Using QtArchitect

Klaus Ebner [1] Jeff Harris [2]

September 9, 1998

[1]klaus@gaspode.ndh.com
[2]jharris@primenet.com

Abstract

QtArchitect is written to help users building dialogs with the Qt library.

QtArchitect is still under construction and will remain there until it's perfect. As this won't happen in the next future, expect that some parts of this document were outdated before you read them. If you want to look inside QtArchitect and write a new submodule (If you don't know what to do with your spare time, ask me...), take a look in Appendix 7 for some hints and drop us a mail.

As I mentioned, QtArchitect isn't perfect, yet. So it might happen that you find a small bug. If you dare and track it down, fine, send us the patches, if not, never mind: probably we found it already and didn't managed to track it down, too. But nevertheless, send bug reports to one of us. At least we will answer as fast as possible.

Contents

1 Install QtArchitect

1.1 Requirements

Before installing QtArchitect you should have Qt 1.4 already installed. Please follow the instruction which come with the Qt distribution.

QtArchitect need flex to parse its files, so you need at least flex 2.5.3 installed.

While compiling you'll need about 70MB of free space. After that, you could strip QtArchitect down to approximately. 1.2MB.

1.2 Unpacking and compiling

[1] Unpack your distribution file in your favorite directory. We assume that this will be `/usr/src/qtarch`. Change to this directory, take a short look at Makefile and probably change INSTDIR to your needs. Then start make and relax. QtArchitect needs its time to compile as it consists of more than 440 source files.

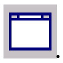
After compilation is done you might want to strip down QtArchitect from 17MB by using `strip -g qtarch`. A `make install` will copy QtArchitect to the place specified by INSTDIR.

[1] You might skip this section if you got the binary distribution

Figure 1: QtArchitect after start

2 A first example

Start up QtArchitect . You'll get the following dialog (Figure 1). The menu bar has only one entry over which you can open or create new projects, pixmaps or dialogs. You may do this also by using the tool bar (from left to right: Create project, dialog and pixmap, open project, dialog and pixmap). The next two buttons are "Save all objects" and "Generate all objects". In the main window the loaded or created objects would be listed.

We start by creating a new dialog. Choose *New Dialog* from the *File* menu or click on the *New Dialog* tool button: . A new window appears with the dialog editor (Figure 2). At first, you have to change file generation parameters (because if you don't do that first, you'll forget it). Use *Properties* from the *Edit* menu or the context menu which appears after clicking the working area with the right mouse button.

In the dialog properties you select the second page *File Generation* and enter `FirstExample` as Window Base Class. While typing the default values for the other entries are filled in (Figure 3). If you don't like this defaults, simply fill in your own, QtArchitect won't touch this field anymore.

Now we add a pushbutton with the tool bar button . The new pushbutton appears in the upper left corner of the dialog and you can move and resize it by using the mouse. When you enter the pushbuttons properties

