subclass. Compiled dialogs can contain code either in the .ui file or in a subclass.
- Dynamic dialogs will load slower because the .ui file must be read and a **QWidget** instance instantiated based on the .ui file's parse tree. Compiled code will load much faster because no file reading or parsing is necessary. Note that the user may not notice any difference in speed since the difference may be mere fractions of a second.
- Dynamic dialogs allow you to change the .ui file independently of the code so long as none of the changes impact the code. This means that you can change the appearance of the form, e.g. move widgets and lay them out differently. If you want to change a compiled dialog you must change the .ui file and recompile. If you are building an application and want your customers to be able to customize aspects of the user interface you can give them a copy of *Qt Designer* and use dynamic dialogs.

Creating Custom Widgets

Custom widgets are created in code. They may comprise a combination of existing widgets but with additional functionality, slots and signals, or they may be written from scratch, or a mixture of both.

Qt Designer provides two mechanisms for incorporating custom widgets:

1. The original method involves little more than completing a dialog box. Widgets incorporated this way appear as flat pixmaps when added to a form in *Qt Designer*, even in preview mode. They only appear in their true form at runtime. We'll explain how to create custom widgets using the original approach in "Simple Custom Widgets".
2. The new method involves embedding the widgets in a plugin. Widgets that are incorporated through plugins appear in their true form in *Qt Designer*, both when laying out the form and in preview mode. This approach provides more power and flexibility than the original method and is covered in Creating Custom Widgets with Plugins.

Simple Custom Widgets

There are two stages to creating a custom widget. Firstly we must create a class that defines the widget, and secondly we must incorporate the widget into *Qt Designer*. Creating the widget has to be done whether we are creating a simple custom widget or a plugin, but for simple custom widgets the incorporation into *Qt Designer* is very easy.

We will create a VCR style widget comprising four buttons, rewind, play, next and stop. The widget will emit signals according to which button is clicked.

Coding the Custom Widget

A custom widget may consist of one or more standard widgets placed together in a particular combination, or may be written from scratch. We will combine some **QPushButton** widgets to form the basis of our custom widget.

We'll look at the header file, `qt/tools/designer/examples/vcr/vcr.h` first.

```
#include <qwidget.h>

class Vcr : public QWidget
{
    Q_OBJECT
public:
    Vcr( QWidget *parent = 0, const char *name = 0 );
    ~Vcr() {}
signals:
    void rewind();
    void play();
    void next();
};
```

```

    void stop();
};

```

We include `QWidget.h` since we'll be deriving our custom widget from **QWidget**. We declare a constructor where the widget will be created and the four signals we want our widget to emit. Since we're using signals we must also include the `Q_OBJECT` macro.

The implementation is straightforward. The only function we implement is the constructor. The rest of the file consists of include statements and embedded `.xpm` images.

```

Vcr::Vcr( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
    QHBoxLayout *layout = new QHBoxLayout( this );
    layout->setMargin( 0 );

    QPushButton *rewind = new QPushButton( QPixmap( rewind_xpm ), 0, this, "vcr_rewind" );
    layout->addWidget( rewind );
}

```

We create a **QHBoxLayout** in which we'll place the buttons. We've only shown the rewind button in the code above since all the others are identical except for the names of the buttons, pixmaps and signals. For each of the buttons we require we call the **QPushButton** constructor passing it the appropriate embedded pixmap. We then add it to the layout. Finally we connect the button's `clicked()` signal to the appropriate *signal*. Since the `clicked()` signals aren't specific to our widget we want to emit signals that reflect the widget's use. The `rewind()`, `play()`, etc. signals are meaningful in the context of our widget so we propagate each button's `clicked()` signal to the appropriate widget-specific signal.

The implementation is complete, but to make sure that our widget compiles and runs we'll create a tiny test harness. The test harness will require two files, a `.pro` project file and a `main.cpp`. The `qt/tools/designer/examples/vcr/vcr.pro` project file:

```

SOURCES += vcr.cpp main.cpp
HEADERS += vcr.h
TARGET   = vcr
TEMPLATE     = app
CONFIG += qt warn_on_release
DBFILE = vcr.db
PROJECTNAME = Vcr
LANGUAGE = C++
CPP_ALWAYS_CREATE_SOURCE = TRUE

```

The `qt/tools/designer/examples/vcr/main.cpp` file is also brief:

```

#include <qapplication.h>
#include "vcr.h"

int main( int argc, char ** argv )
{
    QApplication app( argc, argv );
    Vcr *vcr = new Vcr;
    vcr->show();
    return app.exec();
}

```

Once we're satisfied that the custom widget compiles and runs we are ready to incorporate it into *Qt Designer*.

In Base-class Templates the creation of a container custom widget is described.

Adding the Custom Widget to Qt Designer

Click **Tools|Custom|Edit Custom Widgets** to invoke the *Edit Custom Widgets* dialog.

1. Click **New Widget** so that we are ready to add our new widget.
2. Change the Class name from 'MyCustomWidget' to 'Vcr'.
3. Click the ellipsis (...) button to the right of the Headerfile line edit to invoke the file Open dialog. Locate `vcr.h`, select it, and click **Open**. It will now appear as the header file.
4. If you have a pixmap that you want to use to identify your widget on the toolbar click the ellipsis button to the right of Pixmap property. (The ellipsis button appears when you click in the Value part of the Properties list by a *pixmap* or *iconSet* property.)

In our example we have the file `qt/tools/designer/examples/vcr/play.xpm` which we'll use for this purpose.

5. Since we know the minimum sensible size for our widget we'll put these values into the Size Hint spin boxes. Enter a width of 80 (in the left hand spin box), and a height of 20 (in the right hand spin box).

The remaining items to be completed will depend on the characteristics of the widget you've created. If, for example, your widget can be used to contain other widgets you'd check the Container Widget checkbox. In the case of our Vcr example the only items we need to add are its signals.

Click the Signals tab. Click the **New Signal** button and type in the signal name 'rewind()'. Click **New Signal** again and this time type in 'play()'. Add the 'next()' and 'stop()' signals in the same way.

Since our example hasn't any slots or properties we've finished and can click **Close**. A new icon will appear in *Qt Designer's* toolbars which represents the new widget. If you create a new form you can add Vcr widgets and connect the Vcr's signals to your slots.

Incorporating custom widgets that have their own slots and properties is achieved in a similar way to adding signals. All the required information is in our custom widget's header file.

Creating Custom Widgets with Plugins

This section will show you how to write a custom widget and how to embed the custom widget into a plugin. There are no restrictions or special considerations that must be taken into account when creating a widget that is destined to become a plugin. If you are an experienced Qt programmer you can safely skip the section on creating a custom widget and go directly to Creating a Plugin.

Creating a Custom Widget

A custom widget is often a specialization (subclass) of another widget or a combination of widgets working together or a blend of both these approaches. If you simply want a collection of widgets in a particular configuration it is easiest to create them, select them as a group, and copy and paste them as required within *Qt Designer*. Custom widgets are generally created when you need to add new functionality to existing widgets or groups of widgets.

We have two recommendations that you should consider when creating a custom widget for a plugin:

1. Using Qt's property system will provide *Qt Designer* users with a direct means of configuring the widget through the property editor. (See the Qt Properties documentation.)
2. Consider making your widget's public 'set' functions into public slots so that you can perform signal-slot connections with the widget in *Qt Designer*.

In the course of this chapter we will create a simple but useful widget, 'FileChooser', which we'll later make available in *Qt Designer* as a plugin. In practice most custom widgets are created to add functionality rather than to compose widgets, so we will create our widget in code rather than using *Qt Designer* to reflect this approach.

FileChooser consists of a **QLineEdit** and a **QPushButton**. The **QLineEdit** is used to hold a file or directory name, the **QPushButton** is used to launch a file dialog through which the user can choose a file or directory.



The FileChooser Custom Widget

If you've followed the manual up to this point you may well be able to create this custom widget yourself. If you're confident that you can make your own version of the widget, or have another widget that you want to turn into a plugin, skip ahead to Creating a Plugin. If you prefer to read how we created the widget then read on.

Coding the Widget's Interface

We will work step-by-step through the widget's header file, `qt/tools/designer/examples/filechooser/widget/filechooser`

```
#include <qwidget.h>

class QLineEdit;
class QPushButton;
```

Our widget will be derived from **QWidget** so we include the `QWidget.h` header file. We also forward declare the two classes that our widget will be built from.

We include the `Q_OBJECT` macro since this is required for classes that declare signals or slots. The `Q_ENUMS` declaration is used to register the `Mode` enumeration. Our widget has two properties, `mode`, to store whether the user should select a File or a Directory and `fileName` which stores the file or directory they chose.

```
class FileChooser : public QWidget
{
    Q_OBJECT

    Q_ENUMS( Mode )
    Q_PROPERTY( Mode mode READ mode WRITE setMode )
    Q_PROPERTY( QString fileName READ fileName WRITE setFileName )

public:
    FileChooser( QWidget *parent = 0, const char *name = 0);

    enum Mode { File, Directory };

    QString fileName() const;
    Mode mode() const;
```

The constructor is declared in the standard way for widgets. We declare two public functions, `fileName()` to return the filename, and `mode()` to return the mode.

```
public slots:
    void setFileName( const QString &fn );
    void setMode( Mode m );

signals:
    void fileNameChanged( const QString & );

private slots:
    void chooseFile();
```

The two 'set' functions are declared as public slots. `setFileName()` and `setMode()` set the filename and mode respectively. We declare a single signal, `fileNameChanged()`. The private slot, `chooseFile()` is called by the widget itself when its button is clicked.

```
private:
    QLineEdit *lineEdit;
    QPushButton *button;
    Mode md;

};
```

A pointer to **QLineEdit** and **QPushButton**, as well as a Mode variable are held as private data.

Coding the Implementation

We will work step-by-step through the implementation which is in `qt/tools/designer/examples/filechooser/widget/file`

```
FileChooser::FileChooser( QWidget *parent, const char *name )
    : QWidget( parent, name ), md( File )
{
```

The constructor passes the parent and name to its superclass, **QWidget**, and also initializes the private mode data, `md`, to File mode.

```
    QHBoxLayout *layout = new QHBoxLayout( this );
    layout->setMargin( 0 );

    lineEdit = new QLineEdit( this, "filechooser_lineedit" );
    layout->addWidget( lineEdit );
```

We begin by creating a horizontal box layout (**QHBoxLayout**) and add a **QLineEdit** and a **QPushButton** to it.

```
    connect( lineEdit, SIGNAL( textChanged( const QString & ) ),
            this, SIGNAL( fileNameChanged( const QString & ) ) );

    button = new QPushButton( "...", this, "filechooser_button" );
    button->setFixedWidth( button->fontMetrics().width( " ... " ) );
    layout->addWidget( button );

    connect( button, SIGNAL( clicked() ),
            this, SLOT( chooseFile() ) );
```

We connect the `lineEdit`'s `textChanged()` signal to the custom widget's `fileNameChanged()` signal. This ensures that if the user changes the text in the **QLineEdit** this fact will be propagated via the custom widget's own signal. The button's `clicked()` signal is connected to the custom widget's `chooseFile()` slot which invokes the appropriate dialog for the user to choose their file or directory.

```
        setFocusProxy( lineEdit );
    }
```

We set the `lineEdit` as the focus proxy for our custom widget. This means that when the widget is given focus the focus actually goes to the `lineEdit`.

```
void FileChooser::setFileName( const QString &fn )
{
```

```

        lineEdit->setText( fn );
    }

    QString FileChooser::fileName() const
    {
        return lineEdit->text();
    }

```

The `setFileName()` function sets the filename in the **QLineEdit**, and the `fileName()` function returns the filename from the **QLineEdit**. The `setMode()` and `mode()` functions (not shown) are similarly set and return the given mode.

```

void FileChooser::chooseFile()
{
    QString fn;
    if ( mode() == File )
        fn = QFileDialog::getOpenFileName( lineEdit->text(), QString::null, this );
    else
        fn = QFileDialog::getExistingDirectory( lineEdit->text(), this );

    if ( !fn.isEmpty() ) {
        lineEdit->setText( fn );
        emit fileNameChanged( fn );
    }
}

```

When `chooseFile()` is called it presents the user with a file or directory dialog depending on the mode. If the user chooses a file or directory the **QLineEdit** is updated with the chosen file or directory and the `fileNameChanged()` signal is emitted.

Although these two files complete the implementation of the FileChooser widget it is good practice to write a test harness to check that the widget behaves as expected before attempting to put it into a plugin.

Testing the Implementation

We present a rudimentary test harness which will allow us to run our custom widget. The test harness requires two files, a `main.cpp` to contain the FileChooser, and a `.pro` file to create the Makefile from. Here is `qt/tools/designer/examples/filechooser/widget/main.cpp`:

```

#include <qapplication.h>
#include "filechooser.h"

int main( int argc, char ** argv )
{
    QApplication a( argc, argv );
    FileChooser *fc = new FileChooser;
    fc->show();
    return a.exec();
}

```

And here is `qt/tools/designer/examples/filechooser/widget/filechooser.pro`

```

SOURCES += filechooser.cpp main.cpp
HEADERS += filechooser.h
TARGET    = filechooser
TEMPLATE  =app
CONFIG   += qt warn_on release

```

```
DBFILE = filechooser.db
PROJECTNAME = Filechooser
LANGUAGE = C++
```

We can create the makefile using `qmake`: `qmake -o Makefile filechooser.pro`, then we can make and run the harness to test our new widget. Once we're satisfied that the custom widget is robust and has the behaviour we require we can embed it into a plugin.

Creating a Plugin

Qt Plugins can be used to provide self-contained software components for Qt applications. Qt currently supports the creation of five kinds of plugins: codecs, image formats, database drivers, styles and custom widgets. In this section we will explain how to convert our `filechooser` custom widget into a *Qt Designer* custom widget plugin.

A *Qt Designer* custom widget plugin is always derived from **QWidgetPlugin**. The amount of code that needs to be written is minimal.

To make your own plugin it is probably easiest to start by copying our example `plugin.h` and `plugin.cpp` files and changing 'CustomWidgetPlugin' to the name you wish to use for your widget plugin implementation class. Below we provide an introduction to the header file although it needs no changes beyond class renaming. The implementation file requires simple changes, mostly more class renaming; we will review each function in turn and explain what you need to do.

The CustomWidgetPlugin Implementation

We have called our header file `plugin.h` and we've called our plugin class **CustomWidgetPlugin** since we will be using our plugin class to wrap our custom widgets. We present the entire header file to give you an impression of the scope of the implementation required. Most of the functions require just a few lines of code.

```
#include <qwidgetplugin.h>

class CustomWidgetPlugin : public QWidgetPlugin
{
public:
    CustomWidgetPlugin();

    QStringList keys() const;
    QWidget* create( const QString &classname, QWidget* parent = 0, const char* name = 0 );
    QString group( const QString& ) const;
    QIconSet iconSet( const QString& ) const;
    QString includeFile( const QString& ) const;
    QString tooltip( const QString& ) const;
    QString whatsThis( const QString& ) const;
    bool isContainer( const QString& ) const;
};
```

From qt/tools/designer/examples/filechooser/plugin/plugin.h

The QWidgetPlugin Functions

Create your own plugin `.cpp` file by copying our `plugin.cpp` file and changing all occurrences of 'CustomWidgetPlugin' to the name you wish to use for your widget plugin implementation. Most of the other changes are simply replacing the name of our custom control, 'FileChooser', with the name of your custom control. You may need to add extra `else if` clauses if you have more than one custom control in your plugin implementation.

We'll now look at the constructor.


```
CustomWidgetPlugin::CustomWidgetPlugin()
{
}
```

The constructor does not have to do anything. Simply copy ours with the class name you wish to use for your widget plugin implementation.

No destructor is necessary.

The keys function.

```
QStringList CustomWidgetPlugin::keys() const
{
    QStringList list;
    list << "FileChooser";
    return list;
}
```

For each widget class that you want to wrap in the plugin implementation you should supply a key, (often the class name), by which the class can be identified. In our example we add a single key, 'FileChooser'.

The create() function.

```
QWidget* CustomWidgetPlugin::create( const QString &key, QWidget* parent, const char* name )
{
    if ( key == "FileChooser" )
        return new FileChooser( parent, name );
    return 0;
}
```

In this function we create an instance of the requested class and return a QWidget pointer to the newly created widget. Copy this function changing the class name and the feature name and create an instance of your widget just as we've done here. (See the Qt Plugin documentation for more information.)

The includeFile() function.

```
QString CustomWidgetPlugin::includeFile( const QString& feature ) const
{
    if ( feature == "FileChooser" )
        return "filechooser.h";
    return QString::null;
}
```

This function returns the name of the include file for the custom widget. Copy this function changing the class name, key and include filename to suit your own custom widget.

The group(), iconSet(), tooltip() and whatsThis() functions.

```
QString CustomWidgetPlugin::group( const QString& feature ) const
{
    if ( feature == "FileChooser" )
        return "Input";
    return QString::null;
}

QIconSet CustomWidgetPlugin::iconSet( const QString& ) const
{
    return QIconSet( QPixmap( filechooser_pixmap ) );
}
```

```

QString CustomWidgetPlugin::includeFile( const QString& feature ) const
{
    if ( feature == "FileChooser" )
        return "filechooser.h";
    return QString::null;
}

QString CustomWidgetPlugin::toolTip( const QString& feature ) const
{
    if ( feature == "FileChooser" )
        return "File Chooser Widget";
    return QString::null;
}

QString CustomWidgetPlugin::whatsThis( const QString& feature ) const
{
    if ( feature == "FileChooser" )
        return "A widget to choose a file or directory";
    return QString::null;
}

```

We use the `group()` function to identify which *Qt Designer* toolbar group this custom widget should be part of. If we use a name that is not in use *Qt Designer* will create a new toolbar group with the given name. Copy this function, changing the class name, key and group name to suit your own widget plugin implementation.

The `iconSet()` function returns the pixmap to use in the toolbar to represent the custom widget. The `toolTip()` function returns the tooltip text and the `whatsThis()` function returns the What's This text. Copy each of these functions changing the class name, key and the string you return to suit your own widget plugin implementation.

The `isContainer()` function.

```

bool CustomWidgetPlugin::isContainer( const QString& ) const
{
    return FALSE;
}

```

Copy this function changing the class name to suit your widget plugin implementation. It should return `TRUE` if your custom widget can contain other widgets, e.g. like **QFrame**, or `FALSE` if it must not contain other widgets, e.g. like **QPushButton**.

The `Q_EXPORT_PLUGIN` macro.

```

Q_EXPORT_PLUGIN( CustomWidgetPlugin )

```

This macro identifies the module as a plugin — all the other code simply implements the relevant interface, i.e. wraps the classes you wish to make available.

This macro must appear once in your plugin. It should be copied with the class name changed to the name of your plugin's class. (See the Qt Plugin documentation for more information on the plugin entry point.)

Each widget you wrap in a widget plugin implementation becomes a class that the plugin implementation offers. There is no limit to the number of classes that you may include in an plugin implementation.

The Project File

The project file for a plugin is somewhat different from an application's project file but in most cases you can use our project file changing only the `HEADERS` and `SOURCES` lines.

```
SOURCES += plugin.cpp ../widget/filechooser.cpp
HEADERS += plugin.h ../widget/filechooser.h
DESTDIR = ../../../../plugins/designer
TARGET = filechooser

target.path= $$plugins.path
isEmpty(target.path):target.path= $$QT_PREFIX/plugins
INSTALLS += target
TEMPLATE = lib
CONFIG += qt warn_on_release plugin
INCLUDEPATH += $(QTDIR)/tools/designer/interfaces
DBFILE = plugin.db
PROJECTNAME = Plugin
LANGUAGE = C++
```

```
qt/tools/designer/examples/filechooser/plugin/plugin.pro
```

Change the HEADERS line to list your plugin's header file plus a header file for each of your widgets. Make the equivalent change for the SOURCES line. If you create a Makefile with `qmake` and make the project the plugin will be created and placed in a directory where *Qt Designer* can find it. The next time you run *Qt Designer* it will detect your new plugin and load it automatically, displaying its icon in the toolbar you specified.

Using the Widget Plugin

Once the plugin has been compiled it will automatically be found and loaded by *Qt Designer* the next time *Qt Designer* is run. Use your custom widget just like any other.

When you want to distribute your application, include the compiled plugin with the executable. Install the plugin in `$(QTDIR)/plugins/widgets`. If you don't want to use the standard plugin path, have your installation process determine the path you want to use for the plugin, and save the path, e.g. using `QSettings`, for the application to read when it runs. The application can then call `QApplication::addLibraryPath()` with this path and your plugins will be available to the application. Note that the final part of the path, i.e. `styles`, `widgets`, etc. cannot be changed.

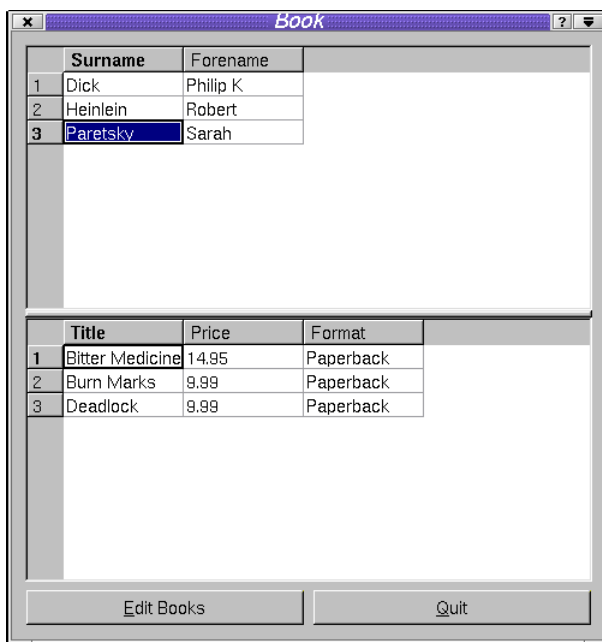
Creating Database Applications

This chapter shows you how to use Qt's data-aware widgets from within *Qt Designer*. It demonstrates INSERT, UPDATE and DELETE in both **QDataTables** (tables) and **QDataBrowsers** (forms). It also shows how to code Master-Detail relationships and Drilldown. A simple approach to foreign key handling is presented here; a more sophisticated approach is shown in the online SQL module documentation.

If you wish to run the examples or create your own applications using these widgets you need access to an SQL database and a Qt database driver that can connect to the database. At the time of writing the drivers that Qt supports are QODBC3 (Open Database Connectivity), QOCI8 (Oracle), QPSQL7 (PostgreSQL 6 and 7) and QMYSQL3 (MySQL).

Although you can use the Qt data-aware widgets to browse and edit data in SQL databases without having to write any SQL, a basic understanding of SQL is highly recommended. We assume that you have some familiarity with SELECT, INSERT, UPDATE and DELETE statements. We also assume a basic understanding of the concepts of normalisation and of primary and foreign keys. A standard text covering SQL databases is *An Introduction to Database Systems (7th ed.)* by C. J. Date, ISBN 0201385902.

In the following text we describe the creation of a 'book' database application. The application demonstrates how to use **QDataTables** including in-place record editing and how to set up master-detail relationships between **QDataTables**. It also explains how to drill down from a **QDataTable** to another widget, for example, to a **QDataBrowser** or a **QDataView** and how to perform record editing in a **QDataBrowser**. A great deal of functionality is available from the classes directly in *Qt Designer* although subclassing is always available for finer control. If you want to build the 'book' examples you will need to create the example schema on your database.



The Book Application

The Example Schema

Note that the examples in this chapter all use the tables, views and records which are defined in the `qt/tools/designer/examples/book/book.sql` file. This file has been tested with PostgreSQL 6 and PostgreSQL 7. You may need to modify the SQL in this file to recreate the example database on your own system.

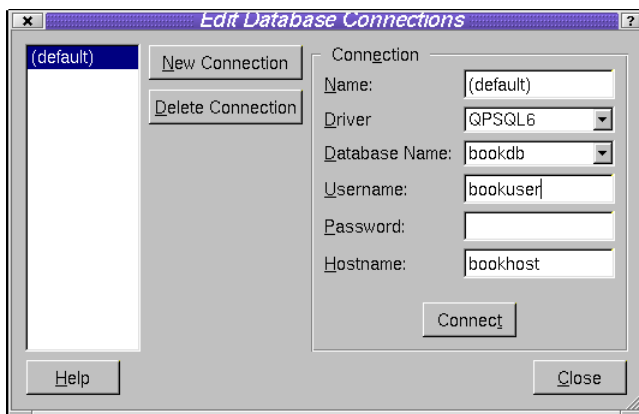
Schema CREATE TABLE Statements

The 'book' table is simplified for the purposes of the example. It can only relate a book to a single author (authorid) and lacks an ISBN field. The 'sequence' table is used for generating unique index values for the example tables. Note that SQL databases often provide their own method for creating sequences (for example, using the `CREATE SEQUENCE` command) which is very likely to be a more optimal solution. For the sake of portability the examples will use a 'sequence' table which will work with the vast majority of SQL databases.

Setting Up Database Connections

There are two aspects of database connections that we must consider. Firstly the connection we wish to use within *Qt Designer* itself, and secondly the connection we wish to use in the applications that we create.

Setting Up Qt Designer's Connections



Database Connections Dialog

Choose **Project|Database Connections** from the menu bar. The *Database Connections* dialog will appear. Click **New Connection**. For applications that use a single database it will probably be most convenient to use the default connection name of '(default)'. If you use more than one database then each one must be given a unique name. A driver must be chosen from the Driver combo box. The database name may be available in the Database Name combo box or may have to be typed in. The database name, username, password and hostname should be provided by your database system administrator. When the Connection information has been completed click **Connect**. If the connection is made the connection name will appear in the list box on the left hand side of the dialog. You can now close the dialog; the connection settings will remain in effect until you change or delete them or exit from *Qt Designer*.

Qt Designer can remember database connection settings in `qmake` project files. Create a new project, e.g. click **File|New**, then click the 'C++ Project' icon to invoke the *Project Settings* dialog. Click the ellipsis button to invoke the *Save As* dialog; navigate to the project's directory (creating it if necessary). Make sure you're in the project's directory, then enter a project name of 'book.pro'. Click the **Save** button to return to the *Project Settings* dialog,

then click **OK**. Next time you start *Qt Designer* instead of opening individual `.ui` files open the `.pro` project file instead and *Qt Designer* will automatically reload the project's connection settings. To activate the connection click **Project|Database Connections**. The connections previously saved with the project will be listed in the left hand list box. Click the connection you wish to use and then click **Connect**. This connection will be used from now on, e.g. for previewing **QDataTables**. Opening a project file also causes *Qt Designer* to load in the list of forms associated with the project into the Form List window. In most of the explanation that follows we will assume that you use project files and have clicked **Connect** so that there is always a connection available when you work in *Qt Designer*.

Setting Up Connections for Applications

The applications you create must make their own connections to the SQL database. `createConnections()` function

```
bool createConnections()
{
    // create the default database connection
    QSqlDatabase *defaultDB = QSqlDatabase::addDatabase( "QPSQL7" );
    if ( ! defaultDB ) {
        qWarning( "Failed to connect to driver" );
        return FALSE;
    }
    defaultDB->setDatabaseName( "book" );
    defaultDB->setUserName( "bookuser" );
    defaultDB->setPassword( "bookpw" );
    defaultDB->setHostName( "bookhost" );
    if ( ! defaultDB->open() ) {
        qWarning( "Failed to open books database: " +
                 defaultDB->lastError().driverText() );
        qWarning( defaultDB->lastError().databaseText() );
        return FALSE;
    }

    return TRUE;
}
```

We call `addDatabase()` passing it the name of the driver we wish to use. We then set the connection information by calling the `set...` functions. Finally we attempt to open the connection. If we succeed we return `TRUE`, otherwise we output some error information and return `FALSE`. From `qt/tools/designer/examples/book/book1/main.cpp`

```
int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

    if ( ! createConnections() )
        return 1;

    BookForm bookForm;
    app.setMainWidget( &bookForm );
    bookForm.show();

    return app.exec();
}
```

All the examples presented in this chapter call `createConnections()` after creating the **QApplication** object in their `main.cpp` file and make use of the default connection. If you need to connect to multiple databases use the two-argument form of `addDatabase()`, passing it both the name of the driver and a unique identifier. This is explained further in the Qt SQL Module documentation.

You do not need to keep a reference to database connections. If you use a single database connection, this becomes the default connection and database functions will use this connection automatically. We can always get a pointer to any of our connections by calling `QSqlDatabase::database()`.

If you create a `main.cpp` file using *Qt Designer*, this file will *not* include `createConnections()`. We do not include this function because it needs the username and password for the database connection, and you may prefer to handle these differently from our simple example function. As a result, applications that preview correctly in *Qt Designer* will not run unless you implement your own database connections function.

Using QDataTable

QDataTables may be placed on any form to provide browsing of database tables and views. **QDataTables** can also be used to update or delete records in-place, i.e. inside the cells themselves. Inserting records via a **QDataTable** usually requires connecting to the `primeInsert()` signal, so that we can generate primary keys for example, or provide default values. If we wish to present records using a form view (perhaps combining data from several tables and views) we might use several **QDataBrowsers** and **QDataViews**.

