Qt/Embedded Whitepaper

Trolltech

www.trolltech.com

Abstract

This whitepaper describes the Qt/Embedded C++ toolkit for GUI and application development on embedded devices. It runs on any device supported by Linux and a C++ compiler. Qt/Embedded provides the entire standard Qt API and can compile out unused features to minimize its memory footprint. Qt/Embedded provides its own windowing system which is far more compact than Xlib and the X Window System that it replaces. Qt/Embedded applications can be developed on familiar desktop systems, e.g. Windows and Unix, and with standard tools. It is provided with all the Qt tools including *Qt Designer* for visual form design, and with tools specifically tailored to the embedded environment.

The Sharp Zaurus PDA using Qt/Embedded

Qt/Embedded Whitepaper

Trolltech

www.trolltech.com

Contents

1. Introduction

Qt/Embedded is a C++ toolkit for GUI and application development for embedded devices. It runs on a variety of processors, usually with Embedded Linux. Qt/Embedded applications write directly to the frame-buffer, eliminating the need for the X Window System. In addition to the class library, Qt/Embedded includes several tools to speed and ease development. Applications can be developed with familiar programming environments on Windows and Unix, using the standard Qt API.

Qt/Embedded is a port of the Qt C++ API for embedded devices. It provides the same API and tools as the Qt/X11, Qt/Windows and Qt/Mac versions. Qt/Embedded also includes classes and tools to specifically support embedded development.

The Qt C++ toolkit upon which Qt/Embedded is built has been at the heart of commercial applications since 1995. Qt is used by enterprises as diverse as AT&T, IBM, NASA, Sharp and Xerox, and by numerous smaller companies and organizations. Qt 3.0 retains the power and ease of use of earlier versions and introduces many new classes. Qt's classes are fully featured to reduce developer workload, and provide consistent interfaces to speed learning. Qt is, and always has been, fully object-oriented.

Qt provides a type-safe alternative to old fashioned callbacks, called signals and slots [p. 8], that facilitates true component programming. Qt supplies a wide range of versatile widgets [p. 11] that can easily be subclassed to create custom components, or combined to create custom dialogs [p. 19]. Pre-defined dialogs for common tasks such as message boxes and wizards are also provided.

Qt/Embedded has much smaller system requirements [p. 4], i.e. lower storage (Flash) and memory (RAM) footprints, than embedded solutions based on the X Window System. It can run on hardware that runs Linux, has a linearly addressable framebuffer, and supports a C++ compiler. And Qt/Embedded can be recompiled to exclude unused features to reduce its memory footprint even further.

The architecture [p. 5] of Qt/Embedded includes its own windowing system [p. 6]. A variety of input devices [p. 7] are supported.

Developers write code using their familiar development environments [p. 8]. *Qt Designer* [p. 22] can be used to visually design user interfaces using Qt's layout [p. 19] system, which automatically adapts to the available screen space. Developers can choose one of the pre-defined look and feel [p. 24] styles or create their own unique styles. Unix users can run and test their applications on a pixel-perfect virtual frame-buffer.

Qt/Embedded also provides many non-graphical components [p. 28] for specialized tasks, such as internationalization [p. 25], networking and database interaction.

Qt/Embedded is a mature, solid C++ toolkit, widely used all over the world [p. 30]. In addition to Qt/Embedded's many other commercial uses, it is the foundation of the Qtopia application environment for small devices. Qt/Embedded makes application development a pleasure, with its simple build system, visual form design and elegant API.

2. System Requirements

Qt/Embedded saves memory because it does not need an X server or Xlib; instead it writes directly to the frame-buffer. Memory consumption can be fine-tuned by compiling out features that are not used. It is also possible to compile all the applications into a single statically linked executable, to save even more memory.

Qt/Embedded is available for all processors supported by Linux that have a C++ compiler, including Intel x86, MIPS, ARM, StrongARM, Motorola 68000 and PowerPC. Trolltech is also exploring the possibility of creating a cross platform toolkit for the embedded market. Qt/Embedded implementations for QNX and for WinCE are both being trialed. Trolltech also provides porting services to other operating systems.

Qt/Embedded applications write directly to the kernel frame-buffer. Linear frame-buffers with 1, 4, 8, 15, 16, 24 and 32 bit depths and VGA16 are supported. Any graphic card supported by the kernel will work, and Qt/Embedded can be customized to benefit from screen acceleration hardware, as described in "Architecture" [p. 5]. There is no arbitrary limit on screen size, and many advanced features such as anti-aliased fonts, alpha-blended pixmaps and screen rotation are provided.