Figure 2: The dialog editor

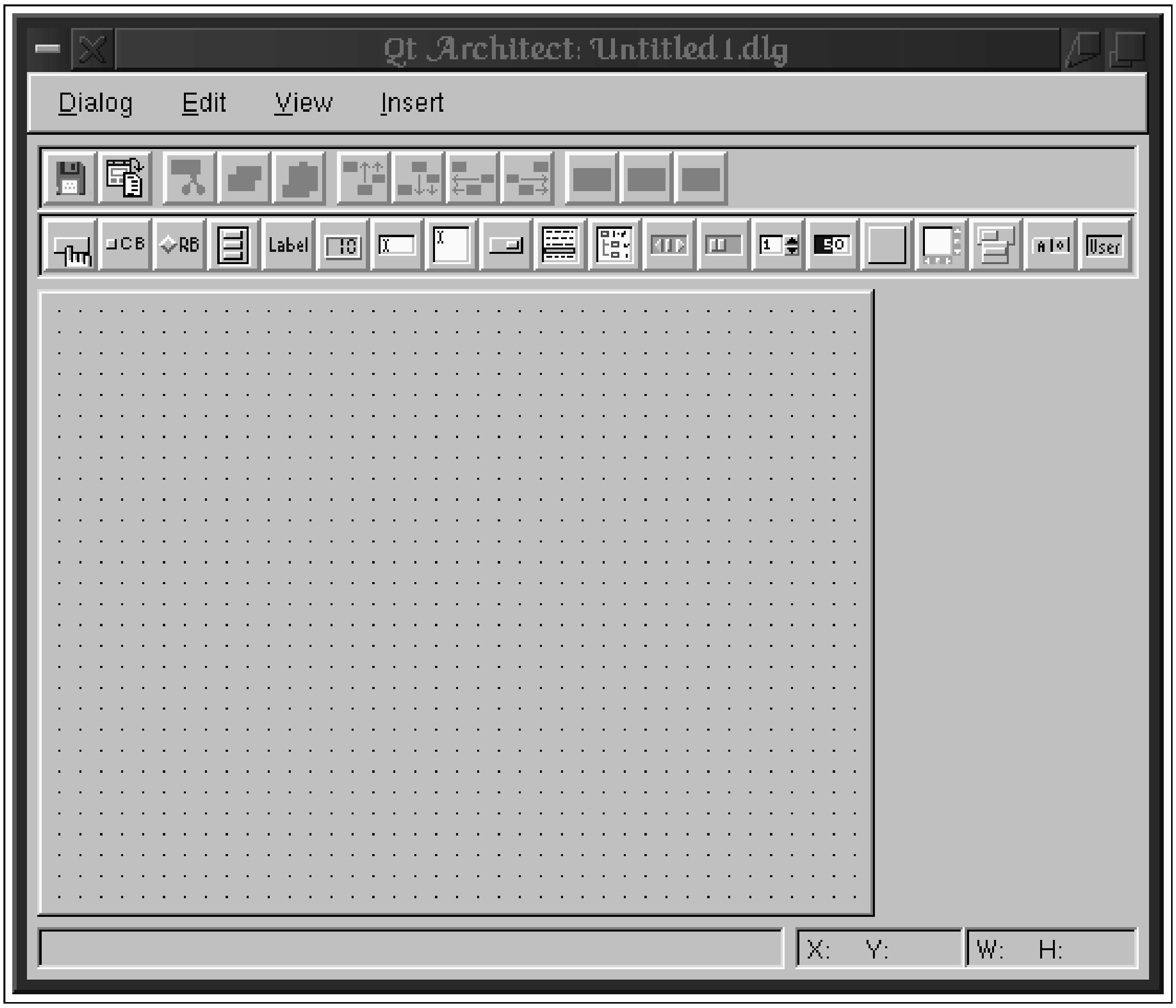


Figure 3: Dialog properties/File generation

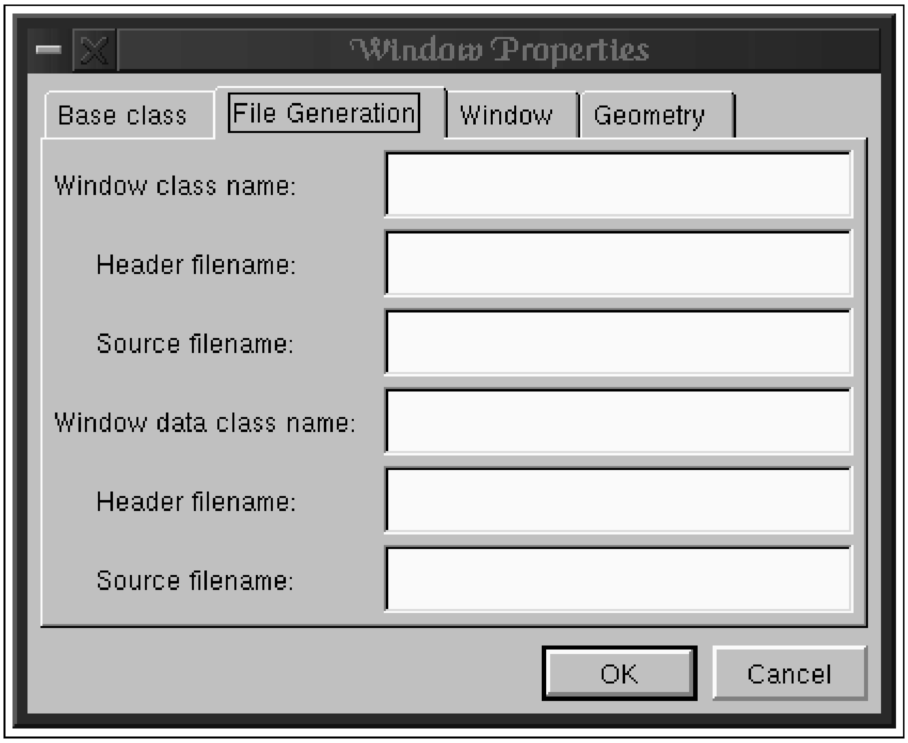
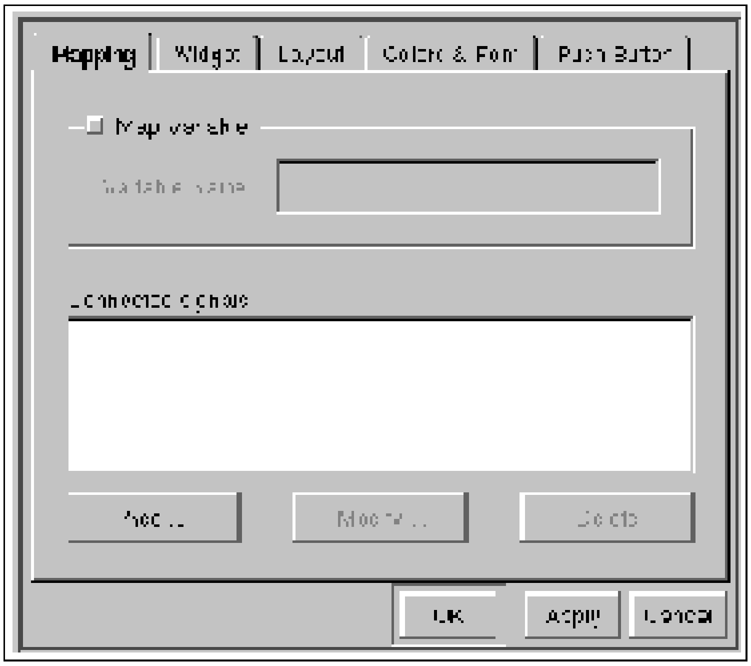