Quickly Viewing a Database Table

This example, along with all the other examples in this chapter, has the project name 'book' and uses the database created by the `book.sql` script. As we work through the chapter we will build the 'book' application step by step. Create or copy the `qt/tools/designer/examples/book/book1/main.cpp` file shown earlier. The project file for this first example is `qt/tools/designer/examples/book/book1/book.pro`. Start a new project by clicking **File|New**, then click the 'C++ Project' icon to invoke the *Project Settings* dialog. Click the ellipsis button to invoke the *Save As* dialog; navigate to the project's directory (creating it if necessary). Make sure you're in the project's directory, then enter a project name of 'book.pro'. Click the **Save** button to return to the *Project Settings* dialog, then click **OK**. Now click **Project|Database Connections**. Fill in the connection information appropriate to your database then press **Connect**. The connection name should now appear in the left hand list box. (If this doesn't happen you'll need to contact your database systems administrator for help.) Close the dialog.

We will now create a new form with a **QDataTable** that's connected to one of our database tables.

Click **File|New**. The *New File* dialog presents us with a number of form templates to choose from. Choose the 'Dialog' form and click **OK**. Now click **File|Save**. You will be prompted for a filename, call it `book.ui`.

Setting up a QDataTable

To place a **QDataTable** widget on the form either click **Tools|Views|QDataTable** or click the **QDataTable** toolbar button. Click on the form and the SQL Table Wizard will appear.

1. The *Database Connection and Table* wizard page is used to set up a connection if one doesn't exist and to choose the table or view for the **QDataTable**. (See *Setting Up Qt Designer's Connections*.)

Click the connection you wish to use, listed in the left hand list box, e.g. "(default)". The available tables and views will appear in the right hand Table list box. Click the 'author' table and then click the **Next** button.

2. The *Displayed Fields* wizard page provides a means of selecting which fields should be displayed in the **QDataTable** and in what order. By default all fields except the primary key (if there is one) are in the Displayed Fields list box. The left- and right-pointing blue arrow buttons can be used to move fields between the Displayed Fields and the Available Fields list boxes. The blue up and down pointing arrow buttons are used to select the display order of the displayed fields.

The default settings are the ones we want so simply click **Next**.

3. The *Table Properties* wizard page provides convenient access to some of the database-related properties of the **QDataTable**.

Make sure the Confirm Deletes checkbox is checked, then click **Next**.

- The *SQL* wizard page is used to set the **QDataTable**'s Filter and Sort properties. The Filter is an SQL *WHERE* clause (without the word 'WHERE'). For example, to only list authors whose surnames begin with 'P', we would enter `title LIKE 'P%'`. We'll leave the filter empty. The Available Fields list box lists all the fields. The Sort By list box lists the fields that the **QDataTable** is to sort by and the direction of their sorting (ASCending or DESCending). The left and right blue arrows are used to move fields between the two list boxes. The up and down blue arrows move fields up and down within the Sort By list box. The ASC or DESC setting is changed with the 'sort order' toolbar button.

Move the surname and forename fields into the Sort By list box and click **Next**.

- The *Finish* wizard page gives us the opportunity to go back and change any of our settings. We will be able to change them later through the **QDataTable**'s properties so we can finish with the wizard.

Click **Finish**.

The table will appear on the form with each column labelled with a default column name. If you wish to change the settings then most of them are available in the property window. The display names, the fields they are based upon, and the order of appearance of the columns can be changed using the *Edit Table* dialog (explained later) by right clicking the **QDataTable** and left clicking **Edit**.

Laying out the Form

Click on the form and click the **Lay Out Vertically** toolbar button. Now click **Preview|Preview Form**; the form will run and the table will automatically display all the records.

To turn the form we've created into an executable application we must add the `main.cpp` file to the project file and make the project. We should also do some renaming to make things easier to understand.

- Click on the form and change its name to 'BookForm' and its caption to 'Book'. Click on the **QDataTable** and change its name to 'AuthorDataTable'.
- Click **File|Save All**.
- Open the project file, e.g. `book.pro`, in a plain text editor and add the line: `SOURCES += main.cpp` at the end of the file.
- Run `qmake` to generate the make file, e.g. `qmake -o Makefile book.pro`, then make and run the book program.

This example shows how easy it is to use **QDataTable** to show the contents of a database table or view. You can use the application we've just built to update and delete author records. In the examples that follow we will cover insertions, setting up master-detail relationships, drilldown and foreign key lookups.

A Note on Foreign Keys

In most relational databases tables contain fields which are foreign keys into other tables. In our 'book' database example the `authorid` in the `book` table is a foreign key into the `author` table. When we present a form to the end user we do not usually want the foreign key itself to be visible but rather the text associated with it. Thus, we would want the author's name to appear rather than the author id when we show book information. In many databases, this can be achieved by using a view. See your database's documentation for details.

Inserting Records in QDataTables

Record insertion into a relational database usually requires the generation of a primary key value which uniquely identifies the record in the table. Also we often want to create default values for some fields to minimize the user's work. We will create a slot to capture the **QDataTables** `primeInsert()` signal and populate the **QSqlRecord** insertion buffer with a unique primary key.

- Click **Edit|Slots** to invoke the *Edit Slots* dialog. Click **New Slot**, then enter the slot name `primeInsertAuthor(QSqlRecord*)` into the Slot Properties' Slot line edit box. Click **OK**.

2. Click the **Connect Signals/Slots** toolbar button, then click the AuthorDataTable, drag to the form and release the mouse. The *Edit Connections* dialog will now appear. Click the `primeInsert()` signal and then the `primeInsertAuthor()` slot to make the connection. Now click **OK**.
3. Click the Source tab of the Object Hierarchy window (click **Window|Object Hierarchy** to make the window visible if necessary). Click the `primeInsertAuthor()` slot and an editor window will appear.
4. We must change the `BookForm::primeInsertAuthor()` slot to specify the parameter name and perform the necessary action:

```
void BookForm::primeInsertAuthor( QSqlRecord * buffer )
{
    QSqlQuery query;
    query.exec( "UPDATE sequence SET sequence = sequence + 1 WHERE tablename='author';" );
    query.exec( "SELECT sequence FROM sequence WHERE tablename='author';" );
    if ( query.next() ) {
        buffer->setValue( "id", query.value( 0 ) );
    }
}
```

A **QSqlQuery** object is used to increment and retrieve a unique 'sequence' number for the author table. The signal passed us a pointer to the insertion buffer and we then put the value we've retrieved, i.e. the next sequence number, into the buffer's id field. (Again, note that SQL databases often support a native 'sequence' function. The method used here is inappropriate for production systems, and is for example purposes only. See your database's documentation for details on how to generate unique keys in code. In many cases, the database can generate them automatically, or the database may provide a special syntax for dealing with sequences.)

If we rebuild the application it will now support `INSERT` as well as `UPDATE` and `DELETE`. We could easily have added additional code to insert default values, e.g. today's date into a date field, if necessary.

Browsing is supported by clicking records and by using the arrow keys. Once a record is active (highlighted) we can edit the it. Press the **Insert** key to `INSERT` a new record; press **F2** to `UPDATE` the current record; press the **Del** key to `DELETE` the current record. All these operations take place immediately. Users can be given the opportunity to confirm their edits by setting the **QDataTable**'s `confirmEdits` property to `True`. If the `confirmEdits` property is `True` then user confirmation will be required for all insertions, updates and deletes. For finer control you can set the `confirmInsert`, `confirmUpdate` and `confirmDelete` properties individually.

QDataTable User Interface Interaction

The default user-interface behaviour for **QDataTables** is as follows:

- Users can move to records by clicking the scrollbar and clicking records with the mouse. They can also use the keyboard's navigation keys, e.g. **Left Arrow**, **Right Arrow**, **Up Arrow**, **Down Arrow**, **Page Up**, **Page Down**, **Home** and **End**.
- `INSERT` is initiated by right-clicking the record and clicking `Insert` or by pressing the **Ins** (Insert) key. The user moves between fields using **Tab** and **Shift+Tab**. The `INSERT` will take place if the user presses **Enter** or **Tab**s off the last field. If `autoEdit` is `TRUE` the insert will take place if the user navigates to another record. `INSERT` is cancelled by pressing **Esc** (Escape). If `autoEdit` is `FALSE` navigating to another record also cancels the `INSERT`. Setting `confirmInsert` to `TRUE` will require the user to confirm each `INSERT`.
- `UPDATE` is initiated by right-clicking the record and clicking `Update` or by pressing **F2**. The update will take place if the user presses **Enter** or **Tab**s off the last field. If `autoEdit` is `TRUE` the update will take place if the user navigates to another record. `UPDATE` is cancelled by pressing **Esc**. If `autoEdit` is `FALSE` navigating to another record also cancels the `UPDATE`. Setting `confirmUpdate` to `TRUE` will require the user to confirm each `UPDATE`.
- `DELETE` is achieved by right-clicking the record and clicking `Delete` or by pressing the **Del** (Delete) key. Setting `confirmDelete` to `TRUE` will require the user to confirm each `DELETE`.

You can change this default behaviour programmatically if required.

Relating Two Tables Together (Master-Detail)

Databases often have pairs of tables that are related. For example, an invoice table might list the numbers, dates and customers for invoices, but not the actual invoice items, which an invoice item table might store. In the 'book' application we wish to have a **QDataTable** that we can use to browse through the authors table and a second **QDataTable** to show the books they've written.

Open the book project if it isn't already open *Qt Designer*. We will modify this project to show two **QDataTables** that relate the author table to the book table.

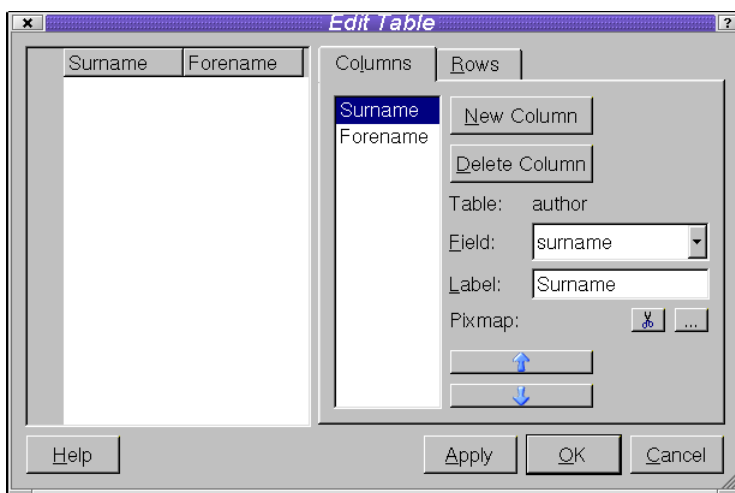
1. Click the author **QDataTable** and then click the **Break Layout** toolbar button.
2. Resize the **QDataTable** so that it only occupies the top half of the form.
3. Now click on the **QDataTable** toolbar button and click on the bottom half of the form. The SQL Table Wizard will appear. (This Wizard is explained in Quickly Viewing a Database Table.)
 - (a) Click the connection you're using and click the book table. Click the **Next** button.
 - (b) Since we do not want them visible, make sure the authorid and id fields are moved to the Available Fields list box by using the arrow buttons. Move the title field to the top of the Displayed Fields, and move the price field above the notes field. Click the **Next** button.
 - (c) On the Table Properties page click the Read Only checkbox then click the **Next** button.
 - (d) On the SQL page we will leave the Filter (WHERE clause) empty. Move the title field to the Sort By list box and click **Next**. Now click **Finish**.
 - (e) Change this **QDataTable**'s name to "BookDataTable".

Shift+Click the top **QDataTable** so that both **QDataTables** are selected and then click the **Lay Out Vertically (in Splitter)** toolbar button.

4. Click on the form and click the **Lay Out Vertically** toolbar button.

Run the form by clicking **Preview | Preview Form**. All the authors are displayed in the top **QDataTable** and all the books are displayed in the bottom **QDataTable**. However we only want the books of the currently selected author showing in the bottom **QDataTable**. We will deal with this by filtering the records in the book table according to the author selected in the author table.

Using the Table Editor



Edit Table Dialog

QDataTables are created and set up using the SQL Table Wizard. Like any other *Qt Designer* widget their properties may be changed in the Properties window. Some of the column and row based properties can also be changed using the *Edit Table* dialog. This dialog is invoked by right clicking the **QDataTable** and left clicking the **Edit** menu item. The right hand half of the *Edit Table* dialog is where we choose the fields we wish to display, their order and their labels. The procedure for creating columns is as follows:

1. Click the **New Column** button.
2. Drop down the Field combobox to list the available fields.
3. Click the field you wish to include at this point.
4. *Optionally* edit the Label if the default isn't appropriate.
5. *Optionally* click the Pixmap ellipsis (...) button to choose a pixmap to be displayed to the left of the column's label. (The ellipsis button appears when you click in the Value part of the Properties list by a *pixmap* or *iconSet* property.)

Repeat the steps listed above for each column you wish to add. Once all the fields have been added you can change their ordering by using the blue up and down arrow buttons. At any point you can press **Apply** to see how the table will look. Finally click the **OK** button to save the properties you have set. You can always return to the table editor to change these settings later.

Filtering One QDataTable by Another

To filter the book table's records we need to capture the author **QDataTable**'s `currentChanged()` signal and change the **BookDataTable**'s filter accordingly.

1. Click **Edit|Slots**. In the *Edit Slots* dialog click **New Slot** and enter a slot name of `newCurrentAuthor(QSqlRecord*)`. Click **OK**.
2. Click **Connect Signals/Slots**, then click the **AuthorDataTable QDataTable** and drag to the form; release the mouse on the form. The *Edit Connections* dialog will appear. Click the `currentChanged()` signal and the `newCurrentAuthor` slot. Click **OK**.
3. Click the Source tab of the Object Hierarchy window (click **Window|Object Hierarchy** to make the window visible if necessary). Click the `newCurrentAuthor()` slot and an editor window will appear.
4. We must change the `BookForm::newCurrentAuthor()` slot to specify the parameter name and perform the necessary action:

```
void BookForm::newCurrentAuthor( QSqlRecord *author )
{
    BookDataTable->setFilter( "authorid=" + author->value( "id" ).toString() );
    BookDataTable->refresh();
}
```

All that's required now is to change the **BookDataTable**'s filter and refresh the **QDataTable** to show the results of the filter.

Preparing the Interface for Drilldown

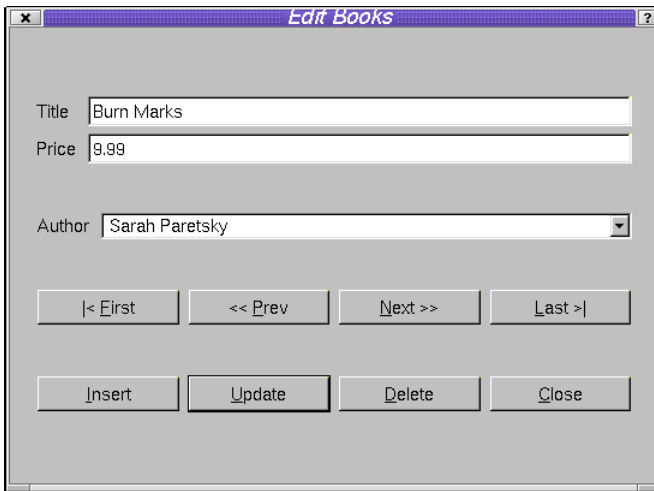
We can now browse and edit authors and see their books in the **BookDataTable**. In the next section we explore **QDataBrowser**, which will allow us to drill down to a dialog through which we can edit books. For now we will add some buttons to the main **BookForm** which we will use to invoke the book editing dialog.

1. Click the form, then click the **Break Layout** toolbar button. Resize the form to make room for some buttons at the bottom.
2. Add two buttons to the bottom of the form. Change their names and labels to the following:
 - `EditPushButton` — `&Edit Books`
 - `QuitPushButton` — `&Quit`

Hold down the Shift key and Click both buttons (i.e. **Shift+Click** the buttons) and click the **Lay Out Horizontally** toolbar button. Click the form and click the **Lay Out Vertically** toolbar button.

3. We will provide the Quit button with functionality now and work on the rest shortly. Click **Connect Signals/Slots**, then click the Quit button and drag to the form; release the mouse on the form. The *Edit Connections* dialog will appear. Click the `clicked()` signal and the `accept()` slot. Click **OK**.

Using QDataBrowser and QDataView



The Book Application's Edit Books Dialog

Drilling Down to a Form using QDataBrowser

Setting up a QDataBrowser

We will now create a new form to allow users to edit book records. Click the **New** toolbar button, click the Dialog template from the *New File* dialog and click **OK**. Change the name of the form to `EditBookForm` and its caption to 'Edit Books'. Click the **Save** toolbar button and call the file `editbook.ui`. Now that we have the form we can add a **QDataBrowser** to show the book records.

1. Click the **Data Browser** toolbar button, then click the form. The Data Browser Wizard will appear.
2. The *Database Connection and Table* wizard page is used to set up a connection if one doesn't exist and to choose the table or view for the **QDataBrowser**. (See Setting Up Qt Designer's Connections.)

Click the connection you wish to use, listed in the Connection list box, e.g. "(default)". The available tables and views will appear in the Table list box. Click the book table and then click the **Next** button.

3. The *Displayed Fields* wizard page provides a means of selecting which fields should be displayed in the **QDataBrowser** and in what order. By default all fields except the primary key (if there is one) are in the right hand Displayed Fields list box. The left and right blue arrow buttons can be used to move fields between the Displayed Fields and the Available Fields list boxes. The blue up and down arrow buttons are used to select the display order of the displayed fields.

We don't want to see the authorid foreign key field on the form, so move it to the Available Fields list box. Also, move the title field to the top of the Displayed Fields list. Click the **Next** button.

4. The *Navigation and Editing* wizard page allows us to choose which navigation and editing buttons should appear on the form.

We will accept the defaults and simply click the **Next** button.

5. The *SQL* wizard page is used to set the **QDataBrowser's** Filter and Sort properties. The Filter is an SQL `WHERE` clause (without the word 'WHERE'). For example, to only list books that cost less than 50 (of some currency, e.g. dollars), we would enter `price < 50`. We will leave the filter empty. The Available Fields list box lists all the fields. The Sort By list box lists the fields that the **QDataBrowser** is to sort by and the direction of their sorting (ASCending or DESCending). The left and right blue arrows are used to move fields between the two list boxes. The up and down blue arrows move fields up and down within the Sort By list box. The ASC or DESC setting is changed with the sort order button.

Move the title field into the Sort By list box and click **Next**.

6. The *Layout* wizard page is used to specify the initial layout of the form. Change the Number of Columns to 1, then click **Next**. Now click **Finish**.
7. The **QDataBrowser** will now appear on the form. Resize the form to make it shorter. Click the **QDataBrowser** then click the **Break Layout** toolbar button. Click the buttons then click the **Break Layout** toolbar button. Add another button called 'PushButtonClose' with the text '&Close' and place it to the right of the Delete button.
8. **Shift+Click** the Insert, Update, Delete and Close buttons, then click the **Lay Out Horizontally** toolbar button. Click the **QDataBrowser**, then click the **Lay Out in a Grid** toolbar button. Finally click the form and click the **Lay Out Vertically** toolbar button. Now click the **QDataBrowser** and rename it 'BookDataBrowser'.
9. *Qt Designer* will generate the necessary code to make the browser operational (including generating the appropriate cursor, sort and filter code).

For finer control over the form, we will be creating our own database cursor. Therefore, set the BookDataBrowser's frameworkCode property to FALSE in the Properties window to prevent *Qt Designer* from generating redundant code for the cursor.

QDataBrowser User Interface Interaction

The user-interface behaviour for **QDataBrowsers** is created by connecting slots and signals. The slots provided are:

- insert(), update() and del() for editing;
- first(), next(), prev(), and last() for navigation;
- refresh() to refresh the cursor from the database;
- readFields() to read data from the cursor's edit buffer and writeFields() to write the form's data to the cursor's edit buffer;
- clearValues() to clear the form's values.

If you use *Qt Designer's* **QDataBrowser** wizard you will be given the option of creating a default set of buttons for navigation and editing. The behaviour of these buttons is set up using the slots described above to provide the following functionality:

- INSERT is initiated by pressing the **Ins** (Insert) button. The user moves between fields using **Tab** and **Shift+Tab**. If the user presses the Update button the INSERT will take place and the user will be taken to the record they have just inserted. If the user presses the Insert button (i.e. a second time) the INSERT will take place and a new insertion will be initiated. If autoEdit is TRUE the INSERT will take place if the user navigates to another record. INSERT is cancelled by pressing the **Esc** key or by pressing the **Del** (Delete) button. If autoEdit is FALSE then navigating to another record also cancels the INSERT. Setting confirmInsert to TRUE will require the user to confirm each INSERT.
- UPDATE is automatically initiated whenever the user navigates to a record. An update will take place if the user presses the Update button. If autoEdit is TRUE the update will take place if the user navigates to another record. UPDATE is cancelled by pressing the **Esc** key or by pressing the **Del** button. If autoEdit is FALSE then navigating to another record also cancels the UPDATE. Setting confirmUpdate to TRUE will require the user to confirm each UPDATE.
- DELETE is achieved by pressing the **Del** button. Setting confirmDelete to TRUE will require the user to confirm each DELETE.

Performing the Drilldown

We now have a working form for editing book records. We need to start the form when the user clicks our 'Edit Books' button, and to navigate to the record they have selected in the BookDataTable. We also need to provide a means of editing the foreign keys, e.g. authorid.

1. We need to make a new slot to connect the Edit Books' button's clicked() signal to. Click on the Book form to make it *Qt Designer's* active form. Invoke the *Edit Slots* dialog and create a new slot called

editClicked(). Now click the **Connect Signals/Slots** toolbar button. Click the Edit Books button and drag to the form; release the mouse on the form. In the *Edit Connections* dialog connect the clicked() signal to the editClicked() slot. Click **OK** to leave the dialog.

2. In the Object Hierarchy window click Source and then click the editClicked function. We need to change it to the following:

```
void BookForm::editClicked()
{
    EditBookForm *dialog = new EditBookForm( this, "Edit Book Form", TRUE );
    QSqlCursor cur( "book" );
    dialog->BookDataBrowser->setSqlCursor( &cur );
    dialog->BookDataBrowser->setFilter( BookDataTable->filter() );
    dialog->BookDataBrowser->setSort(QSqlIndex::fromStringList(
        BookDataTable->sort(), &cur ) );
    dialog->BookDataBrowser->refresh();
    int i = BookDataTable->currentRow();
    if ( i == -1 ) i = 0; // Always use the first row
    dialog->BookDataBrowser->seek( i );
    dialog->exec();
    delete dialog;
    BookDataTable->refresh();
}
```

We create our dialog as before. We also create a cursor over the book table and set the dialog's **QDataBrowser**, BookDataBrowser, to use this new cursor. We set the **QDataBrowser**'s filter and sort to those that applied to the main form's book **QDataTable**. We refresh the **QDataBrowser** and seek to the same record the user was viewing on the main form. Then we exec the dialog and delete it when the user has finished with it. Finally we update the BookDataTable in the main form to reflect any changes that were made in the dialog.

3. Because our code refers to a class declared in editbook.h and to a **QDataBrowser** we need to add two additional include files. Click on the BookForm, then click on the Source tab of the Object Hierarchy window. Right click the 'Includes (In Declaration)' item and click New. Type in "editbook.h". Now add a second include, this time, <qdatabrowser.h>.

Now when we navigate through the author and book records in the BookForm we can click the Edit Books button to launch our Edit Books dialog. Although the dialog supports UPDATE, DELETE and navigation over the book table, we cannot edit the foreign keys nor perform inserts. We will deal with insertion in the same way as we did with the **QDataTable**, then we will handle the foreign key relationship to author.

Inserting into a QDataBrowser

We will create a slot to receive the Edit Books form's primeInsert() signal so that we can insert a unique primary key.

1. Click on the Edit Books form, then create a new Slot called primeInsertBook(QSqlRecord*).
- Click **Edit|Slots**, then click the **New Slot** button and type the new slot name in the Slot Properties Slot edit box. Click **OK**.
2. Connect the BookDataBrowser's primeInsert() signal to the primeInsertBook() slot.
- Click the **Connect Signals/Slots** toolbar button, then click the BookDataBrowser and drag to the form; release the mouse on the form. Now click the primeInsert() signal and the primeInsertBook slot. Click **OK**.
3. In the Object Hierarchy window click Source and then click the primeInsertBook slot. We need to change it to the following:

```
void EditBookForm::primeInsertBook( QSqlRecord * buffer )
```

```

    {
        QSqlQuery query;
        query.exec( "UPDATE sequence SET sequence = sequence + 1 WHERE tablename='book';" );
        query.exec( "SELECT sequence FROM sequence WHERE tablename='book';" );
        if ( query.next() ) {
            buffer->setValue( "id", query.value( 0 ) );
        }
    }
}

```

We will also tidy up the user interface slightly. Click the Update button and set its default property to True. Connect the Close button's `clicked()` signal to the `EditBookForm`'s `accept()` slot.

Handling Foreign Keys in a QDataBrowser

Qt's SQL module provides two approaches to dealing with foreign keys. The most powerful and flexible is to subclass widgets and use property maps to relate the widgets to the database. This approach is described in the Qt SQL Module documentation, particularly the `StatusPicker` example. A simpler approach that can be taken wholly within *Qt Designer* is presented here.

We will add a new field to the `EditBookForm` so that authors can be edited along with the title and price. Once we've handled the visual design we'll write the code to make it all work.

4. First we'll add the new widgets. Click the `BookDataBrowser` and click the **Break Layout** toolbar button. Resize the form to make it larger and drag each set of buttons down to make some room below the title and price `QLineEdit`s. Click the **Text Label** toolbar button and click on the form beneath the Price label. Click the *Text Label* and change its text to 'Author'. Click the **ComboBox** toolbar button and click on the form beneath the price `QLineEdit`. In the Property Window change the *ComboBox*'s *name* to `ComboBoxAuthor` and change its *sizePolicy hSizePolicy* to Expanding.
2. Now we'll lay out the dialog. **Shift+Click** the Author label and the *ComboBox* then click the **Lay Out Horizontally** toolbar button. Now click the `BookDataBrowser` and click the **Lay Out in a Grid** toolbar button.

We need to write some code so that the *ComboBox* will be populated with author names and scroll to the current book's author. We also need to ensure that we put the author's id into the book table's `authorid` field when a book record is inserted or updated. We'll ensure the code is executed at the right time by putting it in slots and connecting signals to our slots.

1. Create two new slots called `beforeUpdateBook(QSqlRecord *buffer)` and `primeUpdateBook(QSqlRecord *buffer)`. (Click **Edit|Slots**, then in the *Edit Slots* dialog click New Slot and enter the first new slot. Click New Slot again and enter the second slot then click **OK**.)
2. When the user navigates through the dialog, each time they move to a new record, a `primeUpdate()` signal is emitted. We connect to this so that we can update the *ComboBox*'s display. Just before a record is updated or inserted into the database a `beforeUpdate()` or `beforeInsert()` signal is emitted. We connect our `beforeUpdateBook()` slot to both these signals so that we can ensure that the book's `authorid` field is correctly populated.

Click the `BookDataBrowser` and drag the mouse to the form; release the mouse and the *Edit Connections* dialog will appear. Connect the `beforeUpdate()` signal to our `beforeUpdateBook()` slot. Connect the `beforeInsert()` signal to our `beforeUpdateBook()` slot. Finally connect the `primeUpdate()` signal to our `primeUpdateBook()` slot.

3. All that remains is to write the underlying code. All the code snippets are taken from `qt/tools/designer/examples/book/book7/editbook.ui`.

(a) We start with the `init()` function; this is called after the dialog is constructed and we will use it to populate the *ComboBox* with author names.

```

void EditBookForm::init()
{

```

```

        QSqlQuery query( "SELECT surname FROM author ORDER BY surname;" );
        while ( query.next() )
            ComboBoxAuthor->insertItem( query.value( 0 ).toString() );
    }

```

Here we execute a query to get a list of author names and insert each one into the *ComboBox*.

- (b) We next write the code which will be executed just before a record is updated (or inserted) in the database.

```

void EditBookForm::beforeUpdateBook( QSqlRecord * buffer )
{
    QSqlQuery query( "SELECT id FROM author WHERE surname =" +
        ComboBoxAuthor->currentText() + "';" );
    if ( query.next() )
        buffer->setValue( "authorid", query.value( 0 ) );
}

```

We look up the id of the *ComboBox*'s current author and place it in the update (or insert) buffer's authorid field.

- (c) As the user navigates through the records we ensure that the *ComboBox* reflects the current author.

```

void EditBookForm::primeUpdateBook( QSqlRecord * buffer )
{
    // Who is this book's author?
    QSqlQuery query( "SELECT surname FROM author WHERE id='" +
        buffer->value( "authorid" ).toString() + "';" );
    QString author = "";
    if ( query.next() )
        author = query.value( 0 ).toString();
    // Set the ComboBox to the right author
    for ( int i = 0; i count(); i++ ) {
        if ( ComboBoxAuthor->text( i ) == author ) {
            ComboBoxAuthor->setCurrentItem( i );
            break;
        }
    }
}

```

Firstly we look up the book's author and secondly we iterate through the *ComboBox*'s items until we find the author and set the *ComboBox*'s current item to the matching author.

If the author name has changed or been deleted the query will fail and no author id will be inserted into the buffer causing the INSERT to fail. An alternative is to record the author id's as we populate the *ComboBox* and store them in a **QMap** which we can then look up as required. This approach requires changes to the `init()`, `beforeUpdateBook()` and `primeInsertBook()` functions and the addition of a new function, `mapAuthor()`. The relevant code from `qt/tools/designer/examples/book/book8/editbook.ui` is shown below.