Figure 4: Push button properties

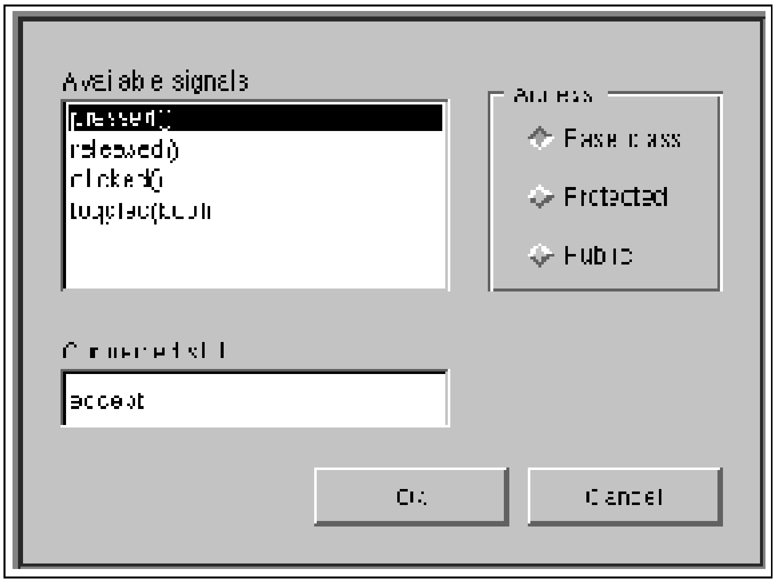


(right mouse button or menu bar), you get a dialog like (Figure 4). Add a signal/slot connection by using *Add...* and fill in the new dialog like (Figure 5). In the properties dialog you now see a line like this `[BaseClass] pressed --> accept()`. Change to the *Layout* page and choose *Add to Layout*. The button will be added to the layout in which it lies, which is the dialog frame in our case. To finish this button choose *Push Button* page and enter a button text. After leaving the properties, the editor get updated and now appears like (Figure 6).

Next we assign a layout to the dialog by opening the *Layout Manager* window from the *View—Layout Manager* menu item. We can add a Box Layout by selecting the *Insert—Box Layout* menu item. Also, you can add the layout by pressing the *Box Layout* tool bar button: . Next, we add a Layout Widget by selecting the menu item *Insert—Widget*. Click on the tree item and select the pushbutton.

Now we are ready to generate c++ sources by using *Dialog/Generate Files* or tool bar . This dialog shows the filename which will get gen-

Figure 5: A sample Signal/Slot connection



erated and allows editing them (Figure 7). Start generation by clicking on *Generate*.

Now all four files are generated: `FirstExample.cpp` and `FirstExample.h` are the interface class for your application and `FirstExampleData.cpp` and `FirstExampleData.h` are the superclass for the first one, containing the user interface:

As you can see, the interface class FirstExample contains no code from the user interface. This way QtArchitect allows you to change the interface code (implementing slots, ctor, dtor, ...) and change the GUI later. QtArchitect first checks the existence of the interface class sources, and if they exists, code generation won't generate them again by default.

Although QtArchitect can generate `Makefiles`, too, I recommend the usage of Trolls `tmake` package. When dealing with Qt programs, tmake