1. First we need to create a class variable to map author names to author id's. Click in the Source tab of the Object Hierarchy, then right click the Class Variables item and click **New**. Type in '`QMap<QString,int> authorMap;`'.
2. We now record the author id's in the `init()` function.

```

void EditBookForm::init()
{
    QSqlQuery query( "SELECT surname, id FROM author ORDER BY surname;" );
    while ( query.next() ) {
        ComboBoxAuthor->insertItem( query.value( 0 ).toString() );
        int id = query.value( 1 ).toInt();
        mapAuthor( query.value( 0 ).toString(), id, TRUE );
    }
}

```


After inserting each author's name into the *ComboBox* we populate a **QMap** with the author's name and id.

3. Instead of looking up the author's id in the database we look it up in the **QMap**.

```
void EditBookForm::beforeUpdateBook( QSqlRecord * buffer )
{
    int id;
    mapAuthor( ComboBoxAuthor->currentText(), id, FALSE );
    buffer->setValue( "authorid", id );
}
```

We use a single function for storing author id's and returning them so that we can use a static data structure.

4.

```
void EditBookForm::mapAuthor( const QString & name, int & id, bool populate )
{
    if ( populate )
        authorMap[ name ] = id;
    else
        id = authorMap[ name ];
}
```

If the populate flag is TRUE, we store the author's name and id in the **QMap**, otherwise we look up the given author name and set id appropriately.

Another approach which is especially useful if the same foreign key lookups are required in different parts of the application is to subclass a cursor and use this for our lookups. This is described in the Qt SQL Module documentation, particularly the section on subclassing **QSqlCursor**.

The 'book' example demonstrates the basic techniques needed for SQL programming with Qt. Additional information on the Qt SQL classes, especially the **QSqlQuery** and **QSqlCursor** classes is provided in the Qt SQL Module documentation.

Customizing and Integrating Qt Designer

Customizing Qt Designer

Qt Designer can be customized in two ways: you can add custom widgets, and you can change aspects of how *Qt Designer* works. Custom widgets are covered in *Creating Custom Widgets*. This section will focus on customizing *Qt Designer* itself.

Qt Designer's toolbars are all dockable so they can be dragged by their toolbar handles and arranged how you like. The Files, Object Hierarchy, Property Editor and Output Windows are also dockable so you can also drag them to the positions that you prefer. You can also make them into floating windows by dragging them outside *Qt Designer's* dock areas.

General preferences can be set by clicking **Edit|Preferences** to invoke the *Preferences* dialog. If you check the 'Restore Last Workspace on Startup' checkbox then *Qt Designer* will remember the sizes and positions of the toolbars and the dockable windows. You can change *Qt Designer's* main window background either by selecting a color or a pixmap. You can also switch off the grid (uncheck Show Grid) since using layouts makes the grid redundant.

The *Preferences* dialog may have additional tabs, depending on what plugins you have installed. We'll describe the C++ Editor tab since this is installed by default.

The C++ Editor tab is used to set your preferred fonts for syntax highlighting in *Qt Designer's* code editor. The base font for all elements is set in the 'Standard' element which is the last item in the list. If you want one font to be used throughout then set the 'Standard' font and all the other elements will inherit its setting.

Qt Designer's Code Editor

The code editor is available if an Editor plugin is installed. The C++ Editor plugin is installed by default.

The code editor provides the following keystrokes:

- Left Arrow — Moves the cursor one character left
- Right Arrow — Moves the cursor one character right
- Up Arrow — Moves the cursor one line up
- Down Arrow — Moves the cursor one line down
- Page Up — Moves the cursor one page up
- Page Down — Moves the cursor one page down
- Backspace — Deletes the character to the left of the cursor
- Home — Moves the cursor to the beginning of the line
- End — Moves the cursor to the end of the line
- Delete — Deletes the character to the right of the cursor
- Ctrl+A — Moves the cursor to the beginning of the line
- Ctrl+B — Moves the cursor one character left
- Ctrl+C — Copies the selected text to the clipboard (also Ctrl+Insert under Windows)

- Ctrl+D — Deletes the character to the right of the cursor
- Ctrl+E — Moves the cursor to the end of the line
- Ctrl+F — Invokes the *Find Text* dialog
- Ctrl+G — Invokes the *Goto Line* dialog
- Ctrl+H — Deletes the character to the left of the cursor
- Ctrl+I — Indent the line or selected text that contains the cursor
- Alt+I — Starts incremental search (see below)
- Ctrl+K — Deletes from the cursor position to the end of the line
- Ctrl+N — Moves the cursor one line down
- Ctrl+P — Moves the cursor one line up
- Ctrl+R — Invokes the *Replace Text* dialog
- Ctrl+V — Pastes the clipboard text into line edit (also Shift+Insert under Windows)
- Ctrl+X — Cuts the marked text, copy to clipboard (also Shift+Delete under Windows)
- Ctrl+Y — Redoes the last operation
- Ctrl+Z — Undoes the last operation
- Ctrl+Left Arrow — Moves the cursor one word left
- Ctrl+Right Arrow — Moves the cursor one word right
- Ctrl+Up Arrow — Moves the cursor one word up
- Ctrl+Down Arrow — Moves the cursor one word down
- Ctrl+Home Arrow — Moves the cursor to the beginning of the text
- Ctrl+End Arrow — Moves the cursor to the end of the text
- Tab — Completion (see below)

To select (mark) text hold down the Shift key whilst pressing one of the movement keystrokes, for example, **Shift+Right Arrow** will select the character to the right, and **Shift+Ctrl+Right Arrow** will select the word to the right, etc.

Pressing **Alt+I** starts incremental search. The characters you type will appear in the Incremental Search line edit in the Search toolbar and the cursor will be moved to the first matching text in the editor. As you type the search will continue. Press **Return** to move to the next match and press **Esc** to cancel the search at the position you've reached.

Pressing **Tab** after you've typed one or more characters invokes completion. Completion works like this: start typing some text then press **Tab**. If the editor can find another item of text that begins with the same characters it will complete your text for you; if it finds more than one possibility it will pop up a list of choices. You can use the arrow keys to choose a piece of text then press **Return**, or press **Esc** to continue typing. You can switch off completion in the *Preferences* dialog.

When you enter `->` or `.` the editor will pop up a command completion list; use the arrow keys to move to the item you want and press **Return**, or press **Esc** to ignore the list.

Creating and Using Templates

Qt Designer supports two approaches to creating template forms. The simplest approach involves little more than saving a `.ui` file into the templates directory. The second approach involves creating a container widget class to be used as a base class for forms that use the template. We will explain both techniques.

Simple Templates

These templates are most useful when you want to create a whole set of forms which all have some common widgets. For example, you might have a project that will require many forms, all of which need to be branded with a company name and logo.

First we'll create the simple template.

1. Click **File|New** to invoke the *New File* dialog. Click the Dialog template then click **OK**.
2. Click the **Text Label** toolbar button, then click near the top left of the form. Change the font Point Size property to 16 and change the *text* property to your or your company's name. Click the **Line** toolbar button, then click the form below the label; click Horizontal on the pop-up menu.
3. Select the label and the line. (**Ctrl+Click** the form, then drag the rubber band so that it touches or includes the line and the label.) Press **Ctrl+L** to lay them out vertically.
4. Click the **Save** toolbar button. In the *Save As* dialog, navigate to *Qt Designer's* templates directory, e.g. (`qt/tools/designer/templates`). Type in the name 'Simple_Dialog.ui' and click **Save**.
5. Right click the form in the Forms list, then click Remove form from project.

Now that we have the simple template we are ready to use it. Click **File|New** to invoke the *New File* dialog. One of the templates that will appear is 'Simple Dialog'. Click the simple dialog, then click **OK**. A new form will appear with the same widgets and layout as the template. Add any other widgets and functionality. When you attempt to save the form you will be prompted for a new form name.

Base-class Templates

These templates are useful when you want to provide some default functionality that all the forms based on the base class can inherit. In our example we'll use a class called **SizeAware** that remembers and restores its size as the basis of a template. We won't describe the class itself, but will focus instead on making use of it as a *Qt Designer* template. The source for the class is in `qt/tools/designer/examples/sizeaware`.

The template can either be based on a custom widget or on any existing container widget.

If you want to base the template on a custom widget you must first add it to *Qt Designer's* custom widgets. Click **Tools|Custom|Edit Custom Widgets** to invoke the *Edit Custom Widgets* dialog. (This dialog is explained in more detail in Simple Custom Widgets.) Click **New Widget**. Change the Class from 'MyCustomWidget' to 'SizeAware'. Click the Headerfile ellipsis button and select the file `qt/tools/designer/examples/sizeaware/sizeaware.h`. Check the Container Widget checkbox. This class provides two properties. Click the Properties tab. Click **New Property** and change the property name to 'company'. Click the **New Property** again and change the property name to 'settingsFile'. Click **Close**.

To create a template, based on an existing widget or on your own custom widget, click **File|Create Template** to invoke the *Create Template* dialog. Change the Template Name to 'SizeAware' and click the *SizeAware* base class, then click **Create**. The dialog will create the template and close itself immediately. Close *Qt Designer* and restart it.

A new template, 'SizeAware' is now available from the list of templates. Click **File|New**, click *SizeAware* and click **OK**. Note that the two properties, *company* and *settingsFile*, are available in the Properties window. Any forms based on this template will remember their size and resize when reloaded. (In practical applications having one *settingsFile* per form is not recommended, so this template would only really be useful for applications that have a single main window.)

Integrating Qt Designer with Visual Studio

Qt Designer can be integrated into Visual Studio using the `qmsdev.dsp` file that is supplied with Qt.

Start up Visual Studio and click **File|Open Workspace**. Open `%QTDIR%\tools\designer\integration\qmsdev\qmsdev.dsp`. Click **Build|Set Active Configuration** and in the list click 'QMsDev - Win32 Re-

lease', then click **OK**. Now click **Build|Build qmsdev.dll**. You should now copy the file `%QTDIR%\tools\designer\integration\qmsdev\Release\qmsdev.dll` into `Microsoft Visual Studio\Common\MSDev98\AddIns`. Now click **Tools|Customize**. Click the Add-in Macro Files tab, then click the **Browse** button. Change the file type to 'Add-ins (.dll)' and navigate to `Microsoft Visual Studio\Common\MSDev98\AddIns`. Click the `qmsdev.dll` file, click **Open**, then click **Close**.

A new toolbar will appear in Visual Studio with the following toolbar buttons:

- New Qt Project — A small application wizard
- Generate Qt Project — Runs `qmake` (or the functionally equivalent `tmake`) with a `.pro` file
- New Qt Dialog — Add an empty Qt Dialog to the active project
- Qt GUI Designer — Run *Qt Designer*
- Use Qt — Add the Qt libraries to the active project
- Add MOC — Add the `moc` precompiler to the active file
- Add UIC — Add the `uic` precompiler to the active file

Double clicking a `.ui` file in the workspace overview will now launch *Qt Designer*.

If you create a `.cpp` file which contains the `Q_OBJECT` macro you will need an additional file which is generated by the `moc` to be included in your project. For example, if you have 'file.cpp', then the last line would be `#include "file.moc"` and the additional file would be called 'file.moc'. To ensure that Visual Studio executes the `moc` and generates this file you must create a custom dependency. Double click the `.cpp` file (in your project workspace) that contains the `Q_OBJECT` macro. Click the **Add MOC** toolbar button; this will create an empty `.moc` file in your project workspace. Right click the newly created `.moc` file, then click **Settings** from the pop-up menu to invoke the Project Settings dialog. Click the Custom Build tab. Click the **Dependencies** button to pop up the User Defined Dependencies dialog. Type in `$(InputDir)\$(InputPath)`, then press **Return**. Click **OK** to leave the Dependencies dialog, then click **OK** to leave the Project Settings dialog.

If you wish to delete the add-in remove it from the toolbar then delete the `qmsdev.dll` file from the add-ins directory.

Creating Makefiles without qmake

The `qmake` tool provided with Qt can create Makefiles appropriate to your platform based on `.pro` project files. This section describes the dependencies involved in building a Qt application and gives a couple of simple example Makefiles. This section assumes that you have a good understanding of Makefiles.

Qt Designer produces `.ui` files which are used to generate `.h` and `.cpp` files for the compiler to compile. The `.ui` files are processed by `uic`. Classes which inherit from **QObject**, e.g. those which use slots and signals, require an additional `.cpp` file to be generated. These files are generated by the `moc` and are named '`moc_file.cpp`' where the original `.cpp` file is called 'file.cpp'. If your `.cpp` file contains the `Q_OBJECT` macro an additional file 'file.moc' should be generated which must be `#included` in the `.cpp`, normally at the end. This requires an extra dependency being created.

Processing `.ui` files with `uic` is done *twice*:

```
uic myform.ui -o myform.h
uic myform.ui -i myform.h -o myform.cpp
```

The first execution creates the header file, the second creates the `.cpp` file. If you wish to subclass a form you can use `uic` to generate subclass skeletons:

```
uic formbase.ui -o formbase.h
uic formbase.ui -i formbase.h -o formbase.cpp
uic -subdecl Form formbase.h formbase.ui -o form.h
uic -subimpl Form formbase.h formbase.ui -o form.cpp
```

First we generate the header and implementation file for our base class. Then we generate the header and implementation skeletons for our subclass. Note that the use of `uic` to generate skeletons is not something that would be done in a Makefile, we mention it here because it can be useful for command line users.

For implementation files that contain classes which inherit from **QObject** we must create moc files:

```
moc myform.h -o moc_myform.cpp
```

We'll look at a simple Makefile to see the dependencies in practice.

```
myapp: moc_myform.o myform.o main.o
    g++ -lqt -o myapp moc_myform.o myform.o main.o

main.o: main.cpp
    g++ -o main.o main.cpp

moc_myform.o: moc_myform.cpp
    g++ -o moc_myform.o moc_myform.cpp

moc_myform.cpp: myform.h
    moc myform.h -o moc_myform.cpp

myform.o: myform.cpp
    g++ -o myform.o myform.cpp

myform.cpp: myform.h myform.ui
    uic myform.ui -i myform.h -o myform.cpp

myform.h: myform.ui
    uic myform.ui -o myform.h
```

Note that you may need to include the full path to the commands in your Makefile, and under Windows the filenames are `moc.exe` and `uic.exe`.

In Unix/Linux environments the `make` command may be able to do more for us, so we should be able to use a simpler Makefile like this:

```
myapp: moc_myform.o myform.o main.o
    g++ -lq -o $@ $^

%.o: %.cpp
    g++ -o $^ $@

moc_%.cpp: %.h
    moc $^ -o $@

myform.cpp: myform.h myform.ui
    uic myform.ui -i myform.h -o myform.cpp

myform.h: myform.ui
    uic myform.ui -o myform.h
```

To see more sophisticated Makefiles simply generate them using `qmake` on any of your Qt projects or any of the examples supplied with Qt.

Importing Foreign File Formats

To import a file in a supported foreign file format click **File | Open**, then click the File Type combobox to choose the file type you wish to load. Click the required file and *Qt Designer* will convert and load the file.

The filters that *Qt Designer* uses to read foreign file formats are 'works in progress'. You may have different filters available in your version of *Qt Designer* than those described here. The easiest way to see which filters are available is to invoke the file open dialog; all your filters are listed in the File Type combobox.

Importing Qt Architect Files

Qt Architect is a free GUI builder for Qt written by Jeff Harris and Klaus Ebner. The `.dlg` extension is associated with Qt Architect dialog files.

Qt Designer can read files generated by Qt Architect version 2.1 and above. When given a `.dlg` file from a previous version of Qt Architect, *Qt Designer* tells you how to convert it to the file format of version 2.1. (The conversion procedure varies depending on the version of the `.dlg` file.)

The import filter does a good job of importing `.dlg` files; the result is almost identical to what you get in Qt Architect. However, the C++ code that uses the dialogs will probably need some adaptation.

There are a few drawbacks to converting Qt Architect files to *Qt Designer's* format due to differences between the two tools; these are listed below:

- Layout spacing and margins

If the `.dlg` file layouts use the Qt Architect defaults for layout spacing and margins, *Qt Designer* will override these with its standard defaults. You can change the "layoutSpacing" and "layoutMargin" properties manually afterwards if necessary.

- Layout stretches and spacings

Qt Architect gives access to more features of Qt's layout system than *Qt Designer*, namely stretches and spacings. *Qt Designer* will attempt to cope with `.dlg` files that use these features, but sometimes the resizing will not be what you want. The solution typically involves setting the "sizePolicy" properties of some widgets and inserting or deleting spacers.

- Mixing managed and unmanaged widgets

Qt Architect allows a widget to have some child widgets managed by a layout and other child widgets with fixed positions. When presented with a `.dlg` file that uses this facility, *Qt Designer* will silently put the fixed position widgets into the layout.

- Pixmaps

Qt Designer ignores pixmaps specified in `.dlg` files. These have to be restored manually in *Qt Designer*.

Importing Glade Files

Glade is a free GUI builder for GTK+ and GNOME written by Damon Chaplin. The `.glade` extension is associated with Glade files.

Qt Designer has been tested with Glade files up to version 0.6.0 and might work with later versions as well.

Although Glade does not target Qt, the layout system and the widget set of GTK+ are similar to those of Qt, so the filter will retain most of the information in the `.glade` file.

There are some considerations regarding the conversion of Glade files, as listed below:

- Ampersands (&) in labels

Qt displays an ampersand when a **QLabel** has no buddy. (A buddy is a widget that accepts focus on behalf of a **QLabel**.) Glade allows GtkLabel widgets with an (underlined) accelerator key but with no buddy. This

is an error since users expect underlined characters to be accelerators. In this situation, Qt displays the ampersand itself instead of underlining the accelerator key. You should go over these **QLabel** widgets and set their "buddy" property.

- Layout placeholders

GTK allows a layout position to be occupied by a placeholder. *Qt Designer* converts those placeholders into **QLabels** whose text is "?" in red, so that you can find them and fix them manually.

- GTK+ or GNOME widget with no Qt equivalent

Qt has equivalents for most GTK+ widgets, but Glade also supports GNOME, whose goal is to provide a complete desktop environment. Because Qt's scope is narrower, when *Qt Designer* encounters a widget it cannot convert, it replaces it with a label that indicates the problem. For example, a `GnomePaperSelector` will be replaced by a **QLabel** whose text is "GnomePaperSelector?" in red. If you are porting to KDE, you might want to use the corresponding KDE widget.

Other GTK+/GNOME widgets are only supported in certain contexts. For example, the `GnomeDruid` can be embedded in another widget, whereas the corresponding **QWizard** class cannot.

- Message boxes and other high-level dialogs

Glade supports editing of `GnomeMessageBox`, `GtkFileSelection`, `GtkFontSelectionDialog` and others. This is trivially achieved in Qt by means of a **QMessageBox** dialog, a **QFileDialog**, a **QFontDialog**, etc., in C++ code.

- Stand-alone popup menus

Qt Designer only supports popup menus inside a **QMainWindow**. If you need a stand-alone popup menu (presumably a context menu), you can easily write code that does this using **QPopupMenu**.

- Size policy parameters

Glade provides size policies in the "Place" tab of the property editor. *Qt Designer* does not attempt to make use of the padding, expand, shrink and fill information, as the Qt defaults are usually good enough. In a few cases, you might have to set the "sizePolicy" property manually to obtain the effect you want.

- GNOME standard icons

GNOME provides a large set of standard icons. *Qt Designer* will ignore references to these. If you are porting to KDE, you might want to manually set the standard KDE icons.

- Packer layout

GTK+ provides a class called `GtkPacker` that provides for exotic layouts; Qt provides no **QPackerLayout** and never will. *Qt Designer* will treat packer layouts as if they were vertical layouts and you will probably have to change them to whatever combination of layouts that produces the right effect.

- Incorrectly-justified text after conversion

The "hAlign" property is sometimes set wrongly, in which case you have to change it manually. It is caused by a quirk in Glade.

Reference: Key Bindings

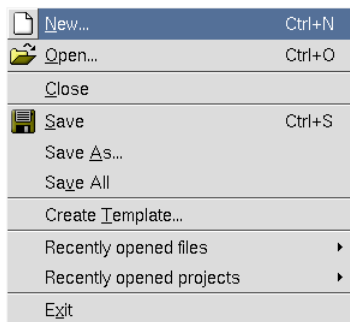
- Ctrl+A — Selects all GUI elements in the active form.
- Ctrl+B — Breaks the selected layout so that you can add or delete GUI elements.
- Ctrl+C — Copies the selected GUI elements from the active form into the clipboard.
- Alt+E — Pulls down the **Edit** menu.
- Alt+F — Pulls down the **File** menu.
- Ctrl+G — Applies a grid layout to the selected container, or creates a new container containing the selected GUI elements and applies a grid layout to this container.
- Ctrl+H — Applies a horizontal box layout to the selected container, or creates a new container containing the selected GUI elements and applies a horizontal box layout to this container.
- Alt+H — Pulls down the **Help** menu.
- Ctrl+J Adjusts the size of the selected GUI element (or elements) so that it has the minimal size needed for displaying itself properly.
- Ctrl+L — Applies a vertical box layout to the selected container, or creates a new container containing the selected GUI elements and applies a vertical box layout to this container.
- Alt+L — Pulls down the **Layout** menu.
- Ctrl+M — Opens an online version of this manual in Qt Assistant.
- Ctrl+N — Invokes the *New File* dialog.
- Ctrl+O — Invokes the *Open File* dialog.
- Alt+P — Pulls down the **Preview** menu.
- Ctrl+R — Checks the accelerators in the active form for duplicates.
- Ctrl+S — Saves the active form.
- Ctrl+T — Previews the active form in the default GUI style of the platform.
- Alt+T — Pulls down the **Tools** menu.
- Ctrl+V — Pastes the GUI element (or elements) in the clipboard into the active form at the position it had in its original form plus a little offset. Does nothing if the clipboard does not contain a GUI element.
- Alt+W — Pulls down the **Window** menu.
- Ctrl+X — Cuts the selected GUI element (or elements) from the active form and puts it into the clipboard.
- Ctrl+Y — Redoes the last undo action.
- Ctrl+Z — Undoes the last action.
- Del — Deletes the selected GUI elements from the active form.
- F1 — Opens the introductory page of the Qt Designer manual in Qt Assistant.
- Shift-F1 — Turns on What's This mode, which lets you click on a GUI element in Qt Designer to get a small description window for this element.
- F2 — Activates the pointer tool that lets you select GUI elements.
- F3 — Activates the connection tool that lets you edit the connections between signals and slots in a form.
- F4 — Activates the tab order tool that lets you change the tab order of the GUI elements on the active form.

- Ctrl+F4 — Closes the active window.
- Ctrl+F6 — Activates the next window in the order of window creation.
- Ctrl+Shift-F6 — Activates the previous window in the order of window creation.

Reference: Menu Options

Introduction

Qt Designer provides menu options that invoke actions that are used to help create applications. Many menu options lead to dialog boxes that provide specific functionality. The most common menu options also have corresponding toolbar buttons. This chapter explains each menu option and its use. For menu options that invoke a dialog box or which have a corresponding toolbar button, there is a cross-reference to the detailed explanation that appears in the relevant chapter.



The File Menu

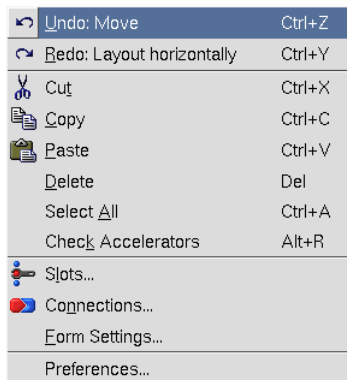
The File Menu

This menu is invoked with **Alt+F**, and provides the following options:

- **File|New** Click this menu option (or press **Ctrl+N**) to create a new project, form or file. This option invokes the New File Dialog.
- **File|Open** Click this menu option (or press **Ctrl+O**) to open existing projects, forms or files. The File Open Dialog is invoked through which a file name can be selected.
- **File|Close** Click this menu option to close the currently open project. If the project has unsaved changes, the Save Form Dialog appears.
- **File|Save** Click this menu option (or press **Ctrl+S**) to save the project along with its forms and files. For a project that has forms or files, click 'Save' to save the project before exiting. For new forms, click 'Save' and the Save Form As Dialog appears. For forms that have been saved previously click 'Save'. For new files or for files that have been changed, click 'Save'.
- **File|Save As** Click this menu option to save and name the current form or file. This option invokes the Save Form As Dialog.
- **File|Save All** Click this menu option to save every open file and form in every open project.
- **File|Create Template** Click this menu option to create a form template. This option invokes the Create Template Dialog dialog.

- **File|Recently Opened Files** Click this menu option to list the most recently opened files. Click one of the files listed to open it. Note that we recommend that you open projects rather than files. You can open a file by clicking the file's name in the project's File Overview Window.
- **File|Recently Opened Projects** Click this menu option to list the most recently opened projects. Click one of the projects listed to open it.
- **File|Exit** Click this menu option to exit *Qt Designer*. If any open files have unsaved changes, the *Save Form Dialog* message box will appear for each of them, before *Qt Designer* exits. Note that for a form that has not been saved previously but has had changes made to it or that has been saved but has had changes made to it, the *Save Form Dialog* is invoked. Click **Yes** to invoke the *Save Form As* dialog.

See also The File Toolbar Buttons.



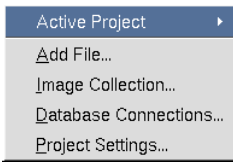
The Edit Menu

The Edit Menu

This menu is invoked with **Alt+E**, and provides the following options:

- **Edit|Undo** Click this menu option (or press **Ctrl+Z**) to undo an action. The name of the last action that was performed appears after the word 'Undo'.
- **Edit|Redo** Click this menu option (or press **Ctrl+Y**) to redo an action. The name of the last action that was performed appears after the word 'Redo'.
- **Edit|Cut** Click this menu option (or press **Ctrl+X**) to delete the selected item from the current form or file and copy it to the clipboard.
- **Edit|Copy** Click this menu option (or press **Ctrl+C**) to copy the selected item from the current form or file to the clipboard.
- **Edit|Paste** Click this menu option (or press **Ctrl+V**) to paste the clipboard item (if any) into the current form or file.
- **Edit|Delete** Click this menu option (or press **Del**) to delete the selected item from the current form or file.
- **Edit|Select All** Click this menu option (or press **Ctrl+A**) to highlight all the widgets on the current form or all the text in the current file.
- **Edit|Check Accelerators** Click this menu option (or press **Alt+R**) to verify that all the accelerators are used only once. If an accelerator is used more than once, a message box appears with the statement 'The accelerator 'x' is used 'y' times'. Click **Select** to highlight the widgets with the same accelerator, or click **Cancel** to exit the message box without taking any action.
- **Edit|Slots** Click this menu option to edit and create slots. This option invokes the Edit Slots Dialog.
- **Edit|Connections** Click this menu option to invoke the View Connections Dialog.
- **Edit|Form Settings** Click this menu option to invoke the Form Settings Dialog.
- **Edit|Preferences** Click this menu option to invoke the Preferences Dialog.

See also The Edit Toolbar Buttons.



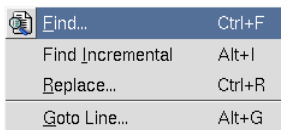
The Project Menu

The Project Menu

This menu is invoked with **Alt+O**, and provides the following options:

- **Project| Active Project** Click this menu option to toggle between projects if there is more than one project open. You can also toggle between projects using the Active Project drop-down combobox in the The File Toolbar Buttons.
- **Project| Add File** Click this menu option to invoke the Add Dialog
- **Project| Project Settings** Click this menu option to invoke the Project Settings Dialog.
- **Project| Image Collection** Click this menu option to invoke the Image Collection Dialog.
- **Project| Database Connections** Click this menu option to invoke the Edit Database Connections Dialog.

See also The File Toolbar Buttons.



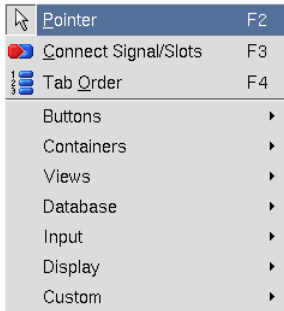
The Search Menu

The Search Menu

This menu is invoked with **Alt+S**, and provides the following options:

- **Search| Find** Click this menu option (or press **Ctrl+F**) to invoke the Find Text Dialog.
- **Search| Find Incremental** Click this menu option (or press **Alt+I**) to place the cursor in the text box located next to the **Find** toolbar button. Type characters into the text box; as you type, *Qt Designer* will highlight the first occurrence of the text that it finds in the file. Press the **Enter** key to go to the next occurrence of the text. Press the **Esc** key once you have found the word you are looking for to place the cursor in the editor.
- **Search| Replace** Click this menu option (or press **Ctrl+R**) to invoke the Replace Text Dialog to replace specific words or characters.
- **Search| Goto line** Click this menu option (or press **Alt+G**) to invoke the Goto Line Dialog to go to a specific line in the file.

See also The Search Toolbar Buttons.



The Tools Menu









The Tools Menu

This menu is invoked with **Alt+T**, and provides the following options:

- **Tools|Pointer** Click this menu option (or press **F2**) to de-select any selected widget toolbar button. The pointer is also used to stop inserting new widgets on the form if you double clicked a widget toolbar button. Press the **Esc** key to return to the pointer at any time.
- **Tools|Connect Signals and Slots** Click this menu option (or press **F3**) to connect signals and slots. Click on a widget and drag the connection line to the widget (or form) that you want to connect to. Release the mouse button and the Edit Connections Dialog will appear.
- **Tools|Tab Order** Click this menu option (or press **F4**) to set the tab order for all the widgets on the form that can accept keyboard focus. Choose this option and blue circles with numbers on them appear next to the widgets. Click the widget that you want to be first in the tab order, then click the widget that should be next in the tab order, and continue until all the widgets have the tab order numbers you want. If you make a mistake, double click the first widget and start again. Press **Esc** to leave tab order mode. If you want to revert your changes, leave tab order mode, then undo.
- **Tools|Buttons|PushButton** Click this menu option and then click the form to place a PushButton on the form.
- **Tools|Buttons|ToolButton** Click this menu option and then click the form to place a ToolButton on the form.
- **Tools|Buttons|RadioButton** Click this menu option and then click the form to place a RadioButton on the form. It is recommended that you place RadioButtons inside ButtonGroups so that Qt will automatically ensure that only one RadioButton in the group is active at any one time.
- **Tools|Buttons|CheckBox** Click this menu option and then click the form to place a CheckBox on the form.
- **Tools|Containers|GroupBox** Click this menu option and then click the form to place a GroupBox on the form.
- **Tools|Containers|ButtonGroup** Click this menu option and then click the form to place a ButtonGroup on the form.
- **Tools|Containers|Frame** Click this menu option and then click the form to place a Frame on the form.
- **Tools|Containers|TabWidget** Click this menu option and then click the form to place a TabWidget on the form. To add or remove tabs, right click the tab widget and choose 'Add Page' or 'Remove Page'.
- **Tools|Views|ListBox** Click this menu option and then click the form to place a ListBox on the form.
- **Tools|Views|ListView** Click this menu option and then click the form to place a ListView on the form.
- **Tools|Views|Icon View** Click this menu option and then click the form to place an IconView on the form.
- **Tools|Views|Table** Click this menu option and then click the form to place a Table on the form.
- **Tools|Database|DataTable** Click this menu option and then click the form to place a DataTable on the form.
- **Tools|Database|DataBrowser** Click this menu option and then click the form to place a DataBrowser on the form.
- **Tools|Database|DataView** Click this menu option and then click the form to place a DataView on the form.

- **Tools|Input|LineEdit** Click this menu option and then click the form to place a LineEdit on the form.
- **Tools|Input|SpinBox** Click this menu option and then click the form to place a SpinBox on the form.
- **Tools|Input|DateEdit** Click this menu option and then click the form to place a DateEdit on the form.
- **Tools|Input|TimeEdit** Click this menu option and then click the form to place a TimeEdit on the form.
- **Tools|Input|DateTimeEdit** Click this menu option and then click the form to place a DateTimeEdit on the form.
- **Tools|Input|TextEdit** Click this menu option and then click the form to place a TextEdit on the form.
- **Tools|Input|ComboBox** Click this menu option and then click the form to place a ComboBox on the form.
- **Tools|Input|Slider** Click this menu option and then click the form to place a Slider on the form.
- **Tools|Input|ScrollBar** Click this menu option and then click the form to place a Scrollbar on the form.
- **Tools|Input|Dial** Click this menu option and then click the form to place a Dial on the form.
- **Tools|Display|TextLabel** Click this menu option and then click the form to place a TextLabel on the form.
- **Tools|Display|PixmapLabel** Click this menu option and then click the form to place a PixmapLabel on the form.
- **Tools|Display|LCDNumber** Click this menu option and then click the form to place a LCDNumber on the form.
- **Tools|Display|Line** Click this menu option and then click the form to place a Line on the form.
- **Tools|Display|ProgressBar** Click this menu option and then click the form to place a ProgressBar on the form.
- **Tools|Display|TextBrowser** Click this menu option and then click the form to place a TextBrowser on the form.
- **Tools|Custom|Edit Custom Widgets** Click this menu option to invoke the Edit Custom Widgets Dialog.
- **Tools|Custom|** Click this menu option and then click the form to place the Custom Widget on the form. Note that this menu option only appears if you have created a widget using **Tools|Custom|Edit Custom Widgets**.

See also The Tools Toolbar Buttons.

	Adjust Size	Ctrl+J
	Lay Out Horizontally	Ctrl+H
	Lay Out Vertically	Ctrl+L
	Lay Out in a Grid	Ctrl+G
	Lay Out Horizontally (in Splitter)	
	Lay Out Vertically (in Splitter)	
	Break Layout	Ctrl+B
	Add Spacer	

The Layout Menu

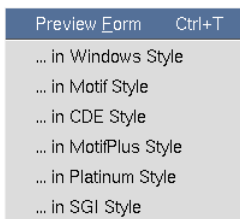
The Layout Menu

This menu is invoked with **Alt+L**, and provides the following options:

- **Layout|Adjust Size** Click this menu option (or press **Ctrl+J**) to adjust the size of the widget to its recommended size.
- **Layout|Lay Out Horizontally** Click this menu option (or press **Ctrl+H**) to lay out the selected widgets or layouts side by side. Use **Shift+Click** to select each widget or layout, and then choose this menu option to group them horizontally. Note that for complex widgets it is sometimes easiest to select widgets and layouts by clicking them in the Widgets tab of the Object Explorer Window. If only one widget is selected, its child widgets will be laid out horizontally.

- **Layout|Lay Out Vertically** Click this menu option (or press **Ctrl+L**) to lay out the selected widgets one above the other. Use **Shift+Click** to select each widget or layout, and then choose this menu option to group them vertically. Note that for complex widgets it is sometimes easiest to select widgets and layouts by clicking them in the Widgets tab of the Object Explorer Window. If only one widget is selected, its child widgets will be laid out vertically.
- **Layout|Lay Out in a Grid** Click this menu option (or press **Ctrl+G**) to lay out the selected widgets in a grid. If only one widget is selected, its child widgets will be laid out in a grid.
- **Layout|Lay Out Horizontally (in Splitter)** Click this menu option to lay out the selected widgets or layouts side by side with a splitter between each. Use **Shift+Click** to select each widget or layout, and then choose this menu option to group them horizontally. Note that for complex widgets it is sometimes easiest to select widgets and layouts by clicking them in the Widgets tab of the Object Explorer Window.
- **Layout|Lay Out Vertically (in Splitter)** Click this menu option to lay out the selected widgets or layouts one above the other with a splitter between each. Use **Shift+Click** to select each widget or layout, and then choose this menu option to group them vertically. Note that for complex widgets it is sometimes easiest to select widgets and layouts by clicking them in the Widgets tab of the Object Explorer Window.
- **Layout|Break Layout** Click this menu option (or press **Ctrl+B**) to break a layout. Click on the layout, then select this option; the layout is deleted.
- **Layout|Add Spacer** Click this menu option to add a vertical or horizontal spacer to widgets that take up too much space on the form. The spacer consumes extra space in the layout.

See also The Layout Toolbar Buttons.

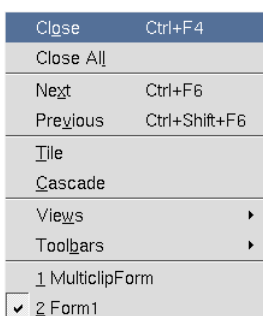


The Preview Menu

The Preview Menu

This menu is invoked with **Alt+P**, and provides the following options:

- **Preview|Preview Form** Click this menu option (or press **Ctrl+T**) to preview the form within *Qt Designer*.
- **Preview|...in Windows Style** Click this menu option to preview the form in the Windows style.
- **Preview|...in Motif Style** Click this menu option to preview the form in the Motif style.
- **Preview|...in CDE Style** Click this menu option to preview the form in the CDE style.
- **Preview|...in MotifPlus Style** Click this menu option to preview the form in the MotifPlus style.
- **Preview|...in Platinum Style** Click this menu option to preview the form in the Platinum style.
- **Preview|...in SGI Style** Click this menu option to preview the form in the SGI style.



The Window Menu

The Window Menu

This menu is invoked with **Alt+W**, and provides the following options:

- **Window|Close** Click this menu option (or press **Ctrl+F4**) to close the window that is currently active.
- **Window|Close All** Click this menu option to close all the windows that are currently open.
- **Window|Next** Click this menu option (or press **Ctrl+F6**) to make the next window active. The order is the order in which the windows were opened.
- **Window|Previous** Click this menu option (or press **Ctrl+Shift+F6**) to make the previous window active. The order is the order in which the windows were opened.
- **Window|Tile** Click this menu option to arrange all the open files and forms side by side so that each window is visible.
- **Window|Cascade** Click this menu option to stack all the open file and forms, one on top of the other, but with an overlap so that each window's title bar is visible.
- **Window|Views|File Overview** Click this menu option to make the File Overview Window visible, or to hide it if it is already visible. If the window is currently visible, a check mark will appear next to the name in the menu.
- **Window|Views|Property Editor/Signal Handlers** Click this menu option to make the Property Editor/Signal Handlers Window visible, or to hide it if it is already visible. If the window is currently visible, a check mark will appear next to the name in the menu.
- **Window|Views|Object Explorer** Click this menu option to make the Object Explorer Window visible, or to hide it if it is already visible. If the window is currently visible, a check mark will appear next to the name in the menu.
- **Window|Views|Line Up** Click this menu option to eliminate any extra space between toolbars and line them up next to each other all at once, rather than moving each individual toolbar into place.
- **Window|Toolbars|File** Click this menu option to make the File toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Edit** Click this menu option to make the Edit toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Search** Click this menu option to make the Search toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Layout** Click this menu option to make the Layout toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Tools** Click this menu option to make the Tools toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Buttons** Click this menu option to make the Buttons toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Containers** Click this menu option to make the Containers toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Views** Click this menu option to make the Views toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.

- **Window|Toolbars|Database** Click this menu option to make the Database toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Input** Click this menu option to make the Input toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Display** Click this menu option to make the Display toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Custom** Click this menu option to make the Custom toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Help** Click this menu option to make the Help toolbar buttons visible, or to hide them if they are already visible. If the toolbar buttons are currently visible, a check mark will appear next to the name in the menu.
- **Window|Toolbars|Line Up** Click this menu option to eliminate extra space between toolbars and line them up next to each other all at once, rather than moving each individual toolbar into place.
- **Window|n** Click one of the numbered menu options that list the currently open files and forms to switch to the named file or form.

C <u>ontents</u>	F1
M <u>anual</u>	Ctrl+M
A <u>bout...</u>	
A <u>bout Qt...</u>	
What's This?	Shift+F1

The Help Menu

The Help Menu

This menu is invoked with **Alt+H**, and provides the following options:

- **Help|Contents** Click this menu option (or press **F1**) to invoke the Qt Assistant application which provides on-line help. The on-line help is context sensitive, so you can type the item you want more information about in the line edit and Qt Assistant will automatically find it if it is available.
- **Help|Manual** Click this menu option (or press **Ctrl+M**) to invoke the Qt Assistant application which opens showing this manual.
- **Help|About...** Click this menu option to invoke the *About Qt Designer* dialog which gives the version number and some licensing information.
- **Help|About Qt...** Click this menu option to invoke a dialog which provides information about Qt.
- **Help|What's This?** Click this menu option to invoke a small question mark that is attached to the mouse pointer. Click on a feature which you would like more information about. A popup box appears with information about the feature.

See also The Help Toolbar Button.

Reference: Toolbar Buttons

Introduction

Qt Designer's toolbar buttons provide fast access to common functionality.

Toolbar buttons are grouped in several toolbars. Toolbars have a handle at the left hand side which can be clicked to minimize the toolbar. Toolbars that have been minimized have their handle appear just under the menu bar; click the handle to restore the toolbar to the last position it occupied. You can drag a toolbar's handle to move the toolbar to a different position in the toolbar area. Toolbars can be dragged out of the toolbar area entirely and made into stand-alone tool dock windows. To hide a tool dock window click its close button. To restore a hidden tool dock window, right click the tool area, then click the name of the tool dock window you wish to restore.



File Toolbuttons

The File Toolbar Buttons

- **New** Click this toolbar button (or press **Ctrl+N**) to create a new project, form or file. This option invokes the New File Dialog.
- **Open** Click this toolbar button (or press **Ctrl+O**) to open existing projects, forms or files. This button invokes the File Open Dialog which is used to select files.
- **Save** Click this toolbar button (or press **Ctrl+S**) to save the project, forms and files. For a new project that has no forms or files, click 'Save' to save the project before exiting. For new forms, click 'Save' and the Save Form As Dialog appears.
- **Active Project** Click the combobox to view the names of the projects that are currently open and select a project name to toggle between the projects.



Edit Toolbuttons

The Edit Toolbar Buttons

- **Undo** Click this toolbar button (or press **Ctrl+Z**) to undo an action. The name of the last action that was performed appears after the word 'Undo' in this toolbar button's tooltip.
- **Redo** Click this toolbar button (or press **Ctrl+Y**) to redo an action. The name of the last action that was performed appears after the word 'Redo' in this toolbar button's tooltip.
- **Cut** Click this toolbar button (or press **Ctrl+X**) to delete the selected item from the current form or file and copy it to the clipboard.

- **Copy** Click this toolbar button (or press **Ctrl+C**) to copy the selected item from the current form or file to the clipboard.
- **Paste** Click this toolbar button (or press **Ctrl+V**) to paste the selected item (if any) from the clipboard into the current form or file.



Search Toolbuttons

The Search Toolbar Buttons

- **Find** Click this toolbar button (or press **Ctrl+F**) to invoke the Find Text Dialog.
- **Find Incremental** Click this toolbar button (or press **Alt+I**) to place the cursor in the text box located next to the **Find** toolbar button. Type characters into the text box; as you type, *Qt Designer* will highlight the first occurrence of the text that it finds in the file. Press the **Enter** key to go to the next occurrence of the text. Press the **Esc** key once you have found the word you are looking for to place the cursor in the editor.

The Tools Toolbar Buttons

If you want to add the same kind of widget several times to a form, for example, several push buttons, *double click* the widget's toolbar button. After this, every time you click the form a new widget will be added. Click the **Pointer** toolbar button to leave this mode.



Tools

Tools

- **Pointer** Click this toolbar button (or press **F2**) to de-select any selected widget toolbar button. The pointer is also used to stop inserting new widgets if you double clicked a widget toolbar button. Press the **Esc** key to return to the pointer at any time.
- **Connect Signals and Slots** Click this toolbar button (or press **F3**) to connect signals and slots. Then click on a widget and drag the connection line to the widget (or the form) that you want to connect to. Release the mouse button and the Edit Connections Dialog will appear.
- **Tab Order** Click this toolbar button (or press **F4**) to set the tab order for all the widgets on the form that can accept keyboard focus. Click this toolbar button and blue circles with numbers on them appear next to the widgets. Click the widget that you want to be first in the tab order, then click the widget that should be next in the tab order, and continue until all the widgets have the tab order numbers you want. If you make a mistake, double click the first widget and start again. Press **Esc** to leave tab order mode. If you want to revert your changes, leave tab order mode, then undo.



Buttons

Buttons

- **PushButton** Click this toolbar button, then click the form, to place a Pushbutton on the form.

- **ToolButton** Click this toolbar button, then click the form, to place a Toolbutton on the form.
- **RadioButton** Click this toolbar button, then click the form, to place a Radiobutton on the form. It is recommended that you place RadioButtons inside ButtonGroups so that Qt will automatically ensure that only one RadioButton in the group is active at any one time.
- **CheckBox** Click this toolbar button, then click the form, to place a CheckBox on the form.



Containers

Containers

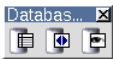
- **GroupBox** Click this toolbar button, then click the form, to place a GroupBox on the form.
- **ButtonGroup** Click this toolbar button, then click the form, to place a ButtonGroup on the form.
- **Frame** Click this toolbar button, then click the form, to place a Frame on the form.
- **TabWidget** Click this toolbar button, then click the form, to place a TabWidget on the form. To add or remove tabs, right click the tab widget and choose 'Add Page' or 'Remove Page'.



Views

Views

- **ListBox** Click this toolbar button, then click the form, to place a ListBox on the form.
- **ListView** Click this toolbar button, then click the form, to place a ListView on the form.
- **Icon View** Click this toolbar button, then click the form, to place an IconView on the form.
- **Table** Click this toolbar button, then click the form to place a Table on the form.



Database Toolbuttons

Database

- **DataTable** Click this toolbar button, then click the form, to place a DataTable on the form.
- **DataBrowser** Click this toolbar button, then click the form, to place a DataBrowser on the form.
- **DataView** Click this toolbar button, then click the form, to place a DataView on the form.



Input Toolbuttons

Input

- **LineEdit** Click this toolbar button, then click the form, to place a LineEdit on the form.
- **SpinBox** Click this toolbar button, then click the form, to place a SpinBox on the form.
- **DateEdit** Click this toolbar button, then click the form, to place a DateEdit on the form.
- **TimeEdit** Click this toolbar button, then click the form, to place a TimeEdit on the form.
- **DateTimeEdit** Click this toolbar button, then click the form, to place a DateTimeEdit on the form.
- **TextEdit** Click this toolbar button, then click the form, to place a TextEdit on the form.
- **ComboBox** Click this toolbar button, then click the form, to place a ComboBox on the form.
- **Slider** Click this toolbar button, then click the form, to place a Slider on the form.
- **ScrollBar** Click this toolbar button, then click the form, to place a Scrollbar on the form.
- **Dial** Click this toolbar button, then click the form, to place a Dial on the form.



Display Toolbuttons

Display

- **TextLabel** Click this toolbar button, then click the form, to place a TextLabel on the form.
- **PixmapLabel** Click this toolbar button, then click the form, to place a PixmapLabel on the form.
- **LCDNumber** Click this toolbar button, then click the form, to place a LCDNumber on the form.
- **Line** Click this toolbar button, then click the form, to place a Line on the form.
- **ProgressBar** Click this toolbar button, then click the form, to place a ProgressBar on the form.
- **TextBrowser** Click this toolbar button, then click the form, to place a TextBrowser on the form.



Custom Widget Toolbutton

Custom

- **My Custom Widget** Click this toolbar button, then click the form, to place a Custom Widget on the form. Note: this toolbar button only appears if you have created a custom widget using **Tools | Custom | Edit Custom Widgets**.



Layout Toolbuttons

The Layout Toolbar Buttons

- **Adjust Size** Click this toolbar button (or press **Ctrl+J**) to adjust the size of the widget to its recommended size.
- **Lay Out Horizontally** Click this toolbar button (or press **Ctrl+H**) to lay out the selected widgets or layouts side by side. Use **Shift+Click** to select each widget or layout, and then choose this toolbar button to group them horizontally. Note that for complex widgets it is sometimes easiest to select widgets and layouts by clicking them in the Widgets tab of the Object Explorer Window.

- **Lay Out Vertically** Click this toolbar button (or press **Ctrl+L**) to lay out the selected widgets one above the other. Use **Shift+Click** to select each widget or layout, and then choose this toolbar button to group them vertically. Note that for complex widgets it is sometimes easiest to select widgets and layouts by clicking them in the Widgets tab of the Object Explorer Window.
- **Lay out in a Grid** Click the widgets you want and then click this toolbar button (or press **Ctrl+G**) to lay out widgets in a grid.
- **Lay Out Horizontally (in Splitter)** Click this toolbar button to lay out the selected groups of widgets or layouts side by side with a splitter between each group. Use **Shift+Click** to select each widget or layout, and then choose this toolbar button to group them horizontally. Note that for complex widgets it is sometimes easiest to select widgets and layouts by clicking them in the Widgets tab of the Object Explorer Window.
- **Lay Out Vertically (in Splitter)** Click this toolbar button to lay out the selected groups of widgets or layouts one above the other with a splitter between each group. Use **Shift+Click** to select each widget or layout, and then choose this toolbar button to group them vertically. Note that for complex widgets it is sometimes easiest to select widgets and layouts by clicking them in the Widgets tab of the Object Explorer Window.
- **Break Layout** Click this toolbar button (or press **Ctrl+B**) to break a layout. Click on the layout and select this option; the layout is deleted.
- **Add Spacer** Click this toolbar button to add a spacer to widgets that take up too much space on the form. The spacer consumes extra space in the layout.



Help Toolbar Button

The Help Toolbar Button

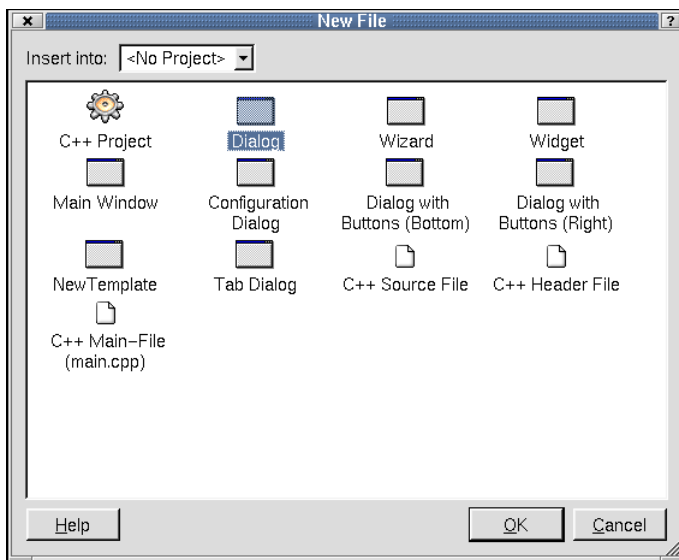
- **What's This?** Click this menu option to invoke a small question mark that is attached to the mouse pointer. Click on a feature which you would like more information about. A message box appears with information about the feature.

Reference: Dialogs

Introduction

This chapter describes and explains every *Qt Designer* dialog.

The File Dialogs



New File Dialog

New File Dialog

Click **File|New** (or press **Ctrl+N**) to invoke the *New File* dialog. This dialog offers four kinds of file to choose from: C++ Project, Forms, Source files, and Main files.

The 'Insert Into' drop-down combobox lists the open projects, defaulting to the current project. New files are added to the project displayed in this combobox. To add a new file to a different project, choose the project you want to use in the 'Insert Into' combobox.

The 'Dialog' file type is highlighted by default when the *New File* dialog pops up. Click on the file type you want to use and click **OK** to create it. Click **Cancel** to leave the dialog without creating a new file. Note that if you select C++ Project, the 'Insert Into' combobox will be disabled, since it is not possible to insert a new C++ Project into an existing project.

C++ Project Files

Click **C++ Project** to start a new project. This option invokes the Project Settings Dialog. C++ projects are saved as `.pro` files, which include the information *Qt Designer* needs to manage projects. When you add a form to your project in *Qt Designer*, it is automatically added to the FORMS section of the project file. The `.pro` file contains the list of forms (`.ui` files) used in the project. *Qt Designer* reads and writes `.ui` files, e.g. `form.ui`. The uic (user interface compiler) creates both a header file, e.g. `form.h`, and an implementation file, e.g. `form.cpp`, from the `.ui` file.

Dialog Forms

Click **Dialog** to create a plain dialog form. Typically, this type of form is used to present the user with configuration options, or to present related sets of choices, for example, printer setting dialogs and find and replace dialogs.

Wizard Forms

Click **Wizard** to create a wizard form. A wizard is a special type of input dialog that consists of a sequence of dialog pages. A wizard's purpose is to assist a user by automating a task by walking the user through the process step by step. Wizards are useful for complex or infrequently occurring tasks that people may find difficult to learn or do. Initially the wizard form consists of a single dialog page. Use the right click context menu to add additional pages and to change page titles.

Widget Forms

Click **Widget** to create a form whose superclass is `QWidget` rather than `QDialog`.

Main Window Form

Click **Main Window** to invoke the Main Window Wizard. This wizard is used to create actions, menu options and toolbars through which the user can invoke actions. This form is used to create typical main-window style applications.

Configuration Dialog Form

Click **Configuration Dialog** creates a form with a listbox on the left, and a tabwidget filling the body of the form, along with Help, OK and Cancel buttons.

Dialog with Buttons (Bottom) Form

The **Dialog with Buttons (Bottom)** form is a template with default buttons at the bottom of the form.

Dialog with Buttons Form (Right)

The **Dialog with Buttons (Right)** form is a template with default buttons at the right of the form.

Tab Dialog Form

The **Tab Dialog** form has a tab widget as its central widget, along with Help, OK and Cancel buttons along the bottom.

C++ Source File

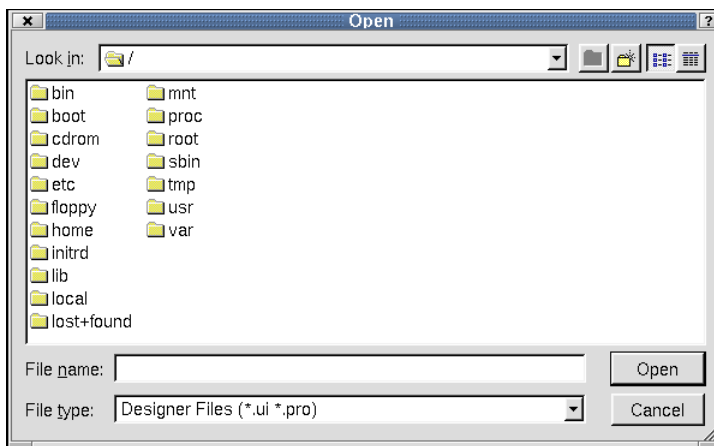
Click **C++ Source File** to create a new empty C++ file. The file will automatically be added to the project when it is saved.

C++ Header File

Click **C++ Header File** to create a new empty C++ header file. The file will automatically be added to the project when it is saved.

C++ Main File

Click **C++ Main File** to invoke the Configure Main-File Dialog which will create a basic `main.cpp` file automatically.



File Open Dialog

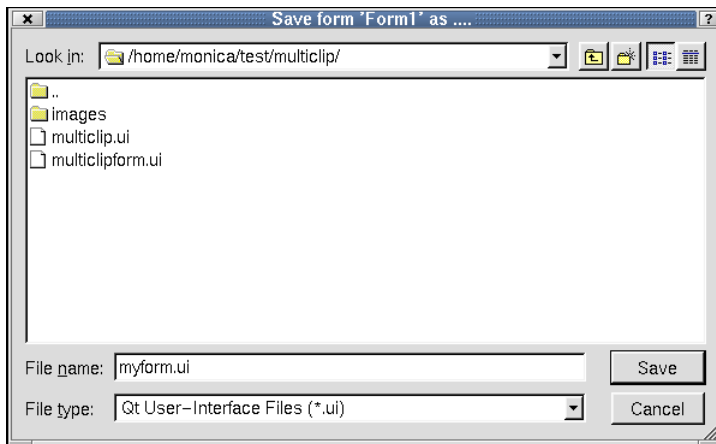
File Open Dialog

Click **File|Open** (or press **Ctrl+O**) to invoke the *Open* dialog. Use this dialog to open existing files.

The *Open* dialog shows the current directory and default file type. To choose a different directory, click the 'Look In' combobox. Choose a file and the name will appear in the 'File Name' combobox. To choose a different file type, click the 'File Type' combobox. Click the 'Create New Folder' toolbar button to create a new directory. Click the 'List View' toolbar button to view folders and files in a list with only the names showing. Click the 'Details' toolbar button to view the folders and file names along with their size, type, date, and attributes. Click the Size, Type, Date, or Attributes column headers to sort the folders or files.

Click **Open** to open the selected file. Click **Cancel** to leave the dialog without opening a new file.

Note: For Windows, the System File Dialogs are used.



Save As

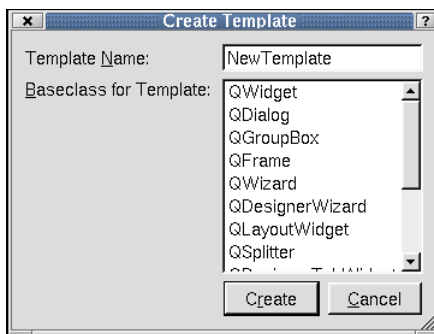
Save As

Click **File|Save As** to invoke the *Save As* dialog. Use this dialog to save files to a directory.

The *Save As* dialog shows the current directory and default file type. To choose a different directory, click the 'Look In' combobox. Choose a file and the name will appear in the 'File Name' combobox. To choose a different file type, click the 'File Type' combobox. Click the 'Create New Folder' toolbar button to create a new directory. Click the 'List View' toolbar button to view folders and files in a list with only the names showing. Click the 'Details' toolbar button to view the folders and file names along with their size, type, date, and attributes. Click the Size, Type, Date, or Attributes column headers to sort the folders or files.

Click **Save** to save the selected file. Click **Cancel** to leave the dialog without saving the file.

Note: For Windows, the System File Dialogs are used.



Create Template

Create Template Dialog

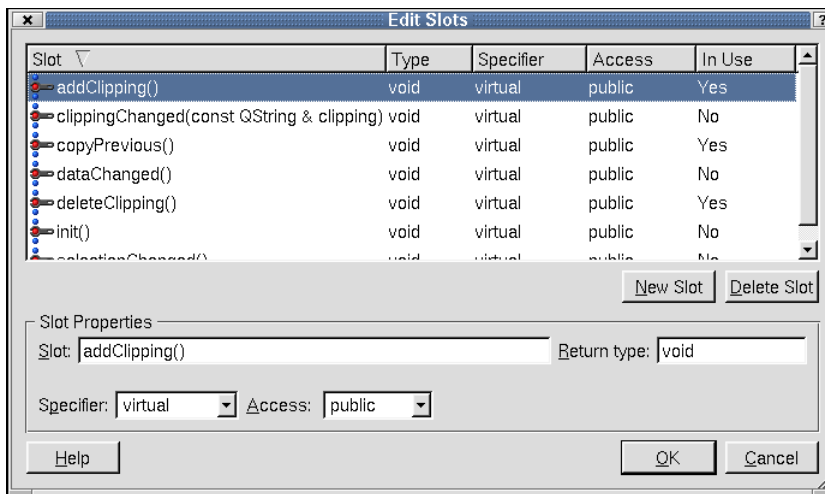
Click **File|Create Template** to invoke the *Create Template* dialog. Use this dialog to create templates.