Figure 6: Our first dialog

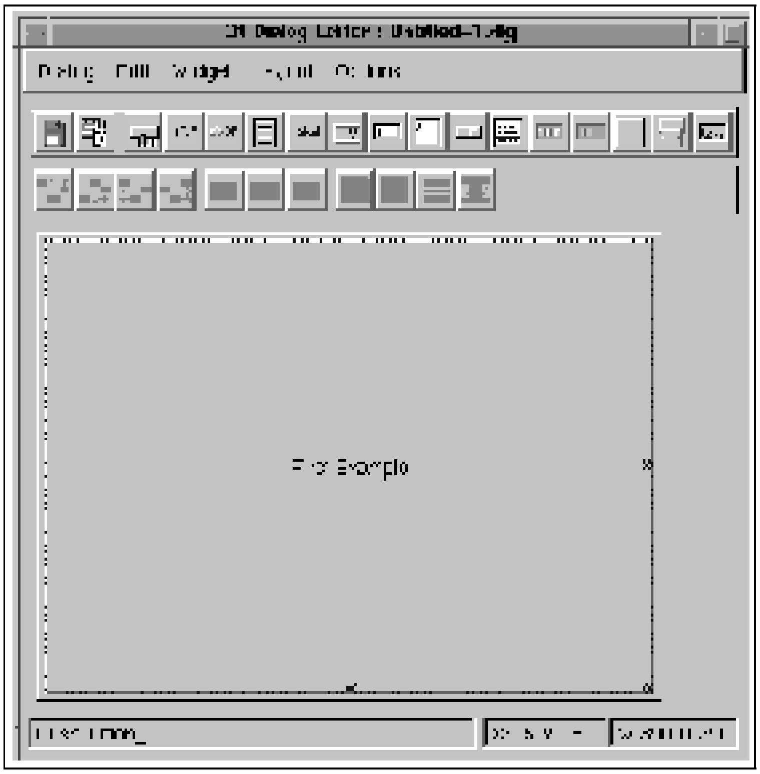
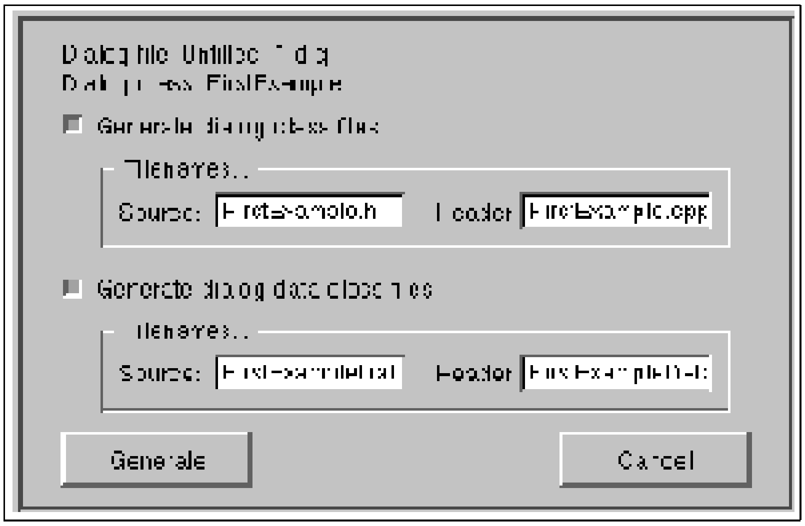


Figure 7: Generate Sources



does a good job. QtArchitect will generate tmake input files in its next version, up to then, you have to use `progen` from the same package to generate the needed.

There's one step left to accomplish: saving the QtArchitect file for our dialog. Raise the *QT Dialog Editor* window and click with the right mouse button on the only list entry available (should be `Untitled-1.dlg`). Choose *Save as...* and enter `FirstDialog.dlg`.

Now you can load your first dialog later and add some more controls, change the layout or text fields and generate the data classes again.

3 Creating and editing dialogs

There's a html online help for the dialog editor under ./help. Please check it out in any questions concerning this editor.

3.1 General

If you go through the first chapter and create *FirstExample* you have seen a lot of the mechanisms of QtArchitect . QtArchitect uses very common ways to handle things: menu bars, tool bars and context menus. We tried to make all the icons as descriptive as possible without spending too much time on them. But there's a bubble help for all icons which helps a lot.

3.2 Using the Mouse

You can change position and geometry of all widgets with the mouse [2]. If a widget is selected, there are eight pick marks drawn on its inside to resize the widget and if you drag the widget with left mouse button pressed you can reposition it.

To help you building properly layouted dialogs, QtArchitect provides a grid to which the widgets can be constrained. *Options/Grid* shows a small dialog in which you can set the grid's width and enable or disable it. Changing the grid didn't change widgets which are already positioned.

There are known problems with some Qt widgets: `QLineEdit` and `QMultiLineEdit`. They grab input focus and so resizing them can be problematic. If nothing helps, use their property dialog and set position and size by yourself. Or, use the keyboard accelerators described in the section on using the keyboard. `QMultiLineEdit` makes another problem when you have entered an text which forces scrollbars. QtArchitect cannot get the messages for these scrollbars, as they are handled completely inside Qt. If you want to resize this widget, you can only drag the upper left corner.

You can select multiple widgets using *Ctrl* key and left mouse button.

3.3 Using the Keyboard

There are many keyboard accelerators available for use in QtArchitect . The accelerators available for the dialog editor window are:

[2] Using layouts may restrict the ways in which you can position and resize widgets

Widget Insertion The accelerators for inserting widgets follow the form *Ctrl* or *Alt* and a number. For example, *Ctrl-1* will insert a new QPushButton. The complete list is shown in the *Insert* menu item.

Cut, Copy and Paste The accelerators for Cut, Copy and Paste are *Ctrl-X*, *Ctrl-C*, and *Ctrl-V* respectively.

Widget Alignment The accelerators for aligning multiple widgets when they are selected are:

Alt-T Align all selected widgets with the top of the first selected widget.

Alt-B Align all selected widgets with the bottom of the first selected widget.

Alt-L Align all selected widgets with the left of the first selected widget.

Alt-R Align all selected widgets with the right of the first selected widget.

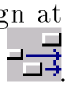
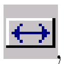
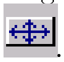
Alt-H Make all selected widgets the same width as the first selected widget.

Alt-V Make all selected widgets the same height as the first selected widget.

Alt-S Make all selected widgets the same size as the first selected widget.

Widget Movement The cursor keys will move the selected widgets. The cursor keys used with the *Shift* button pressed will resize the widgets. *Shift-Up* and *Shift-Down* make the widgets shorter and longer. *Shift-Left* and *Shift-Right* make the widgets narrower and wider.