The 'Template Name' line edit defaults to 'New Template'. To change the name to a different name, type it in the line edit. Click the 'Baseclass for Template' scroll bar to choose a base class for the template.

Click **Create** to create the template. Click **Cancel** to leave the dialog without creating a template.

If you create a template it will appear in the New File Dialog. Templates are useful when you have to produce a large number of similar forms, or where you want to 'brand' your forms.

The Edit Dialogs



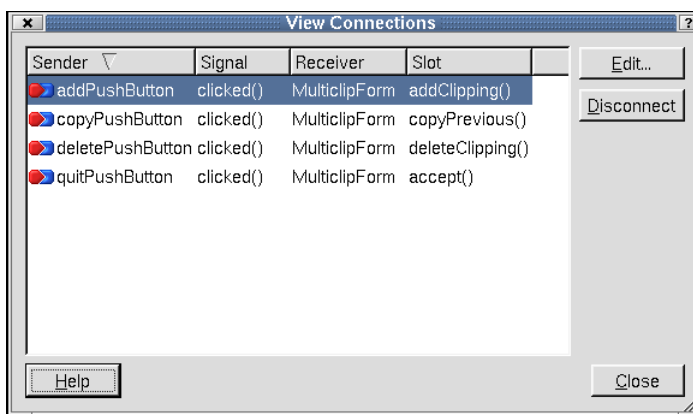
Edit Slots

Edit Slots Dialog

Click **Edit | Slots** to invoke the *Edit Slots* dialog. Use this dialog to edit or create slots which are used in conjunction with signals to provide communication between objects.

When this dialog is invoked, all existing slots are shown in the 'Slot' listbox. The column headers Slot, Type, Specifier, Access, and In-Use provide details about each slot that is listed. Click on any of the column headers to sort the slots. To create a new slot, click the **New Slot** button. The new slot has a default name that you should replace by typing the new name in the 'Slot' line edit. The 'Return Type' is also a default that can be changed by typing in the line edit. To change the 'Specifier' or 'Access', click the combobox and choose the required specifier or access. To remove a slot, click the slot you want to delete, and then click the **Delete Slot** button.

Click **OK** to save all changes made to slots. Click **Cancel** to leave the dialog without making any changes to slots.



View Connections

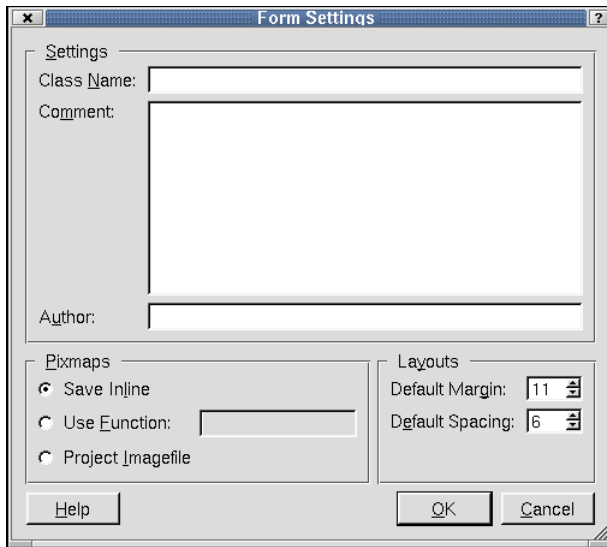
View Connections Dialog

Click **Edit | Connections** to invoke the *View Connections* dialog. This dialog displays the current signals and slots connections.

When this dialog is invoked, all existing connections are shown in the listbox. The column headers Sender, Signal, Receiver, and Slot provide details about each connection. Click the column headers to sort the connections. To

remove a connection from the listbox, click the connection you want to disconnect and then click **Disconnect**. To edit a connection, click the **Edit...** button to invoke the *Edit Connections Dialog*.

Changes made in this dialog take immediate effect. Click the **Close** button to leave the dialog.



Form Settings

Form Settings Dialog

Click **Edit | Form Settings** to invoke the *Form Settings* dialog. Use this dialog to save the form's settings, pixmap, and layout properties.

Settings

In the Settings section, you can change or add the name of the class that will be created by typing in the 'Class Name' line edit. Note that the default name is the form name, but it can be changed. You can also enter text to the 'Comment' and 'Author' line edits or leave them blank, since they are not required.

Pixmaps

The default (for projects) is 'Project Imagefile'. This is the recommended option. Images are handled automatically, with *Qt Designer* storing the images in a subdirectory, and *uic* producing code that contains the images and the necessary supporting code. Each image is stored just once, no matter how many forms it is used in.

If you do not want *Qt Designer* to handle the images, (or are not using a project) choose either 'Save Inline' or 'Use Function'. 'Save Inline' saves the pixmaps in the `.ui` files. The disadvantage of this approach is that it stores images in the forms in which they're used, meaning that images cannot be shared across forms. Click 'Use Function' to use your own icon-loader function for loading pixmaps. Type the function's name (with no signature) in the 'Use Function' line edit. This function will be used in the generated code for loading pixmaps. Your function will be called with the text you put in the pixmap property (e.g. the image name) whenever an image is required.

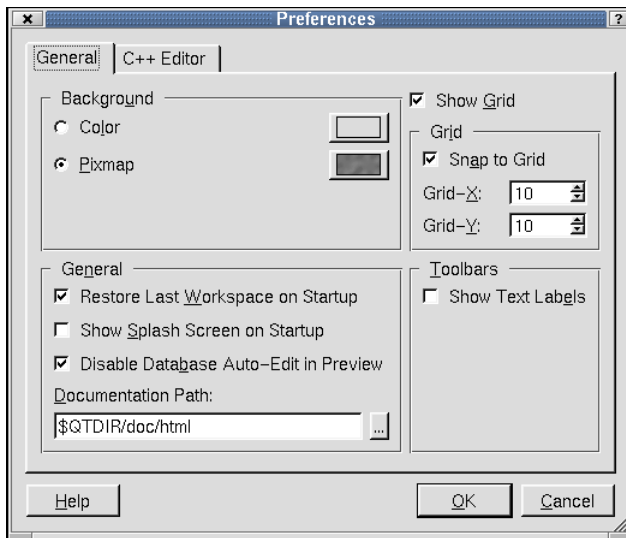
Layouts

Click the 'Default Margin' spinbox or the 'Default Spacing' spinbox to change the default layout of the current form.

Click **OK** to accept changes to the form settings. Click **Cancel** to leave the dialog without making any changes.

Preferences Dialog

Click **Edit | Preferences** to invoke the *Preferences* dialog. This dialog has a tab for 'General' preferences. If you have the C++ Editor plugin, the dialog will also have a tab for the C++ Editor.



Preferences- General Tab

General Tab

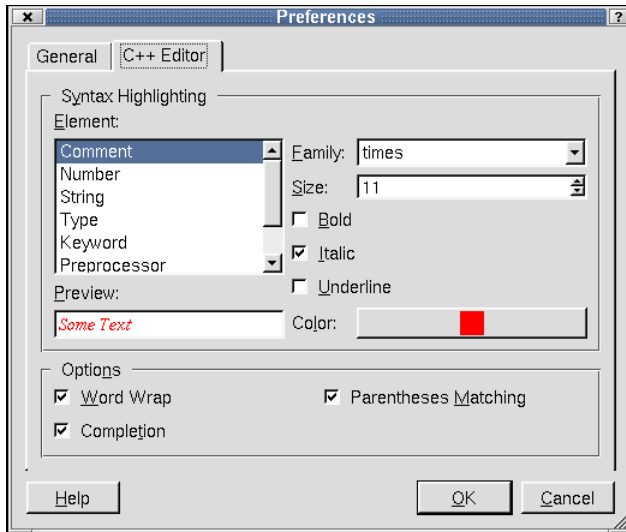
The 'General' tab has sections for Background, Grid, General, and Toolbars.

The Background section defaults to 'Pixmap'. To change the default, click the **Select a Pixmap** button next to the 'Pixmap' radio button to invoke the *Choose a Pixmap... Dialog*. Click the 'Color' radio button to change the background to a color instead of a pixmap. Click the **Choose a Color** button located to the right of the 'Color' radio button to invoke the *Select Color Dialog*.

The 'General' section of the General tab has three checkboxes that are checked by default. Click the 'Restore last workspace on startup' checkbox to save the size and positions of the windows and toolbars of *Qt Designer*. The next time you start up *Qt Designer*, the windows and toolbars are restored to their last positions. Click the 'Show Splash Screen on startup' checkbox to display the *Qt Designer* splash screen when you start up the application. Click the 'Disable Database Auto-Edit in Preview' checkbox to disable the ability to update or delete data in the database to which you are connected when working with database widgets. To change the path *Qt Designer* uses to find its online documentation, click the 'Documentation Path' line edit and type a new path. It would be unlikely to have to change this path. Another way to change the path is to click the (**ellipsis**) button located to the right of the line edit. This invokes the *Find Directory Dialog*.

The 'Grid' section has options for customizing the grid on the form. The 'Show Grid' checkbox located above the 'Grid' section is checked by default. Developers using *Qt Designer* almost always use Qt's layouts to design their forms and rarely make any use of the grid. The grid is provided for the rare occasions when a form is created using widgets with fixed sizes and positions. When 'Show Grid' is checked, you can customize the grid's appearance. When it is unchecked, the 'Grid' section is disabled. The 'Snap to Grid' checkbox is also checked by default. When it is checked, widgets are placed on a dot (snap to the grid) using the X|Y resolution. When it is unchecked, the 'Grid-X' and 'Grid-Y' spin boxes are disabled. Click the 'Grid X' and 'Grid Y' spinboxes to customize the grid settings for all forms.

The 'Toolbars' section has a 'Show Text Labels' checkbox. Click the checkbox to display the text labels for each icon shown in the *Qt Designer* toolbar.



Preferences- C++ Editor Tab

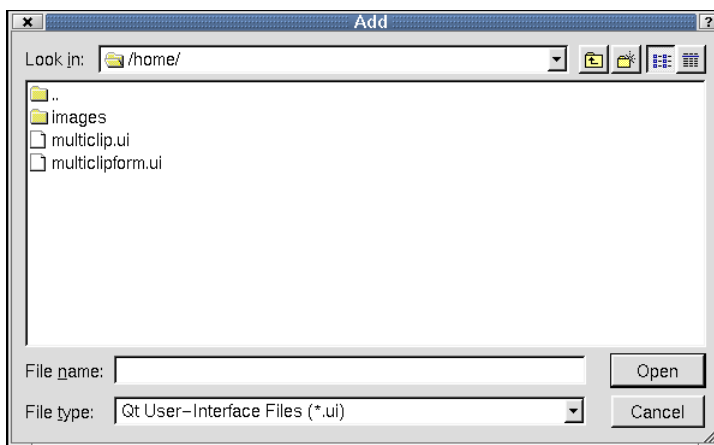
C++ Editor Tab

The C++ Editor tab provides options for customizing the editor. The 'Syntax Highlighting' section lets you change the way the syntax is viewed in the editor. Click the 'Element' listbox and choose an element. Click the 'Family' listbox to change the font style for that element. Click the 'Size' spinbox to choose a font size. You can change the font to Bold, Italic, or Underline by clicking the corresponding checkbox. Note, that all the fonts used derive from the 'Standard' element, so if you want to change the font used for everything, change the 'Standard' element. Click the **Color** button to invoke the *Select Color Dialog*. As you make changes to each element, you can view the changes in the 'Preview' line edit.

The 'Options' section has the Wordwrap, Completion, and Parentheses Matching checkboxes checked by default. Click the checkboxes to de-select them.

Click **OK** to accept changes to *Preferences* dialog. Click **Cancel** to leave the dialog without making any changes.

The Project Dialogs



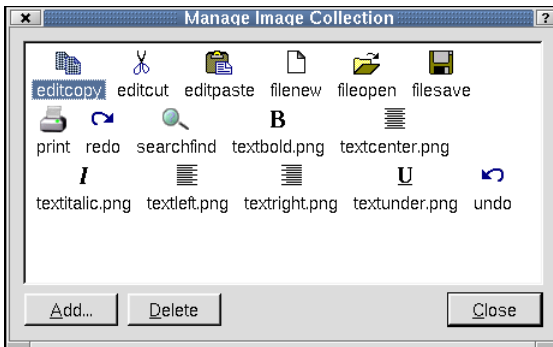
Add Dialog

Add Dialog

Click **Project|Add File** to invoke the *Add* dialog. Use this dialog to add files to the current project.

The *Add* dialog defaults the directory and file type. To choose a different directory, click the 'Look In' combobox. Choose a file and the name will appear in the 'File Name' combobox. To choose a different file type, click the 'File Type' combobox. Click the 'Create New Folder' toolbar button to create a new directory. Click the 'List View' toolbar button to view folders and files in a list with only the names showing. Click the 'Details' toolbar button to view the folders and file names along with their size, type, date, and attributes. Click the Size, Type, Date, or Attributes column headers to sort the folders or files.

Click **Open** to open the selected file. Click **Cancel** to leave the dialog without opening a file.



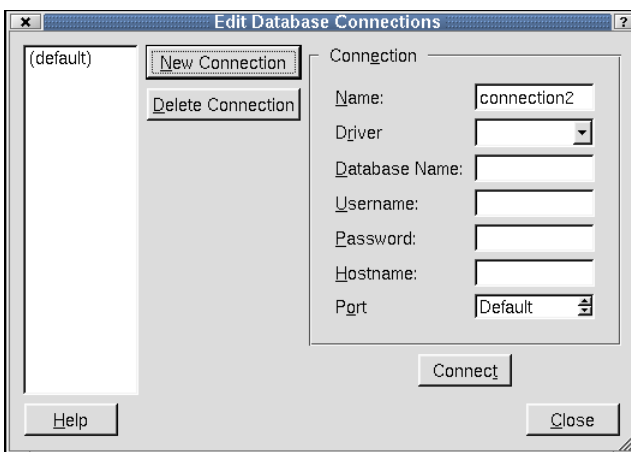
Manage Image Collection

Image Collection Dialog

Click **Project|Image Collection** to invoke the *Manage Image Collection Dialog*. Use this dialog to view the project's images, add new images, or delete images.

To add an image, click the **Add** button to invoke the *Choose Images... Dialog*. To delete an image from from the iconview, click the image and then click the **Delete** button.

Changes made to the image collection are applied immediately. Click the **Close** button to leave the dialog.



Edit Database Connections

Edit Database Connections Dialog

Click **Project|Database Connections** to invoke the *Edit Database Connections Dialog*. Use this dialog to connect your project to a database or to edit the current connections.

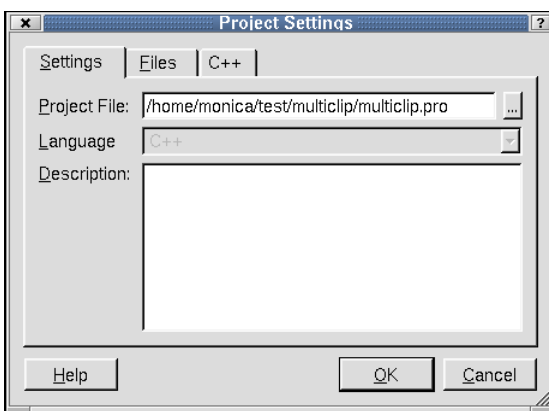
Click **New Connection** to create a new database connection. For applications that use a single database it will probably be most convenient to use the default connection name of '(default)'. If you use more than one database then each one must be given a unique name. A driver must be chosen from the Driver combo box. The database name may be available in the Database Name combo box or may have to be typed in. The database name, user-name, password and hostname should be provided by your database system administrator. When the Connection information has been completed click Connect. If the connection is made the connection name will appear in the list box on the left hand side of the dialog.

To remove a connection, click the connection in the listbox and then click the **Delete Connection** button.

Click **Close** to leave the *Database Connections* dialog.

Project Settings Dialog

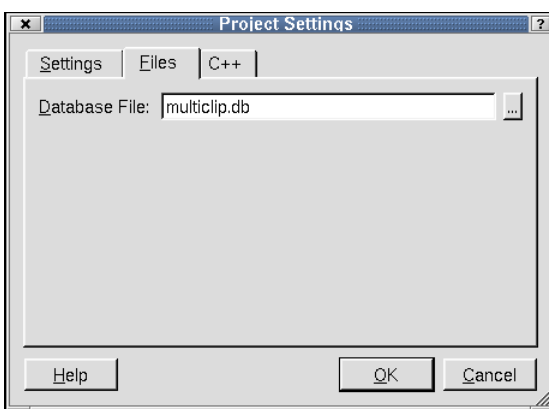
Click **Project|Project Settings** to invoke the *Project Settings Dialog*. Use this dialog to make changes to the project settings.



Project Settings- Settings Tab

Settings Tab

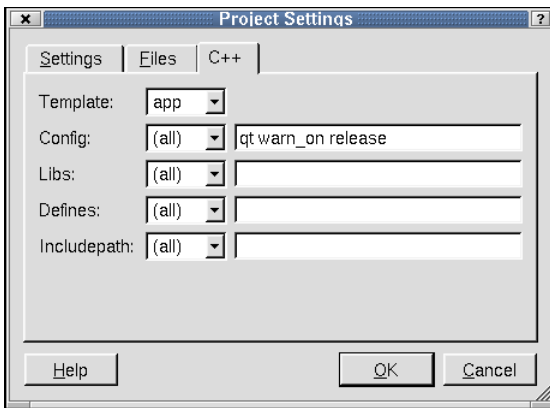
The 'Settings' tab shows information about the project. The Project File line edit defaults the project name. To change the name, type a new name in the line edit. To save the project, click the **(ellipsis)** button located next to Project File to invoke the *Save As Dialog*. The 'Language' combobox is disabled. Click the 'Description' line edit if you want to add additional information about the project.



Project Settings- Files Tab

Files Tab

Click the 'Files' tab to enter a name in the 'Database File' line edit. Click the (**ellipsis**) button to invoke the *Save As Dialog*.



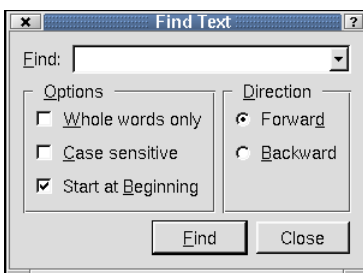
Project Settings- C++ Tab

C++ Tab

Click the C++ Tab to change the `qmake` options. See the `qmake` documentation for details on what these options mean. Click the 'Template' combobox and choose application or library to create makefiles for building applications or libraries. Click the 'Config' combobox to select the project configuration and compiler options for all platforms, or specific platforms. Type the Config value in the line edit. Click the 'Libs' combobox to select a platform. Type the libraries in the line edit. Click the 'Defines' combobox and select a platform. 'Defines' values are added as compiler pre-processor macros. Type the 'Defines' values in the line edit. Click the 'Includepath' combobox to select a platform. Includepath specifies the directories that should be searched for include files when compiling the project. Type the 'Includepath' values in the line edit.

Click **OK** to accept changes to the project settings. Click **Cancel** to exit the dialog without making any changes to the project settings.

The Search Dialogs



Find Text

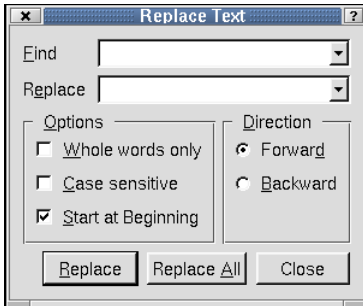
Find Text Dialog

Click **Search|Find** (or press **Ctrl+F**) to invoke the Find Text Dialog. Use this dialog to find specific text in a project file.

To find the text you want in a file, type the text in the 'Find' combobox. You can make the search more specific by checking any or all of the checkboxes in the 'Options' section. Click the 'Whole words only' checkbox to narrow the

search to whole words. Click 'Case Sensitive' to search for text that is identical to the text typed in the combobox. Click 'Start at Beginning' to start the search from the beginning of the file. The 'Direction' section offers the 'Forward' radio button and the 'Backward' radio button to specify the direction to perform the search in the file. Click the **Find** button to start the search. When the text is found, it is highlighted in the file. Continue clicking **Find** to search for subsequent occurrences of the search text.

Click the **Close** button to leave the dialog.



Replace Text

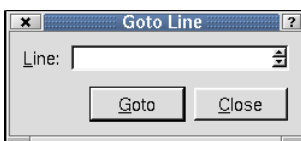
Replace Text Dialog

Click **Search|Replace** (or press **Ctrl+R**) to invoke the *Replace Text Dialog*. Use this dialog to replace text in a project file.

To replace text, type the text you would like to replace in the 'Find' combobox. Type the new text in the 'Replace' combobox. You can make the search more specific by checking any or all of the checkboxes in the 'Options' section. Click the 'Whole words only' checkbox to narrow the search to whole words. Click 'Case Sensitive' to search for text that identical to the text you typed in the combobox. Click 'Start at Beginning' to start the search from the beginning of the file. The 'Direction' section offers the 'Forward' radio button and the 'Backward' radio button to specify the direction to perform the search in the file.

Click the **Replace** button to search and replace the text. When the text is found, it is highlighted in the file. Continue clicking **Replace** button to search and replace each occurrence of the text in the file. Click **Replace All** button to replace all occurrences of the search text in the file at once.

Click the **Close** button to leave the dialog.



Goto Line

Goto Line Dialog

Click **Search|Goto line** (or press **Alt+G**) to invoke the *Goto Line Dialog*. Use this dialog to go to a specific line in the file.

To choose a line number, type the number in the 'Line' spinbox, or click the up and down arrows in the spinbox. Click the **Goto** button. The cursor is placed at the beginning of the line in the file.

Click the **Close** button to leave the dialog.

The Help Dialogs

Qt Designer Dialog

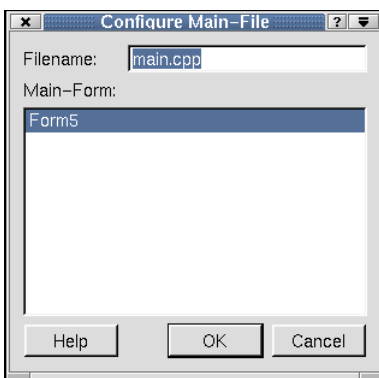
Click **Help|About...** to invoke the *Qt Designer Dialog*. This dialog provides information about *Qt Designer* such as the version, the licensing terms, conditions, and disclaimers.

Click the 'x' located at the top right corner of the dialog to close the dialog.

Qt Designer Dialog

Click **Help|About Qt...** to invoke the *Qt Designer Dialog*. This dialog provides information about Qt.

Click the 'x' located at the title of the dialog to close the dialog.



Configure Main-File Dialog

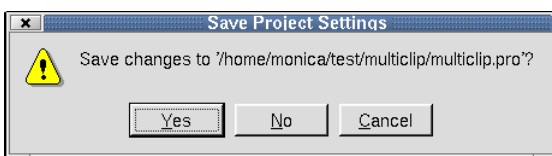
Configure Main-File Dialog

Click **File|New|C++ Main-File** to invoke the *Configure Main-File* dialog. Use this dialog to configure the main file and its forms.

To change the default file name, type it in the 'Filename' line edit. Choose the form to use as the application's main form from the line edit by clicking it.

Click **OK** to accept the configurations and *Qt Designer* will create a default `main.cpp` file. Click **Cancel** to leave the dialog.

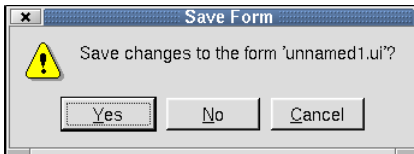
Note for database programmers: If you create a `main.cpp` file using *Qt Designer*, this file will *not* include the `createConnections()` function. We do not include this function because it needs the username and password for the database connection, and you may prefer to handle these differently from our simple example function. As a result, applications that preview correctly in *Qt Designer* will not run unless you implement your own database connections function.



Save Project Settings

Save Project Settings Dialog

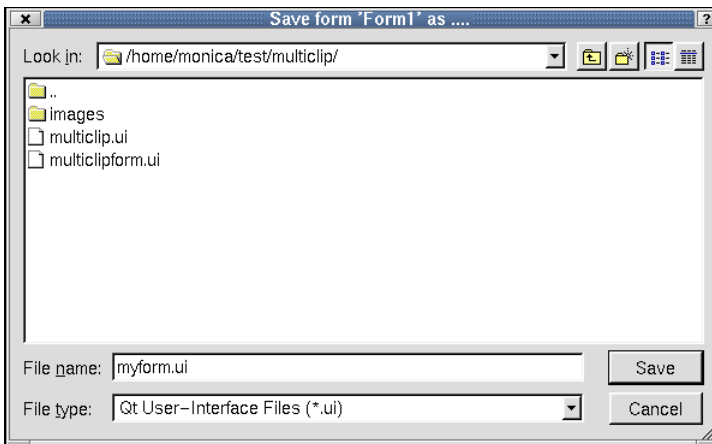
The *Save Project Settings* message box is invoked by clicking **File|Close** or **File|Exit** for an open project with unsaved changes. The dialog displays the text 'Save changes to your project.pro?'. Click **Yes** to save the changes. If the project has any forms with unsaved changes, the *Save Form As* dialog is invoked when you click **Yes**. Click **No** to close the project without saving any changes. Click **Cancel** to leave the dialog without closing the project and without making any changes.



Save Form

Save Form Dialog

The *Save Form* message is invoked in several ways. One way is to click **File|Close** for a form that has never been saved, or has been saved previously but has had changes made to it. The dialog is also invoked by clicking **File|Exit** for a form that has never been saved, or has been saved previously but has had changes made to it. The dialog displays 'Save Changes to the Form?'. Click **Yes** to save the form. If the form has not been previously saved, the *Save Form As Dialog* is invoked. Click **No** to close the form without saving any changes or without saving the form if it has not been saved previously. Click **Cancel** to leave the dialog without closing or exiting the form and without saving the form.



Save Form As Dialog

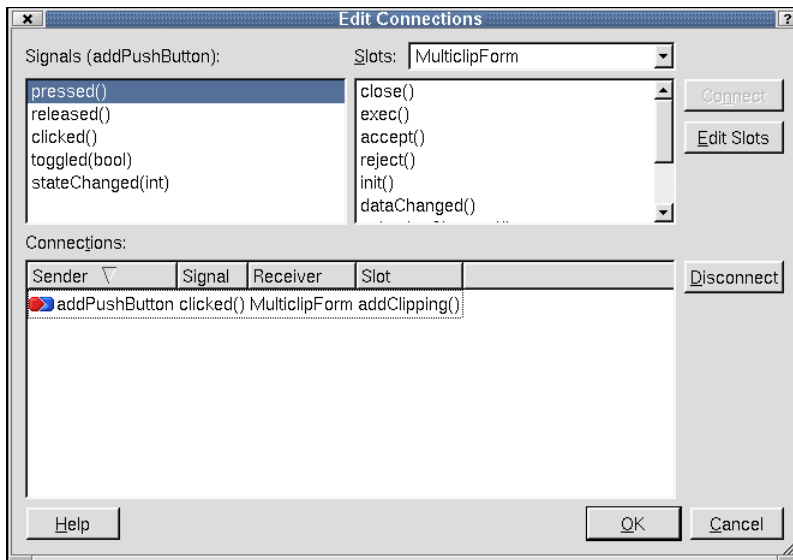
Save Form As Dialog

The *Save Form As* dialog is invoked in three different ways. One way is to click **File|Save** for a form in a project that has never saved. Another way to invoke the dialog is by clicking **File|Close** for a form that has not been previously saved and has had changes made to it. The third way to invoke the dialog is by clicking **File|Exit** for a form that has not been saved previously or that has been saved but has had changes made to it. **File|Close** and **File|Exit** invoke the *Save Form Dialog*. Click **Yes** to invoke the *Save Form As* dialog.

The *Save Form As* dialog shows the current directory and the default file type. To choose a different directory, click the 'Look In' combobox. Choose a file and the name will appear in the 'File Name' combobox. To choose a different file type, click the 'File Type' combobox. Click the 'Create New Folder' toolbar button to create a new directory. Click the 'List View' toolbar button to view folders and files in a list with only the names showing. Click the 'Details'

toolbar button to view the folders and file names along with their size, type, date, and attributes. Click the Size, Type, Date, or Attributes column headers to sort the folders or files.

Click **Save** to save the selected form. Click **Cancel** to leave the dialog without saving the form.



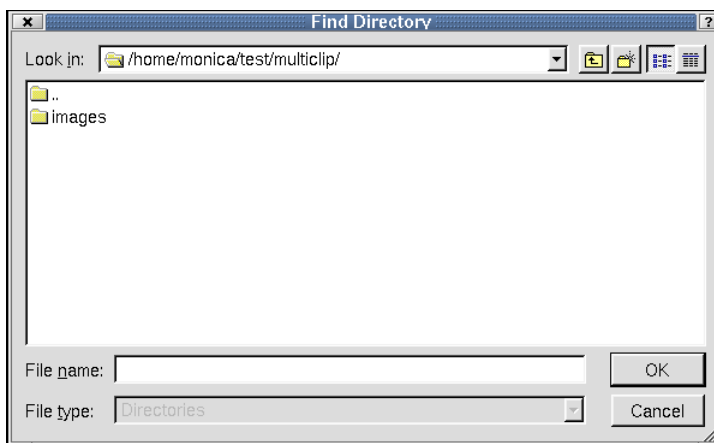
Edit Connections

Edit Connections Dialog

Invoke the *Edit Connections* dialog to modify connections between signals and slots.

The top left hand listbox displays the Signals that the widget can emit. The top right combobox lists the form and its widgets. Beneath the combobox is the 'Slots' listbox which shows the slots available in the form or widget displayed in the 'Slots' combobox which are compatible with the highlighted signal. To connect a signal to a slot, choose a signal from the 'Signals' listbox by clicking on it. Then choose a form or widget from the 'Slots' combobox. Choose a slot for the widget or form you select from the listbox. The **Connect** button will flash and the new connection will appear in the 'Connections' listbox, along with any existing connections. Click the column headers Sender, Signal, Receiver, or Slot to sort the connections. To disconnect an existing connection, choose the connection from the 'Connections' listbox and click the **Disconnect** button. Click **Edit Slots** to invoke the *Edit Slots Dialog*.

Click **OK** to accept changes to the connections. Click **Cancel** to leave the dialog without making changes to the connections.



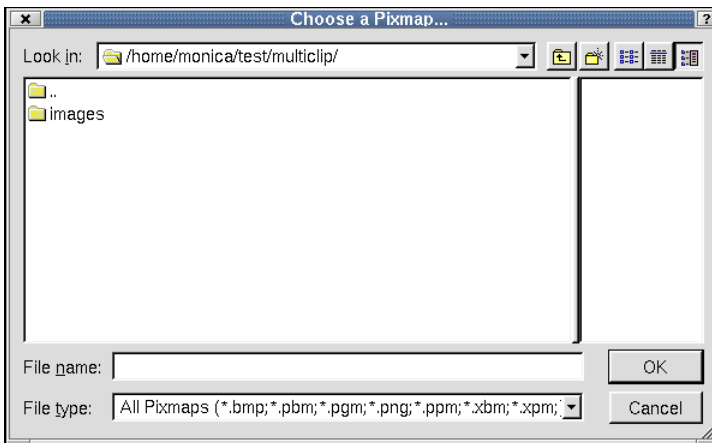
Find Directory

Find Directory Dialog

Invoke this dialog to locate a directory.

The *Find Directory* dialog shows the current directory and the default file type. To choose a different directory, click the 'Look In' combobox. Choose a file and the name will appear in the 'File Name' combobox. To choose a different file type, click the 'File Type' combobox. Click the 'Create New Folder' toolbar button to create a new directory. Click the 'List View' toolbar button to view folders and files in a list with only the names showing. Click the 'Details' toolbar button to view the folders and file names along with their size, type, date, and attributes. Click the Size, Type, Date, or Attributes column headers to sort the folders or files.

Click **OK** to accept the directory. Click **Cancel** to leave the dialog without choosing a directory.



Choose a Pixmap

Choose a Pixmap Dialog

Invoke this dialog to select a pixmap to use in the current project.

The *Choose a Pixmap* dialog shows the current directory and the default file type. To choose a different directory, click the 'Look In' combobox. Choose a file and the name will appear in the 'File Name' combobox. To choose a different file type, click the 'File Type' combobox. Click the 'Create New Folder' toolbar button to create a new directory. Click the 'List View' toolbar button to view folders and files in a list with only the names showing. Click the 'Details' toolbar button to view the folders and file names along with their size, type, date, and attributes. Click the Size, Type, Date, or Attributes column headers to sort the folders or files. View a sample of the pixmap file you select in the preview box located on the right side of the dialog.

Click **OK** to accept the pixmap file. Click **Cancel** to leave the dialog without choosing a pixmap file.

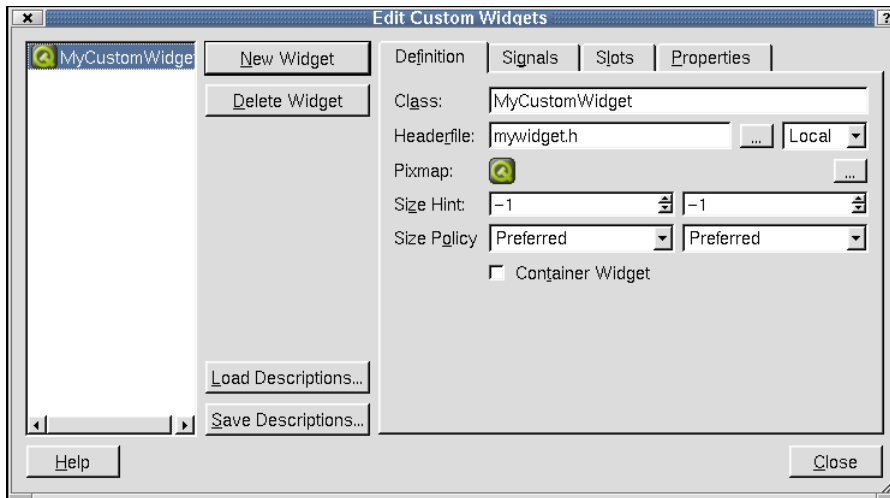
Edit Custom Widgets Dialog

Invoke this dialog by clicking **Tools|Custom|Edit Custom Widgets**. Use this dialog to create custom widgets.

Custom widgets are created in code. They may contain a combination of existing widgets but with additional functionality, slots and signals, or they may be written from scratch, or a mixture of both. A custom widget is often a specialization (subclass) of another widget or a combination of widgets working together or a blend of both these approaches. If you simply want a collection of widgets in a particular configuration it is easiest to create them, select them as a group, and copy and paste them as required within *Qt Designer*. Custom widgets are generally created when you need to add new functionality to existing widgets or groups of widgets. To add create a new widget, click the **New Widget** button. You will find more information about adding new widgets in the 'Definitions Section'. To load a file which contains descriptions of custom widgets, click the **Load Descriptions**

button. Clicking this button invokes the *Open Dialog*. To save the descriptions of the listed custom widgets, click the **Save Descriptions** button, which invokes the *Save As Dialog*. To delete a widget, click the widget in the listbox and then click the **Delete Widget** button.

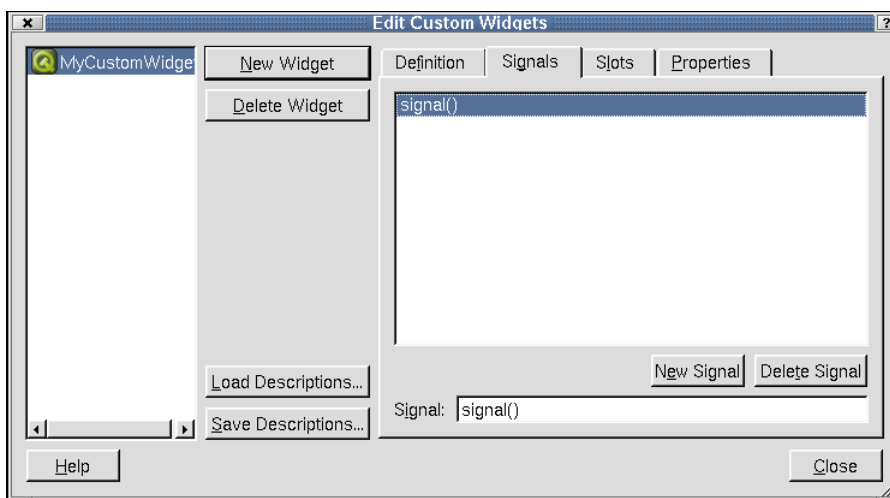
Click **Close** to leave the Edit Custom Widgets dialog.



Edit Custom Widgets- Definition Tab

The Definition Tab

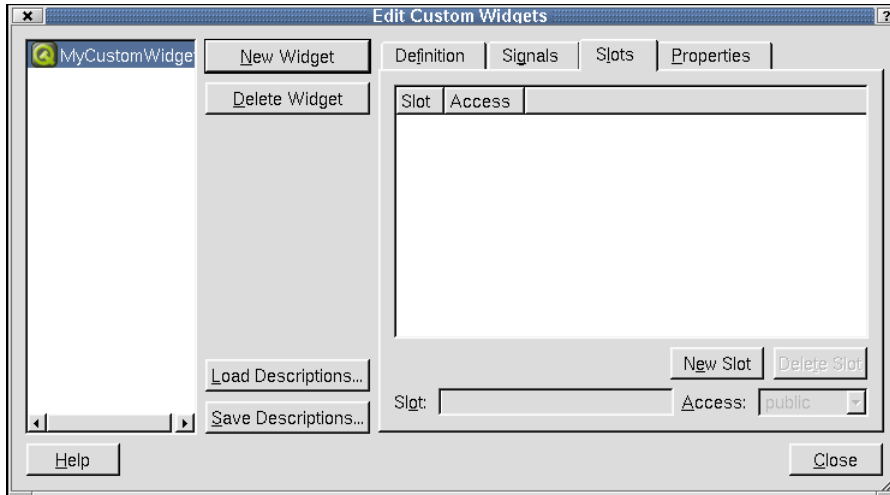
To create a custom widget, click **New Widget**. Click the Definition tab if you are not already there. You should change the 'Class' name from 'MyCustomWidget' to a unique name by typing in the line edit. Type in the 'Headerfile' line edit to change the name or type the name of a header file you want to use. To search for a saved header file in a directory, click the **(ellipsis)** button to the right of the Headerfile line edit to invoke the *Open Dialog*. Click the 'Select Access' combobox to choose how the file will be included. Global include files will be included using angle brackets (<>). Local files will be included using quotation marks. If you have a pixmap that you want to use to identify your widget on the toolbar, click the **(ellipsis)** button to the right of the 'Pixmap' label. This invokes the *Choose a Pixmap Dialog*. Click the 'Size Hint' spin boxes to select the recommended size for the widget. If you do not want to have a recommended size, enter -1/-1 in the spinboxes. Click the 'Size Policy' comboboxes to select the vertical size properties of the widget. Click the 'Container Widget' checkbox if the custom widget you are creating should be able to contain other widgets (children).



Edit Custom Widgets- Signals Tab

The Signals Tab

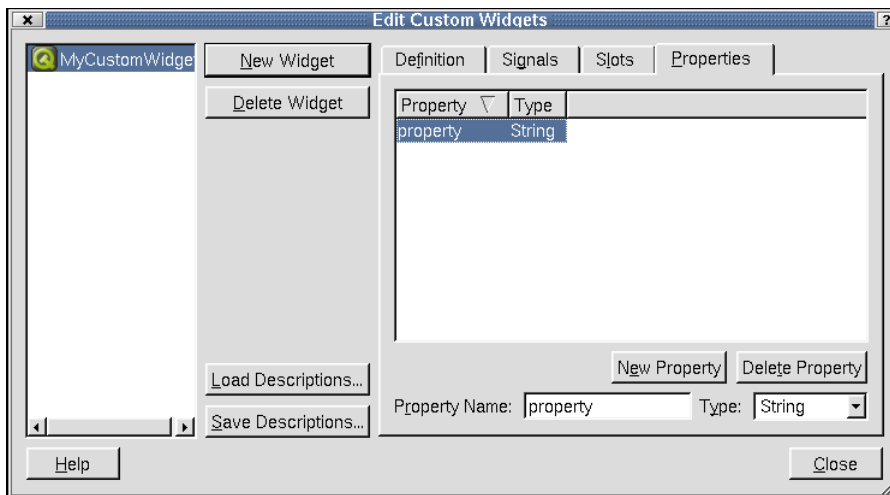
Click the Signals tab to view a list of all the signals the selected custom widget can emit. To add a new signal, click the **New Signal** button. Click the 'Signal' line edit and provide an argument for the signal and give the signal a unique name. To delete a signal from the listbox, click the signal to choose it and then click the **Delete Signal** button.



Edit Custom Widgets- Slots Tab

The Slots Tab

Click the Slots tab to view a list of all the slots for the selected custom widget. Click the 'Slot' or 'Access' column headers to sort the slots in the listbox. To add a slot, click the **New Slot** button. Click the 'Slot' line edit and provide an argument for the slot and give the slot a unique name. Click the 'Access' combobox to choose between public or protected access for your widget. To delete a slot from the listbox, click the slot and then click **Delete Slot**.



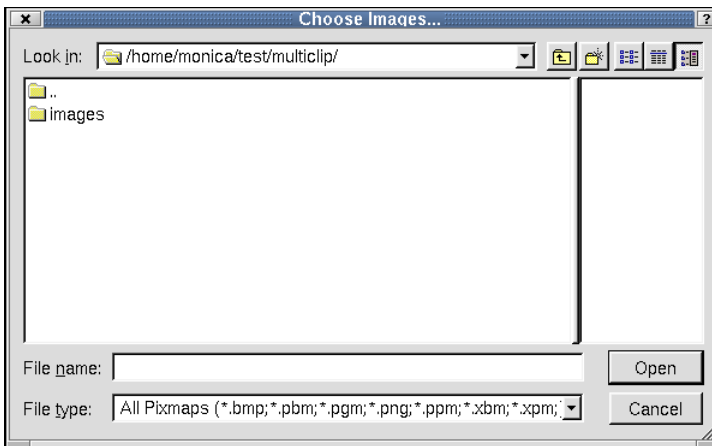
Edit Custom Widgets- Properties Tab

The Properties Tab

Click the Properties tab to view the list of properties for the selected widget. Click the 'Property' or 'Type' column headers to sort the properties in the listbox. To add a property, click the **New Property** button. Click the 'Property Name' line edit if you want to change the default name of the property. Note that properties must be implemented

in the class using the property system of Qt. To choose a property type, click the 'Type' combobox. To delete a property from the listbox, click the property and then click the **Delete Property** button.

Click **Close** to leave the *Edit Custom Widgets* dialog.



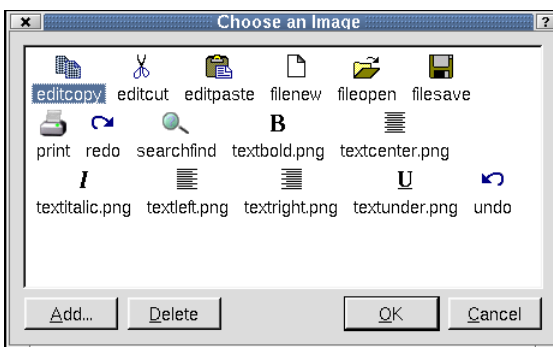
Choose Images

Choose Images Dialog

Invoke the *Choose Images* dialog to choose images to use in a project.

This dialog shows the current the directory and the default file type. To choose a different directory, click the 'Look In' combobox. Choose a file and the name will appear in the 'File Name' combobox. To choose a different file type, click the 'File Type' combobox. As you choose different files, you can preview the images in the window located on the right side of the dialog. Click the 'Create New Folder' toolbar button to create a new directory. Click the 'List View' toolbar button to view folders and files in a list with only the names showing. Click the 'Details' toolbar button to view the folders and file names along with their size, type, date, and attributes. Click the Size, Type, Date, or Attributes column headers to sort the folders or files.

Click **Open** to open the selected file. Click **Cancel** to leave the dialog without opening a file.



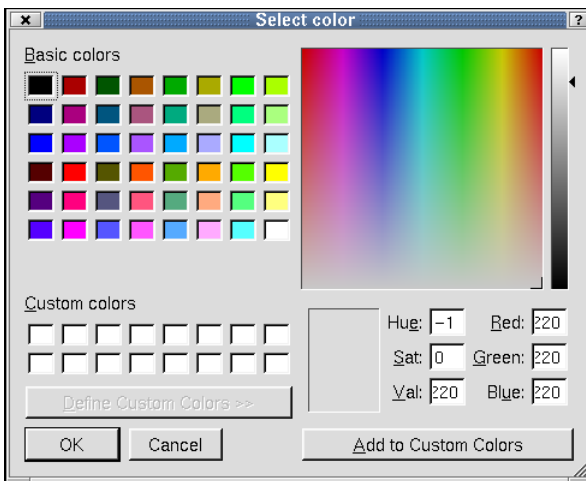
Choose an Image

Choose an Image Dialog

The *Choose an Image* dialog is used to choose an image to use for a widget.

To choose an image from the listbox, click the image and then click **OK**. To add an image, click the **Add** button to invoke the *Choose Images... Dialog*. To delete an image, click the image in the listbox and then click the **Delete** button.

Click **Cancel** to leave the dialog without making any changes to images.



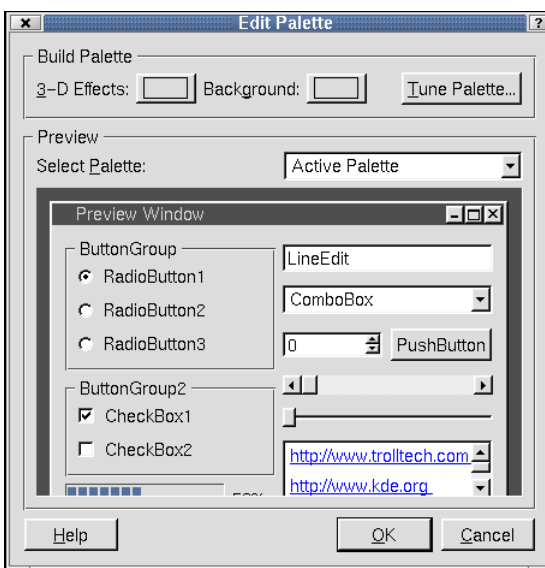
Select Color

Select Color Dialog

The *Select Color* dialog is used to select color preferences or to create color palettes.

Choose a color from the 'Basic Colors' section and a sample of the color will appear in the small preview box at the bottom of the dialog. To the right of the color sample, you will see line edits that have information about the location of the color in the color spectrum. In addition, the crosshairs in the larger color spectrum window show the location of the color. You can also create a palette of custom colors. There are two ways to do add custom colors. Click a color on the color spectrum window. When the color appears in the small box below the window, click the color and drag it to one of the blank boxes in the 'Custom Color' section of the dialog. You can also click and drag colors from the 'Basic Colors' section. Another way to add colors is to click the **Add to Custom Colors** when you have chosen a color.

Click **OK** to accept changes to the *Select Color* dialog. Click **Cancel** to exit the dialog without selecting a color or adding custom colors.



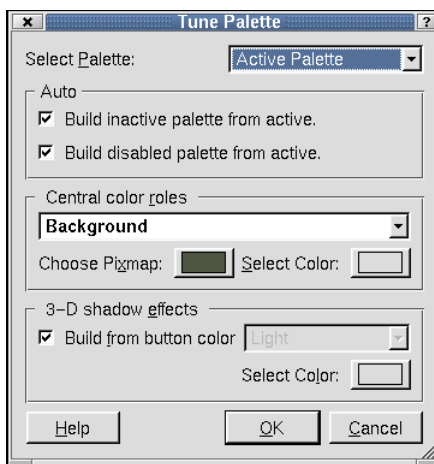
Edit Palette

Edit Palette Dialog

The *Edit Palette* dialog is used to change the palette of the current widget or form. You can use a generated palette, or select colors for each color group and each color role. The palette can be tested with different widget layouts in the preview section.

The 'Build Palette' section contains three buttons to help you build the palette. Click the **3-D Effects** button to invoke the *Select Color Dialog*. Click the **Background** to invoke the *Select Color Dialog*. Click the **Tune Palette** button to invoke the *Tune Palette Dialog*. Click the 'Select Palette' combobox in the 'Preview' section to choose a palette to preview.

Click **OK** to accept the changes to the palette. Click **Cancel** to leave the dialog without making changes to the palette.



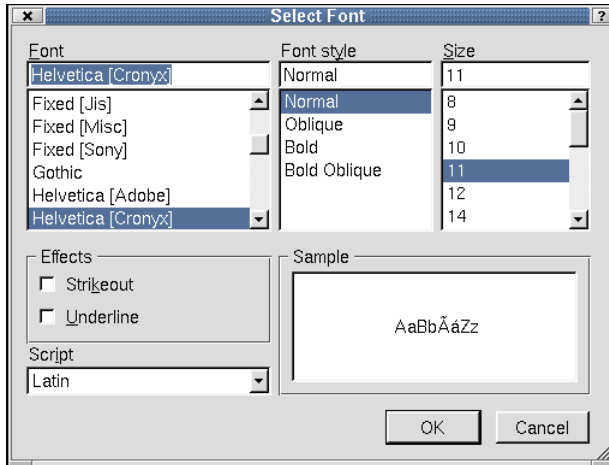
Tune Palette

Tune Palette Dialog

The *Tune Palette* dialog is used to choose options for a widget's palette.

Click the 'Select Palette' combobox to choose options for active, inactive, or disabled palettes. If you choose 'Active Palette', the dialog presents three categories used for designing the palette. The categories are the Auto, Central Color Roles, and 3-D Shadow Effects. If you choose 'Inactive Palette' or 'Disabled Palette', all categories are disabled except 'Auto'. Click the 'Auto' section checkboxes to build the inactive or disabled palettes from the active palette. For an active palette, click the 'Central color roles' combobox to select a color role for the palette. Click the **Choose Pixmap** button to invoke the *Choose a Pixmap Dialog*. Click the **Select Color** button to invoke the *Select Color Dialog*. Check the 'Build from button color' checkbox in the '3-D shadow effects' section to allow 3-D effects colors to be calculated from the button color. Uncheck the checkbox to enable the 'Choose 3-D effect color role' combobox. Click the combobox to select a color role for the 3-D effects. Click the **Select Color** button to invoke the *Select Color Dialog*.

Click **OK** to accept changes to the palette. Click **Cancel** to leave the dialog without making changes to the palette.



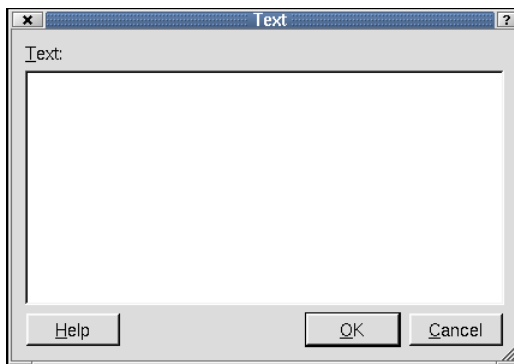
Select Font

Select Font Dialog

The *Select Font* dialog is used to make changes to the font size and style.

Click the 'Font' listbox to choose a font type. The current selected type appears in the line edit above the 'Font' listbox. Click the 'Font Style' listbox to choose a style for the font. The choices available in the listbox are limited to the type of font you choose. Not all fonts have all styles available. The selected style appears in the line edit above the 'Font Style' listbox. Click the 'Size' listbox to choose a size for the font. The current selected size appears in the line edit above the 'Size' line edit. Click the checkboxes in the 'Effects' section to create a 'Strikeout' or 'Underline' effect for the selected font. Click the 'Script' and choose a style of writing. View your font selections and styles in the 'Sample' listbox.

Click **OK** to accept changes to the font. Click **Cancel** to leave the dialog without making any changes to the font.

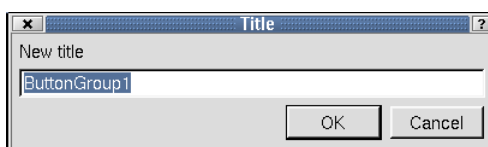


Text Dialog

Text Dialog

The *Text* dialog is used to type text.

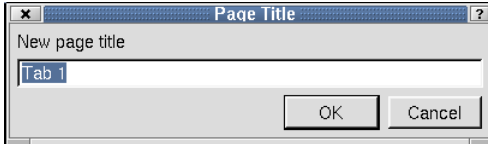
Click **OK** to accept the text. Click **Cancel** to leave the dialog without saving any text.



*Title Dialog***Title Dialog**

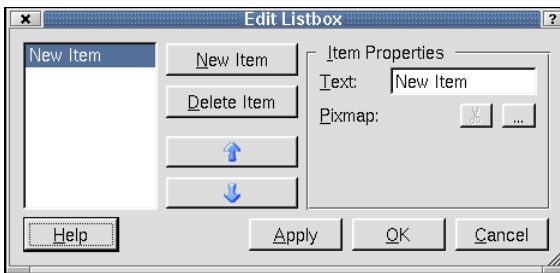
Use this dialog to change the title of a selected widget by typing the new title in the line edit.

Click **OK** to accept changes to the title. Click **Cancel** to leave the dialog without making changes to the title.

*Page Title Dialog***Page Title Dialog**

Right click a tab widget on the form and select **Edit Page Title** to invoke the *Page Title* dialog. Use this dialog to change the name of each tab in the Tab widget.

Click **OK** to accept new page titles. Click **Cancel** to leave the dialog without making any changes.

*Edit Listbox***Edit Listbox Dialog**

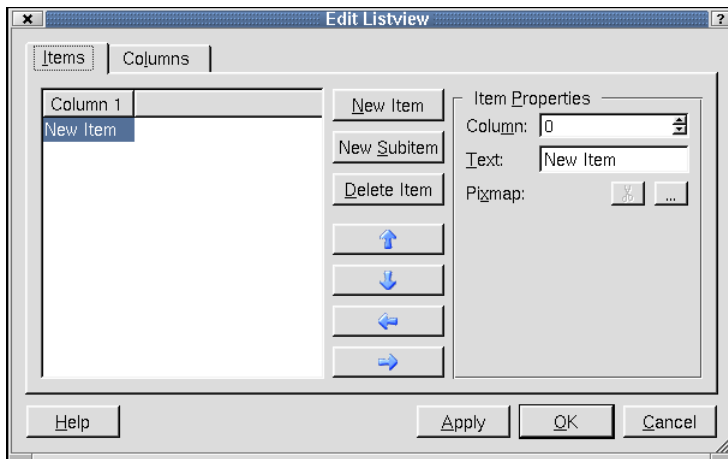
Right click or double click a Listbox on the form and select 'Edit' to invoke the *Edit Listbox* dialog. Use this dialog to add items to the list box and to change the item's properties.

To add an item to the listbox, click the **New Item**. If you want to change the default name of the item, click the 'Text' line edit in the 'Item Properties' section and type a new name for the item. Click the **Select a Pixmap** to invoke the *Choose an Image Dialog*. Click a pixmap and then click the **Delete Pixmap** button to delete the selected pixmap. To delete an item from the listbox, click the item and then click the **Delete** button. To move an item up or down in the listbox, click the **Move Up** or **Move Down** buttons. Click **Apply** to accept the changes.

Click **Apply** to accept changes to the listbox widget. Click **OK** to leave the dialog once the changes have been accepted. Click **Cancel** to leave the dialog without saving any changes.

Edit Listview

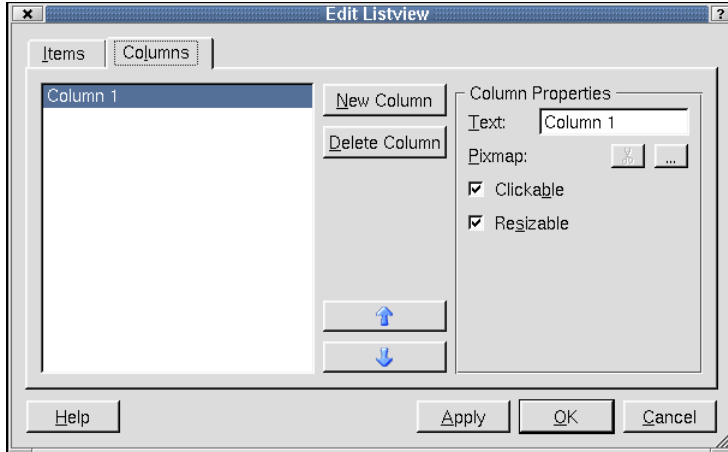
Right click or double click a listview widget on the form and select 'Edit' to invoke the *Edit Listview* dialog. Use this dialog to add items to the listview. The *Edit Listview* dialog has two tabs, one for items and one for columns.



Edit Listview- Items Tab

The Items Tab

The dialog defaults to the Items tab. Use this tab to add, change, or remove items in the listview. To add a new item, click the **New Item** button. The new item is shown at the top of the listbox. To add sub-items to an existing item, click the item and then click the **New Subitem** button. Click the 'Column' spinbox to choose a column for which the item text or pixmap will be placed. Click the 'Text' line edit to type text for a column, or to change the name of an item or subitem. Click a pixmap and then click the **Delete Pixmap** button to delete the selected pixmap. To delete an item from the listbox, click the item and then click the **Delete** button. To move an item up or down within the hierarchy level, click the **Move Up** or **Move Down** buttons. To move an item up or down one level, click the **Move Left** or **Move Right** buttons.



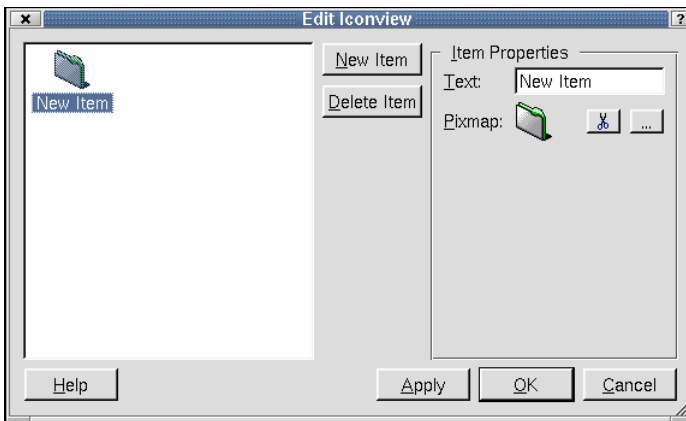
Edit Listview- Columns Tab

The Columns Tab

Click this tab to change the column configuration of the listview. To add a column, click the **New Column** button. The new column is shown at the top of the listbox. To change the column name, click a column in the listbox and then click the 'Text' line edit and type a new name. To add a pixmap, click the **(ellipsis)** button, which invokes the *Choose an Image Dialog*. To remove a pixmap, click the **Delete Pixmap** button. Click the 'Clickable' checkbox if you want the columns to respond to mouse clicks. Click the 'Resizable' checkbox if you want to be able to change the column's width. To remove a column, click the column in the listbox and then click the **Delete Column** button. To move a column up or down in the listbox, click the **Move Up** or the **Move Down** buttons.