3.4 Geometry

When multiple widgets where selected, the alignment buttons in the tool bar become sensitive: Align at the top of the first selected widget , align at the bottom of the first selected widget , align at the left side of the first selected widget  and align on the right side . There's the possibility to resize the widgets to a common size using the size buttons: Resize to the width of the first selected widget , resize to the height  and resize to both width and height of the first selected widget .

3.5 Properties

All the widgets have some things in common: The mapping of variable names to the widget in source code, the availability of signals, a name, a caption, geometry, the belonging to a layout and individual color and font. Every widget has also some special things to offer: special setting or initial values.

Mapping Normally QtArchitect only creates anonymous objects in the data class ctor. This is sufficient for most cases. But when you have to access the widget from your code, you have to map a variable to the widget. To do so, simply check *Map Variable* and enter a name. Beware: QtArchitect didn't check this name, so you have to assure that this is a valid c++ name and that it didn't already exists.

Almost all Qt widgets send out signals when the user interacts with them. You can connect user defined or base class slots to this signals from QtArchitect . There's no need to map a variable to do so. You can see the list of connections in this dialog and add, modify or delete them. To add a connection, click *Add...* and choose from the list of available signals (differ from widget to widget). After you select a signal, you can choose the slot and its attributes. A slot may be located in the base class of your dialog (eg. `accept()` in `QDialog`), or it will be generated in your data class as a virtual method. Then you can decide to make this slot protected or public [3]. At least you have to enter a slot name, to make the *OK* button sensitive.

Widget All Qt widgets may have a name, and QtArchitect uses this name internally. QtArchitect generates a default name which is unique. If you want to change the name (there shouldn't be a need for), please insure uniqueness as QtArchitect won't accept double names.

The geometry of the widget may be adjusted here, too.

[3] You cannot define private slots here, since the interface class will inherit from the data class and so you need access to this slots

Layout If you enable layouts in your dialog, you can add a widget to the layout on which it is currently positioned. In the simplest case, all you have to do, is to check *Add to Layout*. Then you could adjust stretch and alignment in the layout. See chapter 3.7 for more details on this topic.

Colors All widgets may use different colors (please don't do this...). Here you can choose the color group to edit (normal, disabled or active) and set all possible colors by clicking on the colored buttons. A color select dialog appears and you can edit the color the way you want it to be.

For the font, all data is shown and you can change it with the *Choose* button. The font selection dialog allows modifying the font in all ways known to us and Qt.

Bitmaps Some widgets (namely the ones derived from QButton, QLabel, QFrame and QGroupbox) have a page for specifying a background pixmap. If you want a background pixmap, check the check box to use one, give the name of the bmp file and choose from the following options:

- Adjust widget size changes the actual widgets size to the size of the pixmap and fix the size.
- Add Bitmap to current project is currently only half way implemented. The pixmap isn't added, but code generation will assume it. The effect shows up when Load from data is checked:
- If Load from data is checked, you have to enter the variable name where the data is stored and the name of the variable which holds the length of the variable data. The pixmap is then loaded with the method loadPixmapFromData instead of reading from a file. If "Add Bitmap to current project" isn't checked, the data for the pixmap will be generated in the *Data class for this dialog, otherwise no code will be generated.

Widget Specific All widgets have some special settings. You can set everything mentioned in the Qt documentation for this widget. As some widgets are subclassed from QFrame, the frame page is shown in their properties dialog, too.

3.6 Cut & Paste

You can cut or copy selected widgets into the system clipboard. From there you can paste them back into the same or different dialogs. You can paste them into other X11 applications, too. This way, you can copy a special setting into a EMail and send it to friend, he simply copies the buffer into the clipboard again and pasts it into his dialog. This is an example of an copied PushButton:

```
DlgEdit:v1.2:Dialog:
WidgetLayout {

PushButton {
ToggleButton {FALSE}
Default {FALSE}
AutoDefault {TRUE}
Text {OK}
AutoRepeat {FALSE}
AutoResize {FALSE}
Rect {0 0 100 30}
Name {PushButton_2}
Variable {itsOKButton}
Signal {[BaseClass]  pressed --> accept  ()}
LayoutStatus {NoLayout}
MinimumSize {10 10}
MaximumSize {32767 32767}
}}
```

3.7 Layout

Layouts are very useful when designing dialogs which could change their size at runtime (maybe to support different resolutions). Qt provides to different layout algorithms and QtArchitect support both of them.

To open the layout editor, select the *View—Layout Manager* menu item. Once the layout editor is displayed, you can begin to add layouts to your window.

The layouts are displayed hieraricaIly in a tree. At the root of the tree, Box Layouts and Grid Layouts can be inserted. These layouts are defined for the entire dialog. Layouts can be nested inside other layouts. Also, widgets can be added to layouts. These widgets can also have layouts defined inside them. For example, a QButtonGroup can contain a layout containing its QRadioButtons.

A description what can be inserted into each type of layout item is given below:

Box Layout Box Layouts arrange their items in a list form. The elements can be arranged from left to right, right to left, top to bottom or bottom to top. The border between items can be set. Also, a menu bar from the dialog can be set. If the Box Layout is contained in another Box Layout, its stretch can be set.

Box Layouts can contain other Box Layouts and Grid Layouts. They can also contain empty spacing and stretch elements. Layout Widgets can also be inserted into Box Layouts.

Box Spacing Box Spacing is empty space inserted between its surrounding elements in a Box Layout. Its size is set to a fixed number of pixels.

Box Spacing can not contain any other items.

Box Stretch Box Stretch is empty space inserted between its surrounding elements in a Box Layout. Its size shrinks and grows in relation to the stretch values of the other items in the Box Layout.

Box Stretch can not contain any other items.

Grid Layout Grid Layouts arrange their items in a grid. The border between columns and rows can be set. A menu bar can be set for the Grid Layout. The Grid Layout can set the stretch factor of each of its columns and the spacing between the columns. If the Grid Layout is inserted inside a Box Layout, its stretch factor can be set.

Grid Layouts can contain Grid Rows representing each row of the Grid Layout.

Grid Row Grid Rows allow the stretch factor and spacing of the row to be set. A Grid Row contains the items which make up the columns of the row.

Grid Rows can contain Box Layouts, Grid Layouts, Grid Spacer, and Layout Widgets.

Grid Spacer Grid Spacer represents an empty cell in its Grid Layout.

Grid Spacers can not contain any other items.

Layout Widget Layout Widgets represent a widget defined on the dialog. The widget from the dialog can be chosen. Also, the horizontal and vertical alignment of the widget can be set. The alignment is useful if the widget has a fixed size and needs to be positioned in a Grid or Box Layout. If the Layout Widget is in a Box Layout, its stretch factor can be set.

Layout Widgets can contain other Box and Grid Layouts. The parent of the layout is the widget, not the dialog or another layout. So, other widgets inserted in the Layout Widget will have the Layout Widget as their parent.

A widget set in a layout will have its selection box's color changed to blue. The widget will no longer be movable. Any changes in its size will set the widget's fixed size.

3.8 File generation

One of the most important things QtArchitect can do, is generating C++ code for your dialog. As seen in the example, QtArchitect produces four files for each dialog: The interface class specification and body and the data class specification and body. You should never edit the later, as it will be simply overridden when you re-generate your files after a change in the layout. The filenames which are produced are a property of the dialog, so you can edit them in the dialog properties, page *File Generation*. The edit fields in this dialog are connect to each other in order to help you typing: Simply filling the *Window class name* will fill the others with their defaults: *Header/Source filename* will become *Window class name* with appended *.h/.cpp* and *Window data class name* becomes *Window class name* with appended *Data*. After you edit any other field, this group will get disconnected from the mechanism.

To start generation, use the tool bar  or the menu *Dialog/Generate Files*. Now you have a last chance to change the filenames and decide what files you want to create. When generating for the first time, both check boxes are checked and all four files will get generated. Next time, the *Generate dialog class files* check box isn't checked to prevent overriding already edited files accidentally. The files are generated into the current directory [4].

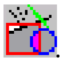
[4] Will get changed next version

4 Pixmaps

Besides dialogs, QtArchitect is a quite handsome pixmap editor, too. This part of QtArchitect is designed to help you build up graphical user interfaces with colorful and fancy buttons all over. It can handle pixmap seizes from 1x1 (very fast) up to the Qt limits (not so fast).

For now, the pixmap editor is a rather separated part of QtArchitect . It will become more integrated next versions.

4.1 Starting the pixmap editor

To start editing a new pixmap, simply use menu *File/New Pixmap* or use tool bar . A new pixmap is created with the following default values:

Background color	—	white
Left mouse color	—	black
Right mouse color	—	white
Width	—	32
Height	—	32
Zoom	—	8
Drawing mode	—	pixel

You can also open existing QtArchitect pixmaps using menu *File/Open Pixmap*. The pixmap will get loaded with the above defaults apart from the size.

4.2 Editing pixmaps

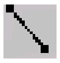
After you create a new, or load an existing pixmap, editing it is simple. Just try your luck and put a black pixel at a white place. You've notice the change ? In the lower right of the editor, there's a preview of your pixmap (Zoom 1) where the new black point should be visible, too. Now click again on this new black pixel, but use now your right mouse button. The pixel is white again.

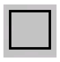
4.2.1 Colors

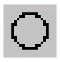
To change the drawing colors for the left or right mouse button, click on the desired color in the color group. Above the color choose buttons, there are two indicator buttons which shows your current selection [5]. Below the color group is an *Edit...* button which shows up a color tuning dialog (the same dialog which appears when editing colors while working on widget properties). You can edit the color of the right button by clicking *Edit...* with the right mouse button.

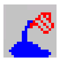
4.2.2 Drawing modes

 Pixel mode. Draw single points.

 Line mode. Start the line with pressing a mouse button (depends on the desired color), draw the line while pressing the button and release when finished. While drawing, a rubber band line is shown from start point to current mouse position. The rubber band has the same line attributes as the final line, so you can judge the result while producing it. This is the same for the next two modes, too.

 Rectangle mode. As with line, simply drag a rectangle to the desired size with left or right mouse button. You can constrain the rectangle to a square by pressing the *Ctrl* key.

 Ellipse mode. Drag a ellipse with the mouse (choose the color by using left or right mouse button). By pressing the *Ctrl* key, you constrain the ellipse to a circle.