Click **Apply** to accept changes to the listview widget. Click **OK** to leave the dialog once the changes have been

accepted. Click **Cancel** to leave the dialog without saving any changes.



Edit Iconview

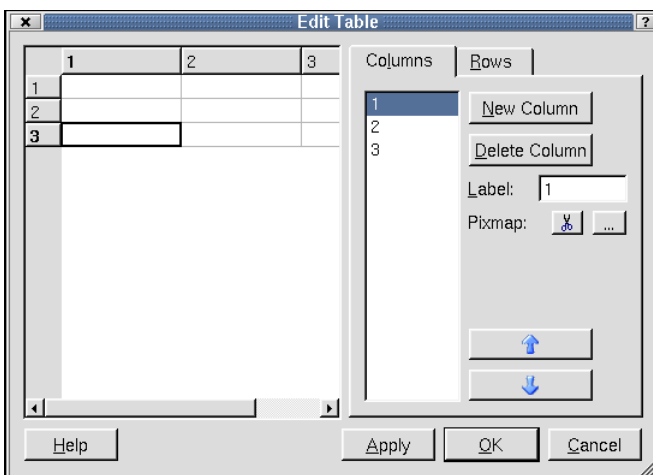
Edit Iconview

Right click or double click an iconview widget on the form and select 'Edit' to invoke the *Edit Iconview* dialog. Use the dialog to add, change, or remove items from the iconview. To add an item to the iconview, click the **New Item** button. To change the name of the item, click the 'Text' line edit and type a new name. To add a pixmap, click the **(ellipsis)** button, which invokes the *Choose an Image Dialog*. To remove a pixmap, click the **Delete Pixmap** button. To delete an item from the iconview, click the item and then click the **Delete Item** button.

Click **Apply** to accept changes to the iconview widget. Click **OK** to leave the dialog once the changes have been accepted. Click **Cancel** to leave the dialog without saving any changes.

Edit Table

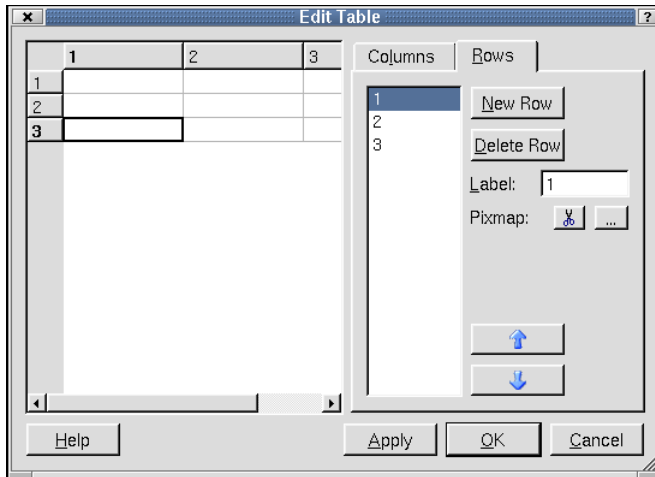
Right click or double click a table widget on the form and select 'Edit' to invoke the *Edit Table* dialog. Use the dialog to add, change, or remove columns or rows from the table.



Edit Table- Columns Tab

The Column Tab

To add a column to the table, click the **New Column** button. To delete a column from the table, click the column you want to delete from the table, or click the column number in the 'Columns' listbox and then click the **Delete Column** button. To change a column name, click the 'Label' line edit and type the new text. To add a pixmap, click the **(ellipsis)** button, which invokes the *Choose an Image Dialog*. To remove a pixmap from the current column of the selected item, click the **Delete Pixmap** button. To move a column in the listbox, click the **Move Up** or **Move Down** buttons.

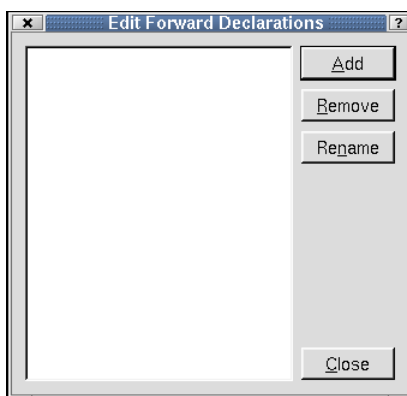


Edit Table- Rows Tab

The Rows Tab

To add a row to the table, click the **New Row** button. To delete a row from the table, click the row you want to delete from the table, or click the row number in the 'Rows' listbox and then click the **Delete Column** button. To change a row's name, click the row, or the row number, and then click the 'Label' line edit and type the new text. To add a pixmap, click the **(ellipsis)** button, which invokes the *Choose an Image Dialog*. To remove a pixmap from the current row of the selected item, click the **Delete Pixmap** button. To move a row in the listbox, click the **Move Up** or **Move Down** buttons.

Click **Apply** to accept changes to the table widget. Click **OK** to leave the dialog once the changes have been accepted. Click **Cancel** to leave the dialog without saving any changes.



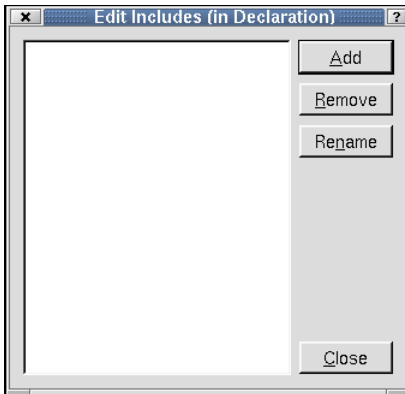
Edit Forward Declarations

Edit Forward Declarations Dialog

From the Source tab in the Object Explorer Window, right click the 'Forward Declarations' folder and select 'Edit' from the context menu to invoke the *Edit Forward Declarations* dialog. Use this dialog to add, edit, or remove declarations in the source code.

To add a new declaration, click the **Add** button. A line edit will appear for you to type the declaration. Press **Enter** after you have typed the declaration. To delete a declaration from the listbox, click the declaration and then click **Remove**. To rename an existing declaration, click the declaration and then click **Rename**. The cursor will appear in the line edit, allowing you to change the name.

Click **Close** to leave the *Edit Forward Declarations* dialog.



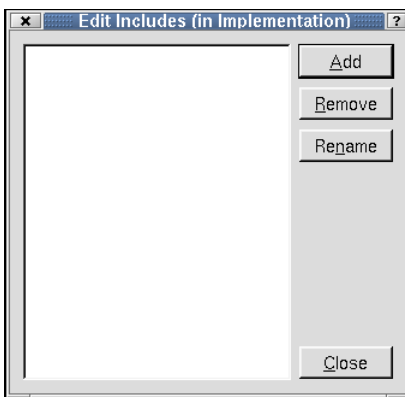
Edit Includes (in Declaration)

Edit Includes (in Declaration) Dialog

From the Source tab in the Object Explorer Window, right click the 'Includes (in Declaration)' folder and select 'Edit' from the context menu to invoke the *Edit Includes (in Declarations)* dialog. Use this dialog to add, edit, or remove includes in the source code.

To add a new include, click the **Add** button. A line edit will appear for you to type the include. Press **Enter** after you have typed the include. To delete an include from the listbox, click the include and then click **Remove**. To rename an existing include, click the include and then click **Rename**. The cursor will appear in the line edit, allowing you to change the name.

Click **Close** to leave the *Edit Include (in Declaration)* dialog.



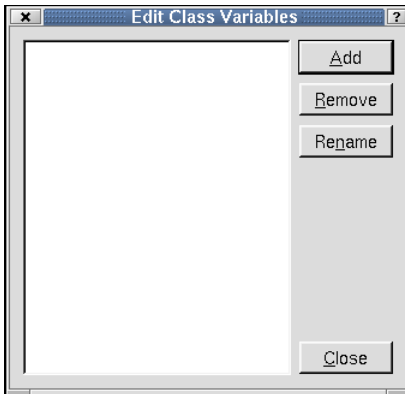
Edit Includes (in Implementation)

Edit Includes (in Implementation) Dialog

From the Source tab in the Object Explorer Window, right click the 'Includes (in Implementation)' folder and select 'Edit' from the context menu to invoke the *Edit Includes (in Implementation)* dialog. Use this dialog to add, edit, or remove includes in the source code.

To add a new include, click the **Add** button. A line edit will appear for you to type the include. Press **Enter** after you have typed the include. To delete an include from the listbox, click the include and then click **Remove**. To rename an existing include, click the include and then click **Rename**. The cursor will appear in the line edit, allowing you to change the name.

Click **Close** to leave the *Edit Include (in Implementation)* dialog.



Edit Class Variables

Edit Class Variables Dialog

From the Source tab in the Object Explorer Window, right click the 'Class Variables' folder and select 'Edit' from the context menu to invoke the *Edit Class Variables* dialog. Use this dialog to add, edit, or remove class variables in the source code.

To add a new variable, click the **Add** button. A line edit will appear for you to type the variable. Press enter after you have typed the variable. To delete an include from the listbox, click the variable and then click **Remove**. To rename an existing variable, click the variable and then click **Rename**. The cursor will appear in the line edit, allowing you to change the name.

Click **Close** to leave the *Edit Class Variables* dialog.

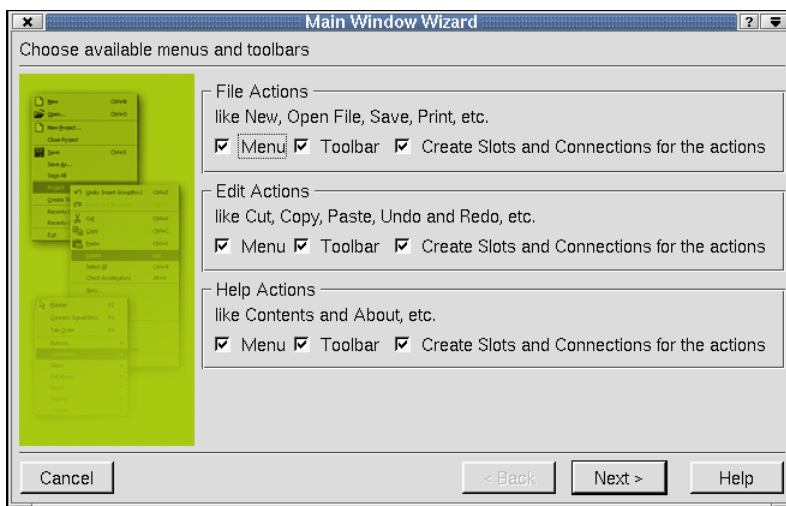
Reference: Wizards

Introduction

In *Qt Designer*, some of the toolbars, menu options and templates invoke wizards to take you step-by-step through particular tasks. In this chapter we explain each *Qt Designer* wizard.

Main Window Wizard

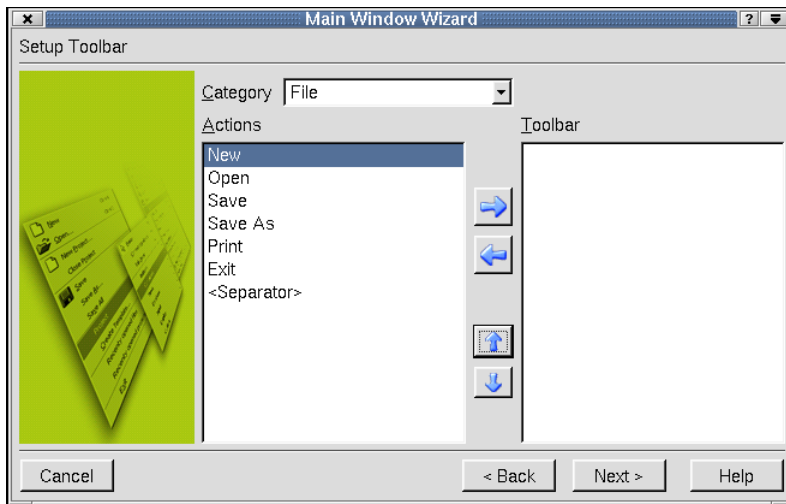
The Main Window Wizard is invoked by clicking the Main Window form template in the New File Dialog. This wizard helps you to create a main window with actions, menu options and toolbars.



Choose Available Menus and Toolbars

The 'Choose available menus and toolbars' wizard page appears first. It presents three categories of default actions, File Actions, Edit Actions and Help Actions. For each category you can choose to have *Qt Designer* create menu items, toolbar buttons and signal/slots connections for the relevant actions. You can always add or delete actions, menu items, toolbar buttons and connections later. Check or uncheck the checkboxes to reflect your preferences.

Click **Next** to move on to the next wizard page.



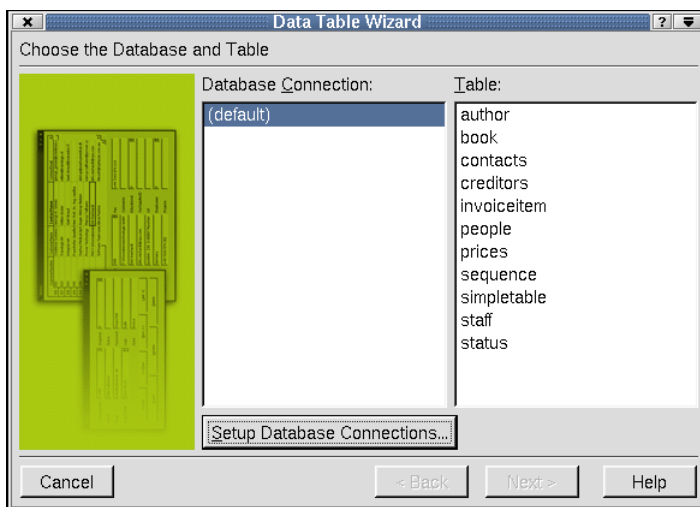
Setup Toolbar

The 'Setup Toolbar' wizard page is used to populate a toolbar with actions from each of the default action categories. Click the Category combobox to select which set of actions you wish to work on. The Actions listbox lists the actions available for the current category. The Toolbar listbox lists the toolbar buttons you want to create. Click the blue left and right arrow buttons to move actions into or out of the Toolbar list box. Click the blue up and down arrow buttons to move actions up and down within the Toolbar list box. Note that the '<Separator>' item in the Actions list box may be moved to the Toolbar list box as often as required and will cause a separator to appear in the finished toolbar.

Click **Back** if you want to return to the 'Choose available menus and toolbars' wizard page. Click **Finish** to populate the main window and to exit the wizard. Click **Cancel** on any of the wizard pages to leave the wizard without making any changes.

Data Table Wizard

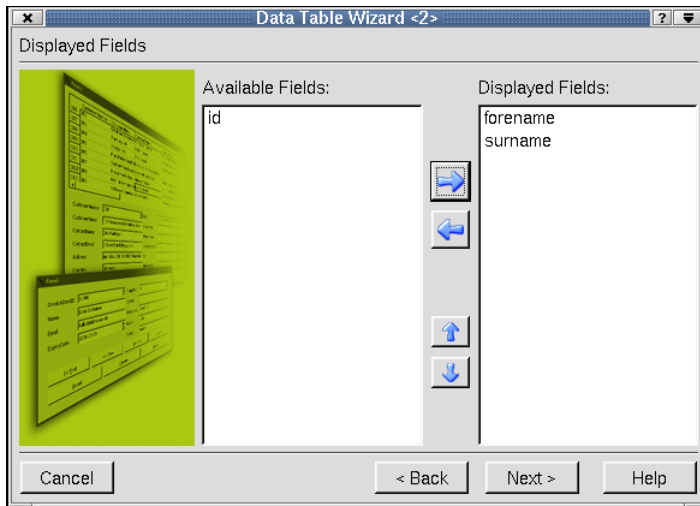
The Data Table Wizard is automatically invoked by clicking the datatable widget and placing it on the form. The datatable widget is used to create tabular views of database data.



Choose the Database and Table

The 'Choose the Database and Table' wizard page appears first. The available databases are displayed in the 'Database Connection' listbox. Choose a connection by clicking it. If there are no connections listed in the listbox, click **Setup Database Connections** to invoke the Edit Database Connections Dialog. The 'Table' listbox shows all the tables and views that are available through the selected database connection. Select a table or view by clicking it.

Click **Next** to move on to the next wizard page.



Displayed Fields

The 'Displayed Fields' wizard page is used to select fields that will be displayed in the table. By default, every field except the table or view's primary key, is initially placed in the 'Displayed Fields' list. Click the blue left and right arrow buttons to move fields from the 'Available Fields' listbox and into or out of the 'Displayed Fields' listbox. Click the blue up and down arrow buttons to move fields up and down within the 'Displayed Fields' listbox. The order in which fields appear in the 'Displayed Fields' listbox is the order they are shown in the Data Table, with the top-most field being in the left-most column.

Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'Choose the Database and Table' wizard page.

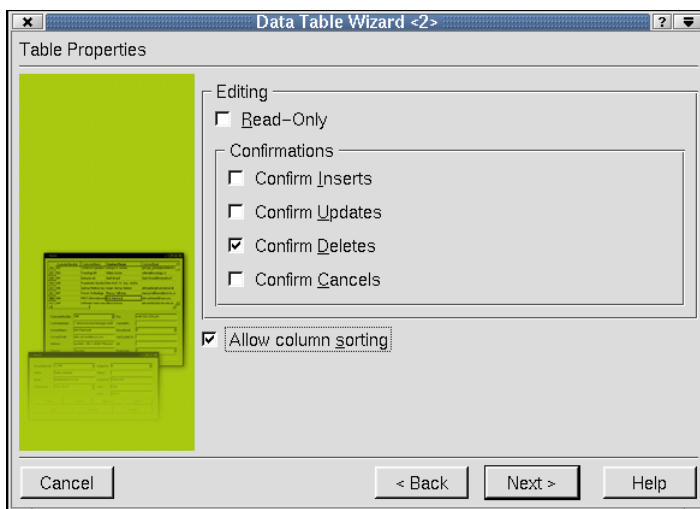
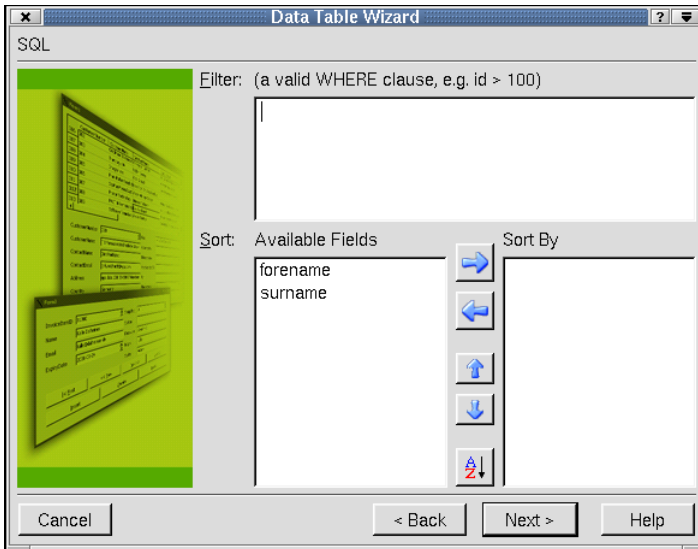


Table Properties

The 'Table Properties' wizard page is used to set the Data Table's initial editing options. Check the 'Read-Only' checkbox to prevent records from being edited, deleted or added. Check the checkboxes in the 'Confirmations' section to force the user to confirm their changes. By default users must confirm deletions. Click 'Allow column sorting' to allow the user to sort the data by clicking a column's header (which displays the field name).

Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'Displayed Fields' wizard page.

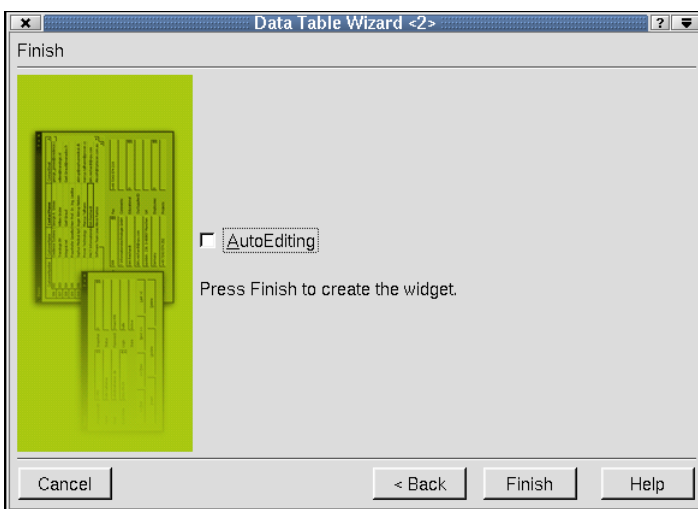


SQL

The 'SQL' wizard page is used to apply filters and sorts to the data in the table. Click the 'Filter' line edit and enter a valid SQL WHERE clause without the WHERE keyword. The filter applies to the data shown in the table.

To sort the available fields in the table, click the blue left and right arrow buttons to move fields from the 'Available Fields' listbox into or out of the 'Sort By' listbox. Click the blue up and down arrow buttons to move fields up and down within the 'Sort By' listbox. Click the **A-Z** button to change the sort order of the selected field in the 'Sort By' listbox from ascending to descending and vice versa.

Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'Table Properties' wizard page.



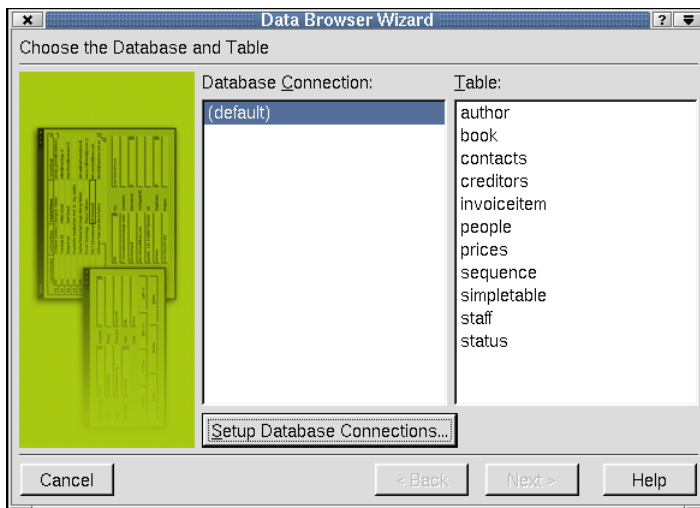
Finish

The 'Finish' wizard page is used to select auto-editing and to leave the wizard. If you want user edits, e.g. inserts and updates, to be automatically applied when the user navigates to another record, check the 'AutoEditing' checkbox. If 'AutoEditing' is unchecked, users must press **Enter** to confirm their edit before moving to another record, or their edit will be lost.

Click **Finish** to create the datatable widget with all of the options you selected in the wizard. Click **Back** if you want to return to the 'SQL wizard page. Click **Cancel** on any of the wizard pages to leave the wizard without making any changes.

Data Browser Wizard

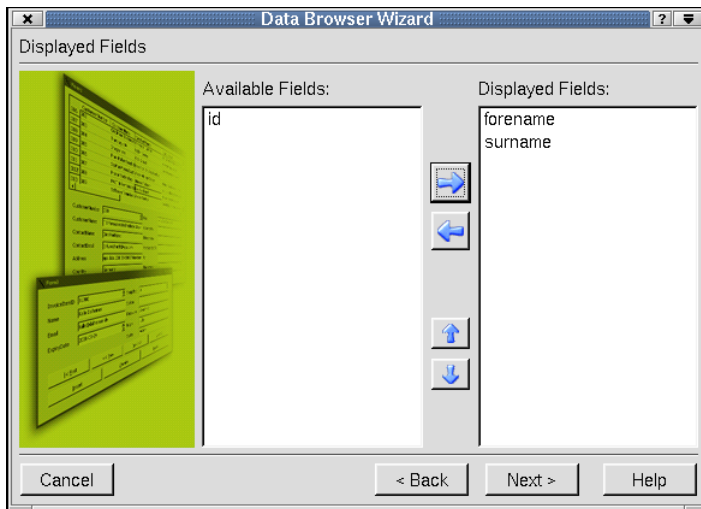
The Data Browser wizard is automatically invoked by clicking the DataBrowser widget and placing it on the form. The DataBrowser widget is used to create a form view of database data.



Choose the Database and Table

The 'Choose the Database and Table' wizard page appears first. The available databases are displayed in the 'Database Connection' listbox. Choose a connection by clicking it. If there are no connections listed in the listbox, click **Setup Database Connections** to invoke the Edit Database Connections Dialog. The 'Table' listbox shows all the tables and views that are available through the selected database connection. Select a table or view by clicking it.

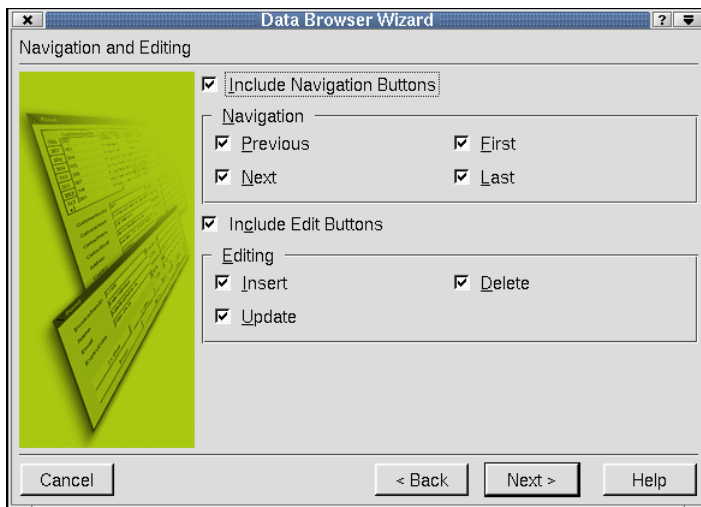
Click **Next** to move on to the next wizard page.



Displayed Fields

The 'Displayed Fields' wizard page is used to select fields that will be displayed in the table. Click the blue left and right arrow buttons to move fields from the 'Available Fields' listbox and into or out of the 'Displayed Fields' listbox. Click the blue up and down arrow buttons to move fields up and down within the 'Displayed Fields' listbox.

Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'Choose the Database and Table' wizard page.



Navigation and Editing

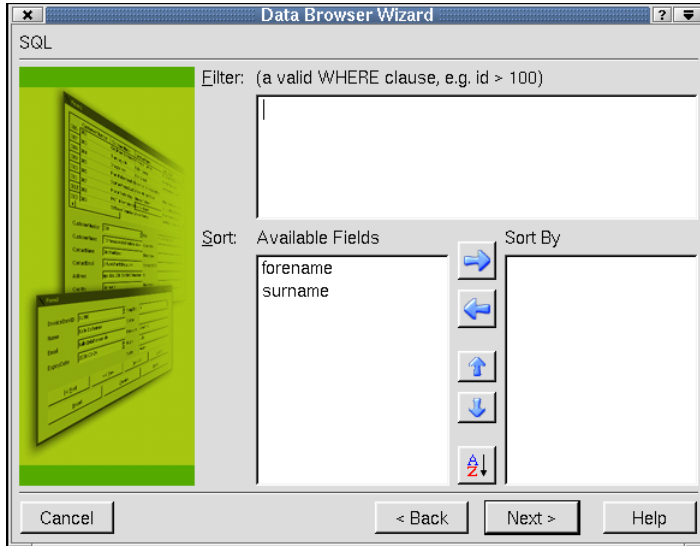
The 'Navigation and Editing' wizard page is used to create navigation and editing buttons.

Check the 'Include Navigation Buttons' checkbox to include navigation buttons. In the 'Navigation' section, click 'Previous' to display the 'Previous' button on the form. This option allows you to navigate to the previous record in the table. Click 'Next' to display the 'next' button on the form. This button allows you to navigate to the next record in the table. Click 'First' to display the 'First' button on the form. This option allows you to navigate to the first record in the table. Click 'Last' to display the 'Last' button on the form. This button allows you to navigate to the last record in the table.

Click the 'Include Edit Buttons' checkbox to include editing buttons. In the 'Editing' section, check the 'Insert' checkbox to create an 'Insert' button for adding new records. Check the 'Update' checkbox to create an 'Update' button for updating existing records. Check the 'Delete' checkbox to create a 'Delete' button for deleting records.

The navigation buttons, and 'Update' and 'Delete' buttons will work without requiring any code. Since most database designs expect new records to be created with a unique key the 'Insert' button will not work. This can easily be fixed by generating the key in a slot connected to the `QDataBrowser::beforeInsert()` signal.

Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'Displayed Fields' wizard page.