 Fill mode. Fill the area of same color as the underlying one with the color of the left or right mouse button. Any other color than the current one will delimit the filling.

4.2.3 Tool menu

You can rotate your pixmap by any degree using the dialog *Rotate...*. Here you could enter a rotation angle or simple flip the pixmap vertical or horizontal.

The *Grid* helps you positioning your pixel more accurately. It displays single points at the edges of zoomed pixels. Therefore grid is disabled when setting zoom to one.

[5] When the text of an indicator disappears, you've most probably choose black as a drawing color

Pixel size... sets some different pixel sizes for drawing. The sizes 1, 2, 4 and 8 are predefined, any other size is available through the *Other sizes...* dialog.

Line Style sets one of the line styles known to Qt.

4.3 Cut & Paste

Selecting parts of pixmaps in QtArchitect offers a nice way of manipulation them. You can select any part of your pixmap and constrain all drawing operations to your selected region. Even filling and rotating is restricted to that region.

To select a region use menu *Edit/Select*. Drag your region like a rectangle using left mouse button. After releasing the mouse, the drawn rectangle will start flashing to mark the borders of your region [6]. *Edit/Deselect* will remove the flashing rectangle and any constraints associated with it. With *Edit/Clip selection*, the flashing rectangle will become the bounding rectangle of all following drawing operations. A mark is shown at the menu entry *Clip selection* when it's active. *Edit/Cut* will move the selected region to an internal buffer. There it will be kept until you use *Deselect*. *Edit/Copy* will copy the selected region. *Edit/Paste* pastes the internal buffer at the current mouse position. To help you positioning the paste buffer, a rectangle is drawn with the size of the internal buffer.

Currently it is not possible to cut and paste between different pixmaps.

4.4 Using Pixmaps with QtArchitect

Currently the only thing that you can do with pixmaps is to *export* them to bitmaps, and then set them as pixmaps in your application. For example: Assume you've build dialog class `MyDialog` with files `MyDialog.cpp`, `MyDialog.h`, ... There's a button in `MyDialog` which you've mapped to `itsColorButton`. The bitmap you've painted is called `ColorButton.bmp`. All you need is to include the following code to your `MyDialog` ctor:

```
.
.
.
  itsColorButton->setPixmap( QPixmap( ``ColorButton.bmp'' ) );
.
.
.
```

[6] This flashing rectangle isn't restricted to exact pixel boundaries, you'll see this when zooming

Now `itsColorButton` should display your pixmap.

This is not the final solution, and it will last only this next version. QtArchitect will generate code and include files for your pixmaps the same way as *qembed* from the trolls does. Then you'll be able to specify a pixmap in the property dialogs and QtArchitect will generate the code for you.

5 Projects

QtArchitect can group dialogs and pixmaps into projects. You can load whole projects at one time and generate all files in project with one mouse click (maybe you need two or three, but “with one mouse click” is better for marketing).

To create a new project, use *File/New Project* or tool bar . A new entry will be appended to the object list with a blue arrow icon pointing down . A double click on a project entry alters the pointer direction and folds a project and its containment to a single line () or shows all included objects ().

All operations on projects (save, close and generate) are handled over to all included objects.

There are several ways (TM by perl) to add objects to a project. The easiest way is to click on the object in the objectlist you wish to enter and drag it with pressed left mouse button over the target project. While dragging, the mouse cursor will change to a small rectangle with a cross in the upper left corner. After releasing the mouse, the dragged object is moved from its start position to the new project. In this context, the free area below all objects is a valid place, too. Objects dragged to this area are removed from all projects they formerly belonged to. You can move objects from one project to another project, too. Only projects can't be nested. Another way is to use the projects context menu and choose *Add Object...*. You'll be prompted by a file selection dialog to choose an already saved dialog or pixmap which will be added.

To remove an object, use *Remove* from the context menu of the object to be removed. The object will appear as a normal object (not indented) in the object list.

5.1 Generating projects

Ahem. Not yet finished is the best what I could say to this subject. I'll rewrite this one to generate input for tmake next version. For now, you can use this to generate all included object with one step, but don't trust the promises of a generated makefile when they appear...

6 Misc.

6.1 Misc.Misc.

There may be some things I left out in this documentation (OK. I'm quite sure that I left some things beside for future books and the earning of big money: "Undocumented QtArchitect "). So you didn't really expect to find any valuable information here, do you ?

7 Inside QtArchitect

You want to look inside QtArchitect ? Good. Help is always welcome !

7.1 Files

QtArchitect consists of appr. 368 files. We grouped them into the following subdirectories:

. main, Project and dialog editor. See section 7.1.1.

./widget All widgets used by the dialog editor. See section 7.1.2.

./prop All widget property dialogs. See section 7.1.3.

./pixmap Pixmap editor. See section 7.1.4.

./layout Layout classes used by the dialog editor. See section 7.1.5.

./doc Documentation. See section 7.1.6.

./bmp All toolbar images. See section 7.1.7.

7.1.1 .