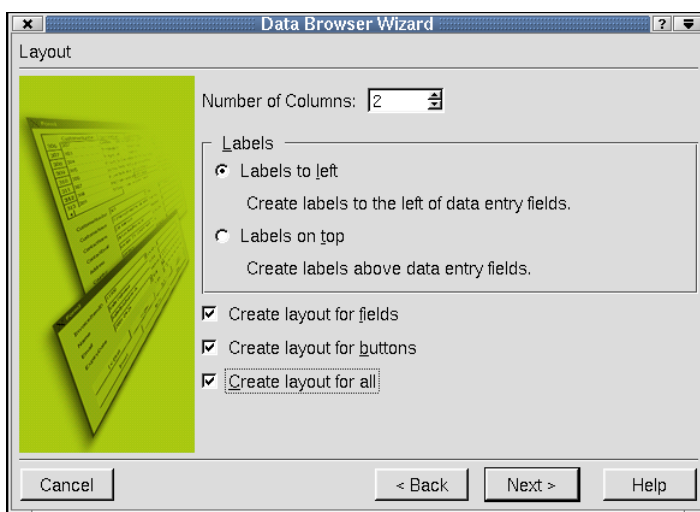


SQL

The 'SQL wizard page is used to apply filters and sorts to the data in the table. Click the 'Filter' line edit and type a valid SQL `WHERE` clause without the `WHERE` keyword. The filter applies to the data shown in the table.

To sort the available fields in the table, click the blue left and right arrow buttons to move fields from the 'Available Fields' listbox into or out of the 'Sort By' listbox. Click the blue up and down arrow buttons to move fields up and down within the 'Sort By' listbox. Click the **A-Z** button to change the sort order of the selected field in the 'Sort By' listbox from ascending to descending and vice versa.

Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'Navigation and Editing' wizard page.



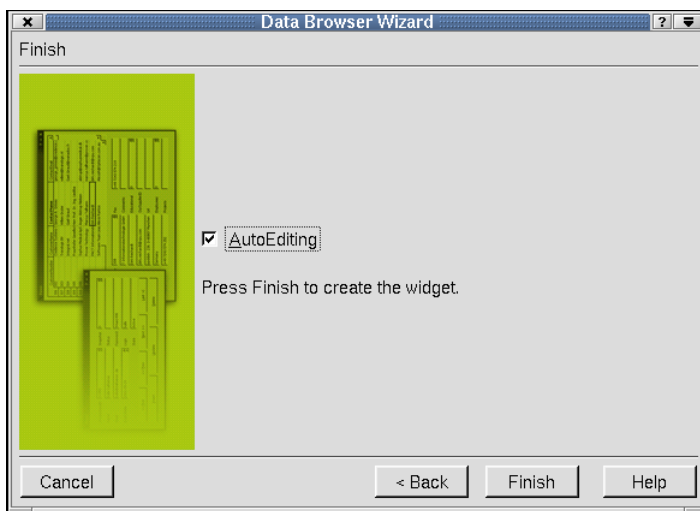
Layout

The 'Layout' wizard page is used to design the layout of the database browser. To choose the number of columns the form will use, click the 'Number of Columns' spinbox. To make labels appear to the left of the data entry fields, click the 'Labels to left' radio button. To make labels appear above the data entry fields, click the 'Labels on top' radio button.

Click the 'Create layout for fields' checkbox to arrange all fields inside of a box layout. Click the 'Create layout for buttons' checkbox to arrange all buttons inside of a box layout. Click 'Create layout for all' to create a box layout for the whole widget.

You can always break the layouts and redo them later if you change your mind.

Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'SQL' wizard page.



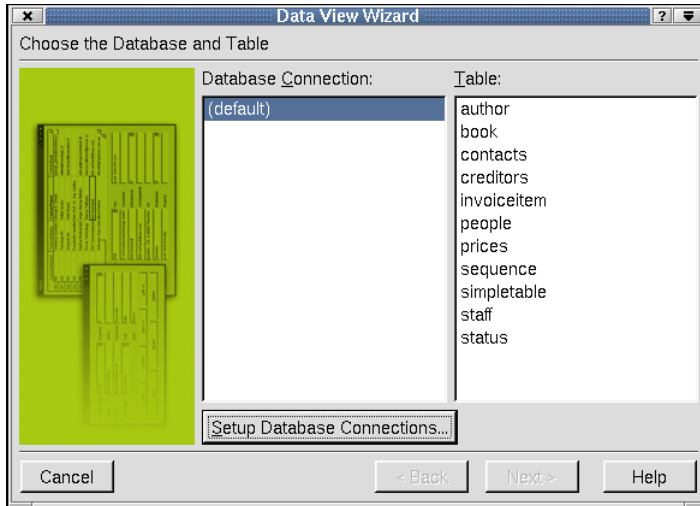
Finish

The 'Finish' wizard page is used to select auto-editing and to leave the wizard. If you want user edits, e.g. inserts and updates, to be automatically applied when the user navigates to another record, check the 'AutoEditing' checkbox. If 'AutoEditing' is unchecked, users must press **Enter** to confirm their edit before moving to another record, or their edit will be lost. This property can be changed later if desired.

Click **Finish** to create the databrowser widget with all of the options you selected in the wizard. Click **Back** if you want to return to the 'Layout' wizard page. Click **Cancel** on any of the wizard pages to leave the wizard without making any changes.

Data View Wizard

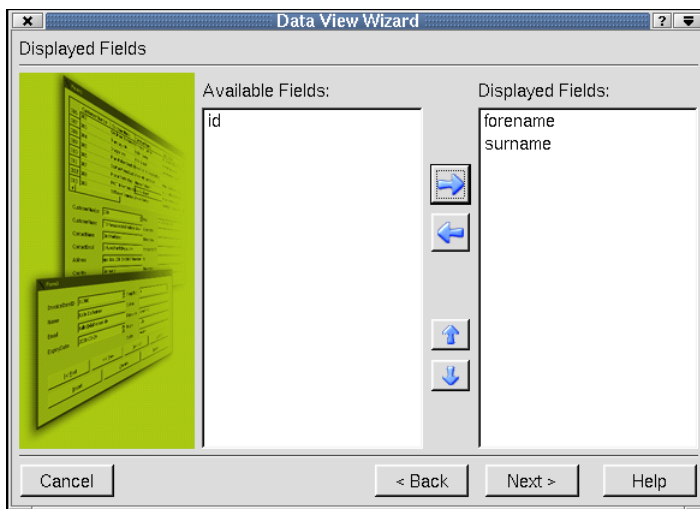
The Data View wizard is automatically invoked by clicking the dataview widget and placing it on the form. The Dataview widget is used to create a read-only form view of database data.



Choose the Database and Table

The 'Choose the Database and Table' wizard page appears first. The available databases are displayed in the 'Database Connection' listbox. Choose a connection by clicking it. If there are no connections listed in the listbox, click **Setup Database Connections** to invoke the Edit Database Connections Dialog. The 'Table' listbox shows all the tables and views that are available through the selected database connection. Select a table or view by clicking it.

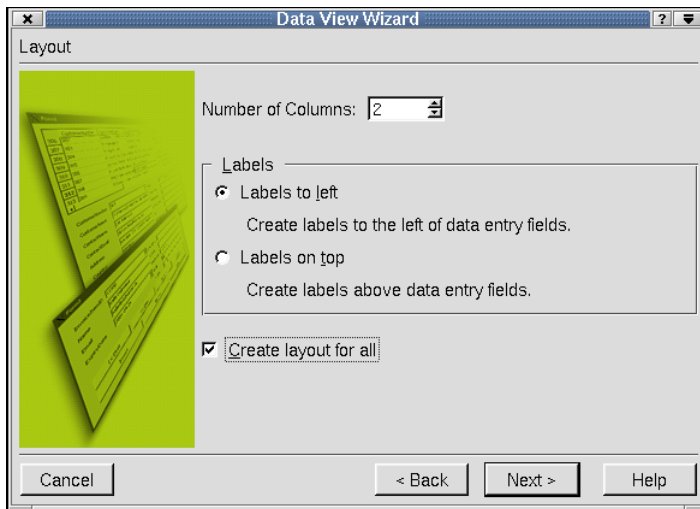
Click **Next** to move on to the next wizard page.



Displayed Fields

The 'Displayed Fields' wizard page is used to select fields that will be displayed in the table. Click the blue left and right arrow buttons to move fields from the 'Available Fields' listbox and into or out of the 'Displayed Fields' listbox. Click the blue up and down arrow buttons to move fields up and down within the 'Displayed Fields' listbox.

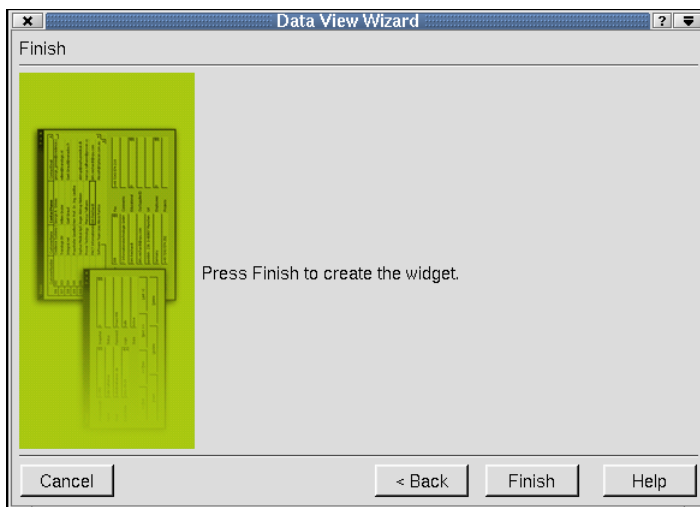
Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'Choose the Database and Table' wizard page.



Layout

The 'Layout' wizard page is used to design the layout of the data view. To choose the number of columns the form will use, click the 'Number of Columns' spinbox. To make labels appear to the left of the data entry fields, click the 'Labels to left' radio button. To make labels appear above the data entry fields, click the 'Labels on top' radio button.

Click **Next** to move on to the next wizard page. Click **Back** if you want to return to the 'Displayed Fields' wizard page.



Finish

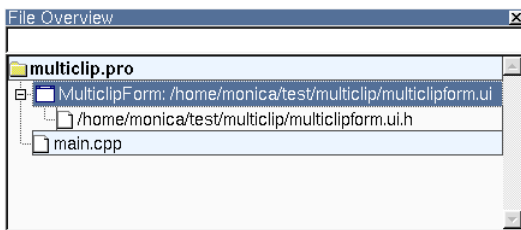
The 'Finish' wizard page is used to create the wizard once you have selected all the options you want on the previous wizard pages.

Click **Finish** to create the databrowser widget with all of the options you selected in the wizard. Click **Back** if you want to return to the 'Layout' wizard page. Click **Cancel** on any of the wizard pages to leave the wizard without making any changes.

Reference: Windows

Introduction

By default *Qt Designer* starts up with three windows on the left hand side. They are the File Overview Window, the Object Explorer Window, and the Property Editor/Signal Handlers Window. This chapter explains each window in detail.



File Overview Window

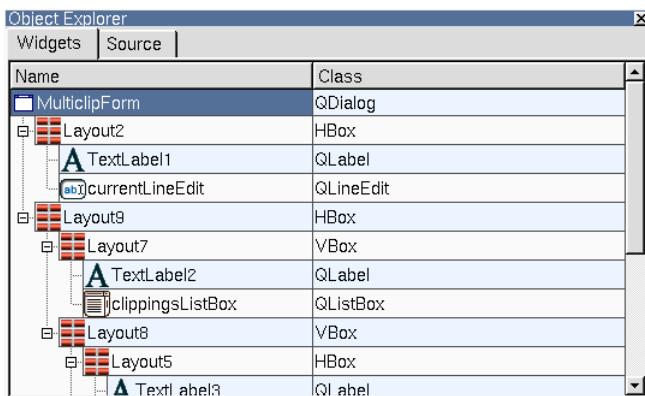
File Overview Window

This window lists all the files associated with the project. To open a form or file single click it in the Files list. To rapidly switch between forms and files, type the name of the file in the line edit above the files list and *Qt Designer* will perform an incremental search to show any matching files or forms.

Right-click a file (or the project) to get a context menu of options, for example, 'Open form' or 'Remove form from project'.

Object Explorer Window

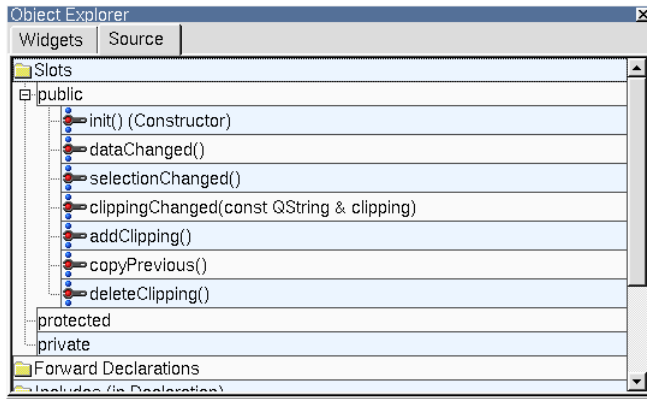
The Object Explorer window lists the current form's widgets and slots. The window contains two tabs, the Widgets tab and the Source tab.



Widget Tab

Widget Tab

Click the Widgets tab to view all the widgets for the current form. The widgets are listed by name and class. Click a widget in the list to highlight it in the corresponding form.



Source Tab

Source Tab

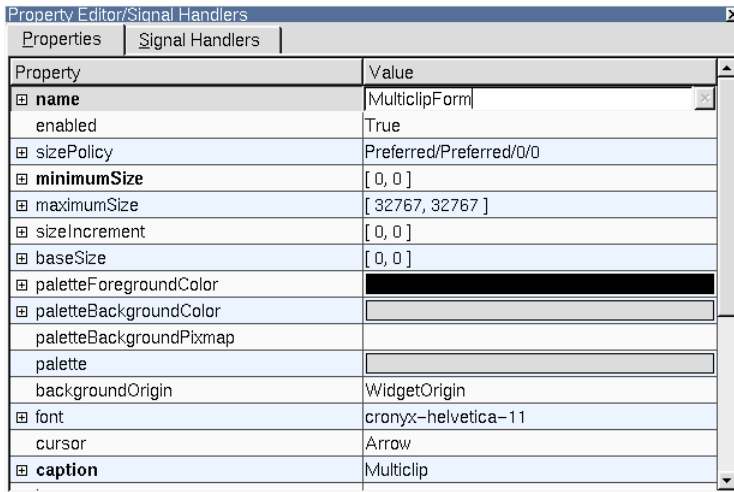
Click the Source tab to view the current form's slots, forward declarations, includes, and class variables. The Source tab uses a tree view to display its information. Items which have a '+' sign have sub-items which are revealed by clicking the '+'. Right click any item in the tree view to pop up a context menu.

To edit or add slots, right click the Slots folder and select 'Edit' to invoke the *Edit Slots Dialog*. Right click the Public, Protected, or Private subdirectories and click 'New' to invoke the *Edit Slots Dialog*. Right click a slot in the list to invoke a menu with additional options for the slot. To add new slots, choose 'New' from the menu, which invokes the *Edit Slots Dialog*. To change the properties of the selected slot, choose 'Properties' which invokes the *Edit Slots Dialog*. To open the C++ editor and jump to the implementation of the selected slot, choose 'Goto Implementation'. To remove the selected slot, choose 'Delete'. Signals can be added or deleted in the same way as slots.

Right click 'Forward Declarations', 'Includes (in declaration)', 'Class Variables', and 'Includes (in implementation)' to invoke a context menu with the 'new' or 'edit' options. Choose 'New' to invoke a line edit for typing a declaration, variable, or include. Right click 'Forward Declarations' and choose 'Edit' to invoke the *Edit Forward Declarations Dialog*. Right click 'Includes (in declaration)' and choose 'Edit' to invoke the *Edit Includes (in Declaration) Dialog*. Right click 'Class variables' and choose 'Edit' to invoke the *Edit Class Variables Dialog*. Right click 'Includes (in Implementation)' and choose 'Edit' to invoke the *Edit Includes (in Implementation) Dialog*.

Property Editor/Signal Handlers Window

Click the Property Editor/Signal Handlers window to view and change the properties of forms and widgets. This window has a 'Properties' tab and a 'Signal Handlers' tab.



Properties Tab

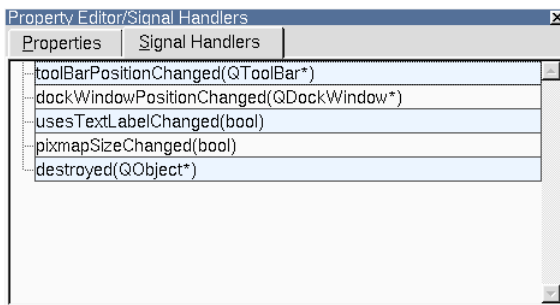
The Properties Tab

Click the 'Properties' tab to change the appearance and behaviour of the selected widget. The Property Editor has two columns, the Property column which lists property names and the Value column which lists the values. Click the column headers to sort the properties or values. Some property names have a plus sign '+' in a square to their left; this signifies that the property name is the collective name for a set of related properties.

Some properties have simple values, for example, the name property has a text value, the width property (within minimumSize for example) has a numeric value. To change a text value click the existing text and type in your new text. To change a numeric value click the value and either type in a new number, or use the spin buttons to increase or decrease the existing number until it reaches the value you want. Some properties have a fixed list of values, for example the mouseTracking property is boolean and can take the values True or False. The cursor property also has a fixed list of values. If you click the cursor property or the mouseTracking property the value will be shown in a drop down combobox; click the down arrow to see what values are available.

Some properties have complex sets of values; for example the font property. If you click the font property an ellipsis button (...) will appear; click this button and a Select Font dialog will pop up which you can use to change any of the font settings. Other properties have ellipsis buttons which lead to different dialogs depending on what settings the property can have. For example, if you have a lot of text to enter for a text property you could click the ellipsis button to invoke the multi-line text editor dialog. The names of properties which have changed are shown in bold. If you've changed a property but want to revert it to its default value click the property's value and then click the red 'X' button to the right of the value. Some properties have an initial value, e.g. 'TextEdit1', but no default value; if you revert a property that has an initial value but no default value (by clicking the red 'X') the value will become empty unless the property, e.g. name, is not allowed to be empty.

The property editor fully supports Undo and Redo (Ctrl+Z and Ctrl+Y, also available from the Edit menu).



Signal Handlers Tab

The Signal Handlers Tab

Click the 'Signal Handlers' tab to view or create the connections between signals of widgets and custom slots of the form.

Index

- .cpp, 11, 33, 34, 69
 - .dlg, 71
 - .glade, 71
 - .h, 33, 34, 69
 - .moc, 69
 - .pro, 33, 34, 43, 69
 - .ui, 3, 6, 13, 33, 34, 37, 39, 41, 67, 69
 - .xpm, 43

 - Absolute positioning, 9, 11
 - accept(), 13, 35, 38, 59, 63
 - Actions and Action Groups, 18, 19
 - Adding Action Groups, 20
 - Adding Actions, 19
 - Adding to a Toolbar, 21
 - Deleting, 21
 - Exclusive Actions, 20
 - activate(), 23
 - activated(), 23, 26
 - addDatabase(), 54
 - Adding
 - Actions and Action Groups, 18–20
 - Actions and Action Groups to a Toolbar, 21
 - Class variables, 11, 13, 26
 - Code, 11, 13
 - Code Editing, 23
 - Custom Widgets to Qt Designer, 44
 - Duplicate Widgets, 8
 - Files to Projects, 16, 33
 - Forms, 7
 - Forward declarations, 11, 13, 26
 - Includes, 11, 13, 26
 - Labels, 8
 - Main Widgets, 22
 - Menu Items, 22
 - Menu Separators, 22
 - Menus, 18, 19, 22
 - Pixmap, 20
 - Push Buttons, 8
 - Source Files to Project Files, 16, 26
 - Text Labels, 8
 - Toolbar Buttons, 18, 19
 - Toolbar Separators, 21
 - Toolbars, 18, 19, 21
 - Widgets, 8, 21
 - Automatically scaling widgets and application windows, 9

 - beforeInsert(), 63
 - beforeUpdate(), 63

 - book.pro, 56
 - book.sql, 55
 - Break layout, 9
 - Browsing Databases, 57

 - child(), 39, 41
 - Class variables, 11, 13, 26
 - clearValues(), 61
 - clicked(), 13, 15, 16, 35, 38, 43, 46, 59, 61–63
 - Clipboard, 13, 22, 23
 - Cross-platform, 14
 - Code Editing, 11, 13, 23
 - Not in Dynamic Dialogs, 37
 - Preferences, 66
 - Compiling and Building Applications, 16
 - Component
 - Plugins, 48
 - Connecting
 - Databases to Database Servers, 53
 - Signals and Slots, 12, 23
 - Controls
 - Widgets, 3
 - copy(), 23
 - create(), 39–41, 49
 - createConnections(), 54
 - Creating Custom Widgets, 44
 - Creating Forms
 - Dialogs, 7
 - Creating Main Windows, 18, 19
 - Creating Menus, 18, 19, 22
 - Creating Plugins, 48
 - Creating Projects
 - Projects, 6
 - Creating Templates, 67
 - Creating Test Harnesses for Forms, 35
 - Creating Toolbars, 18, 19, 21
 - Cross-platform
 - Clipboard, 14
 - Cross-platform previewing, 11
 - currentChanged(), 59
 - Custom Widgets, 42
 - Adding to Qt Designer, 44
 - Plugins, 44
 - Previewing, 42
 - Simple, 42
 - Cut and Paste (in Applications), 8, 22, 23

 - Data Aware Widgets, 52
 - Databases, 52

 - Browsing, 57
 - Confirmations, 57
 - Connecting to Database Servers, 53
 - Connecting to Multiple Database Servers, 54
 - Data Aware Widgets, 52
 - Data Browser Wizard, 60
 - Deleting Records, 57
 - Drilldown, 59–61
 - Drivers, 52
 - QMYSQL3, 52
 - QOCI8, 52
 - QODBC3, 52
 - QPSQL7, 52
 - Foreign Keys, 56, 63, 65
 - In-place Editing, 52
 - Inserting Records, 56
 - Master-Detail Relationships, 58
 - Relating Tables, 59
 - SQL Table Wizard, 55
 - Updating Records, 57
 - User Interface Interaction, 57
- dataChanged(), 14–16
- del(), 61
- Deleting
 - Actions and Action Groups, 21
 - Class variables, 11, 13
 - Forward declarations, 11, 13
 - Includes, 11, 13
 - ListBox Items, 8
 - Menu Items, 22
 - Menu Separators, 22
 - Menus, 22
 - Records
 - Databases, 57
 - Toolbar Separators, 21
 - Toolbars, 21
- destroy(), 11
- Dialogs
 - Creating New, 7
 - Dynamic, 33, 37, 39
 - Value Editors, 8
- Drilldown, 59–61
- Dynamic Dialogs, 33, 37, 39, 40
 - Compared with Compiling, 41
 - Loading and Executing, 39
 - Subclassing, 41
-
- Errors
 - Undefined reference, 33

- Exclusive Actions, Actions and Action Groups, 20
- exec(), 39
- exit(), 26
- Exiting Qt Designer, 6
- first(), 61
- Font Sizes, 23
- Foreign File Formats, 71
 - Glade, 71
 - Qt Architect, 71
- Foreign Keys, 56, 63, 65
- Forms
 - Adding, 7
 - Adding Widgets, 8
 - Class variables, 11, 13, 26
 - Code editing, 11, 13
 - constructor, 11
 - Creating Test Harnesses, 35, 43, 47
 - destructor, 11
 - Forward declarations, 11, 13, 26, 39
- Forward declarations, 11, 13, 26, 39
- Getting Help, 6
- Glade, 71
- group(), 49
- Grouping Widgets, 9, 10
- iconSet(), 49
- In-place Editing, 52
- includeFile(), 49
- Includes, 11, 13, 26, 39
- init(), 11, 13, 23, 26, 63, 64
- insert(), 61
- Inserting Records, 56
- isContainer(), 50
- Keypresses (Qt Designer's code editor), 66
- Keypresses (Qt Designer), 73
- keys(), 49
- last(), 61
- Layouts, 9
 - Break layout, 9, 12
 - Grid, 9, 12
 - Horizontal, 9, 10
 - Rubber band, 10, 12
 - Space filling, 9
 - Spacers, 9, 10, 35
 - Spacing, 11
 - Splitters, 10, 12
 - Undo and Redo
 - Undo and Redo, 9
 - Vertical, 9
- Macros
 - Q_ENUMS, 45
 - Q_EXPORT_PLUGIN, 50
 - Q_OBJECT, 34, 36, 40, 43, 45, 69
- Main Widget, 22
- Main Window
 - Adding a Main Widget, 22
 - Creating, 18, 19
 - Wizard, 19
- main.cpp, 16, 23, 33, 34, 37, 38, 43, 47, 54, 56
- Makefiles, 16, 27, 33, 37, 69
- Master-Detail Relationships, 58
- Menus
 - Adding, 18, 19, 22
 - Adding Menu Items, 22
 - Adding Menu Separators, 22
 - Deleting, 22
 - Moving, 22
- moc.exe, 70
- next(), 61
- Object Explorer, 6, 9, 11
- Object Hierarchy, 26, 62
- paste(), 23
- Pixmap, 6
 - Adding, 20
 - Adding to Forms, 6
 - In Projects, 6
- plugin.cpp, 48
- plugin.h, 48
- Plugins, 42, 66
 - Creating a Plugin, 48
 - Implementing Custom Widgets, 44
- Preferences
 - Code Editing, 66
- prev(), 61
- Preview Mode
 - Previewing, 11
- Previewing, 11
 - Custom Widgets, 42
 - Menus, 22
 - Signals and Slots, 23
 - Toolbars, 21
- primeInsert(), 55–57, 62
- primeUpdate(), 63
- Projects, 6, 18
 - Adding Files, 16, 33
 - Creating New, 6, 34
 - Database Connections, 53
- Properties, 7
 - Creating Custom Properties, 44
 - DataTable, 55
 - Initial values, 7
 - Property Editors, 8
 - Reverting changes, 7
 - Undo and Redo
 - Undo and Redo, 7
- Q_ENUMS, 45
- Q_EXPORT_PLUGIN, 50
- Q_OBJECT, 36, 43, 45, 69
 - Macros, 34, 40
- qApp->clipboard(), 13
- QApplication::clipboard(), 13
- QFileDialog::getOpenFileName(), 24
- qmake
 - HEADERS, 33, 37
 - SOURCES, 33, 37
- qmsdev.dll, 69
- qmsdev.dsp, 68
- QMYSQL3, Database driver, 52
- QObject::queryList(), 39
- QOCI8, Database driver, 52
- QODBC3, Database driver, 52
- QPSQL7, Database driver, 52
- QSqlDatabase::database(), 54
- Qt Architect, 71
- QWidgetFactory::create(), 39, 40
- readFields(), 61
- receiver.pro, 38
- Redo
 - Undo and Redo, 7
- refresh(), 61
- reject(), 35
- Rubber band, Selecting, 9, 10
- Scaling widgets and application windows, 9
- selected(QAction*), 23
- Selecting
 - Rubber band, 9, 10
- Selecting Widgets, 9–11
- selectionChanged(), 14
- Separator
 - Menu item, 19, 22
 - Toolbar button, 19, 21
- setBold(), 22
- setEnabled(), 35
- setFamily(), 23, 26
- setFocus(), 23
- setPointSize(int), 23
- setUnderline(), 22
- Signals and Slots, 12, 13, 15, 16, 22, 35
 - Connecting Actions, 19, 22
 - Connecting for Copy, 23
 - Connecting for Cut, 23
 - Connecting for Font Names, 23
 - Connecting for Font Sizes, 23
 - Connecting for Paste, 23
 - Connecting for Redo, 23
 - Connecting for Text Alignment, 23
 - Connecting for Undo, 23
 - Connecting to Close a Dialog, 38
 - Dynamic Dialogs, 39
 - Previewing, 23
 - Q_OBJECT, 34
- Slots
 - Signals and Slots, 13
- SQL, 52
- Starting Qt Designer, 6
- Subclassing, 11, 13, 33, 35
 - Dynamic Dialogs, 41
 - Widgets, 44
- Tab Order, 12
- Tab Order Mode
 - Tab Order, 12
- Templates
 - Base Class Templates, 68

- Creating and Using, 67
- Text Alignment, 23
- textChanged(), 46
- toggled(), 13, 22, 35
- Toolbar Buttons
 - Adding, 18, 19
- Toolbars
 - Adding Widgets, 21
 - Creating, 18, 19, 21
- toolTip(), 49
- Tooltips, 8

- uic.exe, 70
- Undefined references, Error, 33
- Undo and Redo
 - Layouts, 9, 12
 - Properties, 7

- update(), 61
- Updating Records, 57
- User Interface Interaction, Databases, 57
- Using the Property Editor, 7

- Value Editors, 8
 - Dialogs, 8
 - List Box, 8
 - SQL Table Editor, 58
- valueChanged(int), 23
- Visual Studio, 68

- whatsThis(), 49
- Widgets
 - Adding a Main Widget, 22
 - Adding to Toolbars, 21
 - ComboBox, 21

- Creating a Custom Widget, 44
- Custom, 42
- Data Aware, 52
- Grouping, 9, 10
- Line Edit, 9
- Push Button, 9
- Repeatedly Adding, 8
- SpinBox, 21
- Widgets and Source window
 - Object Explorer, 6
- Windows, Microsoft, 14
- Wizards
 - Data Browser, 60
 - Main Window, 19
 - SQL Table, 55
- writeFields(), 61