For historic reasons the root directory contains the dialog editor and all main app files. Maybe we change this in the future. The files in detail: `main.cpp` is the starter of QtArchitect . No big deal inside, besides checking command line and calling system without starting GUI if desired. `ProjectWnd` is the first window you see after starting QtArchitect . From here all editors are started and controlled. `DlgObject` ist the base class of all editors which can be controlled by `ProjectWnd`. `Dialog` and `Project` are subclasses from `DlgObject` and are the entry points for the dialog editor and the project management. `ObjectList` is the controller class for the list of loaded or created files in QtArchitect . `DlgParser` is the interface class to `Parser`, a flex generated parser for QtArchitect 's input files. `WidgetLayout` and `WidgetFrame` are used inside the dialog editor. The rest are smaller classes for statusbar, tool bar, grid in the dialog editor and makefile generation.

7.1.2 ./widget

This was the first sub created in QtArchitect . It contains all editable widgets in dialog editor. Each widget has two files here: `Dlgxxx` and `xxx`. The `xxx` files contains widget classes `xxx` subclassed from Qt's `Qxxx` class. We need them to overload mouse event handling. They are instanced by their corresponding `Dlgxxx` classes. These classes control file generation, loading and saving of their widgets to QtArchitect files. They are all subclassed from `DlgWidget` which also contains two factory methods to create specific `Dlgxxx` classes based upon parser input when loading a dialog from a file.

7.1.3 ./prop

All widgets in the dialog editor have property dialogs which can be reached via context menu or menu bar. In prop all widgets have three files for this dialogs: `xxxPage` is the widget specific page interface, `xxxPageData` is the QtArchitect generated page class (Yes, QtArchitect generates itself :) and `xxxPropDialog` is the whole property dialog for widget xxx. If the user wants to see a property dialog, the request is handled from `WidgetLayout` to `DlgWidget` which instanced a `WidgetPropDialog`. This class collects all common pages for a widgets and then ask the `Dlgxxx` class for a list of the widget specific pages. The `Dlgxxx` class returns a list of `xxxPage` objects along with a list of labels for the tabs in the dialog.

7.1.4 ./pixmap

The pixmap editor is build just like the dialog editor: `Pixmap` builds the interface to `ProjectWnd` and is therefore subclassed from `DlgObject`. It instances `PixmapWnd` to display the editor itself. `PixmapWnd` instances `PixmapDraw` for the drawing. There are some smaller classes needed for button which could react on the right mouse button and some dialog classes for pixmap specific things.

7.1.5 ./layout

The layout management classes.

7.1.6 ./doc

The documentation of QtArchitect . As I'm a German, there maybe not only some typos in this documentation, but also some hairy grammatical bugs. If you find the time and send me suggestions for some formulations which I used and that where absolute nonsense for you: You're welcome :)

7.1.7 ./bmp

What should I say: The bitmaps used in the tool bars of QtArchitect are here.

7.2 Class model

First of all: Do you know a good modeling tool for Linux (perhaps even able to handle UML ?).

The QtArchitect core consists of `ProjectWnd`. From here all other objects are produced and maintained. `ProjectWnd` handles `DlgObject` objects which it displays in an `ObjectList` object. When you start a new editor, `ProjectWnd` creates a `Dialog`, `Pixmap` or `Project` object (`DlgObject* ProjectWnd::CreateNewObject( DlgObjectType type )`) and adds it to its internal object list and tell `ObjectList` to add it, too. All relevant signal/slot connections were made in `ProjectWnd::ProcessNewObject( DlgObject *obj )`. These are:

- A `DlgObject` emits `Changed` when its contents was changed by the user. It is connected to `ObjectList` which will display an icon () before the entry for this object.
- A `DlgObject` emits `Saved` when it was saved. It is connected to `ObjectList` which will remove the icon ().
- No `DlgObject` handles its `close` event by itself, so `ProjectWnd` can maintain a list of objects without much effort and simply hide the emitting object.
- `CPDataSet` is emitted when the user cut or copy parts of an editor (Pixmap currently didn't use this feature). It is propagated to all objects with signal `ProjectWnd::CPBufferAvailable(bool)`.

The two editors in QtArchitect have the same class structure, so I explain it only for the dialog editor.

The dialog editor is connected to QtArchitect by the `Dialog` class. This class is subclassed from `DlgObject` so it can be accessed from QtArchitect like `Project` or `Pixmap`. `Dialog` by itself has no gui functionality, but instance a `DialogWnd` for this reason. `DialogWnd` instance now a `WidgetLayout` as its drawing area. Nearly all menu callbacks are located in `DialogWnd`, some of them are in `Dialog` (eg. save, ...) and others in `WidgetLayout` for convenience of programming. If the user creates a new widget, eg. a new `QPushbutton`, `DialogWnd` sends this request over to `WidgetLayout::AddWidget( DlgWidget::type )` with the requested type as the argument. From here, `static DlgWidget::NewDlgWidget( QWidget *, DlgWidget::type, const QString *)` is called to construct the requested `DlgPushButton` with the first parameter as parent and the last as its name (We have to use a pointer here, since `DlgWidget`'s are capable of creating default names if necessary). Now a corresponding widget class is created, in our example `PushButton` which is a subclass from the Qt class QPushbutton (these intermediate classes are needed for method overloading).