Qt/Embedded's principal strength is that is doesn't rely on an X server. This leads to significant memory savings compared with other solutions, such as Qt/X11. A single library, the Qt/Embedded library, is all that is necessary to replace the X server, the Xlib library and the widget toolkit of other 'embedded' solutions.

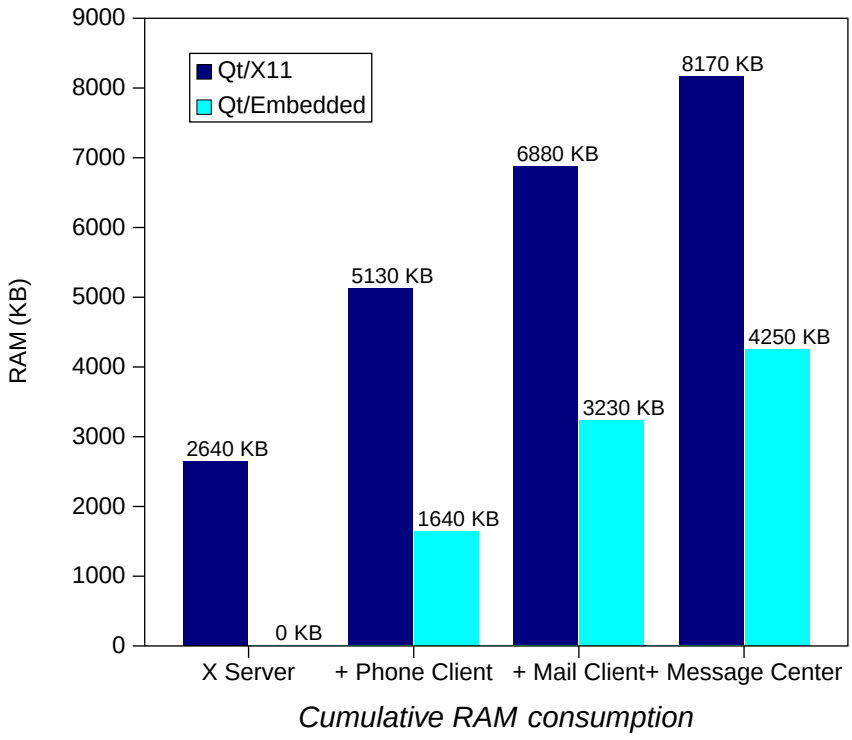

Figure 1. Memory comparison between Qt/X11 and Qt/Embedded for Ericsson's screen phone

The graph illustrates that the X server grabs a lot of RAM on startup, and also requires more memory as each new application is launched. For example, starting the Phone Client requires 2490 KB with Qt/X11, but only 1640 KB with Qt/Embedded.

The footprint of the Qt/Embedded library can be reduced by compiling out unused features. For example, the **QListView** widget can be compiled out by defining the pre-processor symbol `QT_NO_LISTVIEW`, and support for internationalization is compiled out by defining `QT_NO_I18N`. Qt/Embedded provides over 200 configurable features, resulting in libraries varying in size between 700 KB and 5000 KB (Intel x86). Most customers use configurations between 1500 KB and 4000 KB.

Qt/Embedded also benefits from memory-saving techniques such as implicit sharing (copy on write) and caching. Over 20 classes in Qt, including **QBitmap**, **QMap**, **QPalette**, **QPicture**, **QPixmap** and **QString**, use implicit sharing to avoid unnecessary copying and minimize memory usage. Implicit sharing occurs automatically and makes programming much simpler, avoiding the risks related to hand optimization and pointers.

Many Qt components can be compiled into the library or made available as plugins. Custom look and feel components [p. 24], database drivers, font format readers, image format converters, text codecs and widgets can be compiled as plugins, reducing the size of the core library and providing more flexibility. Alternatively, if the applications and components are known in advance, they can be compiled and statically linked with the Qt/Embedded library into a single executable, saving ROM, RAM and CPU.

3. Architecture

Qt/Embedded provides the standard Qt API for embedded devices with a lightweight windowing system. Qt/Embedded's object-oriented design makes it straightforward to support additional devices, from peripherals like keyboards and mice to accelerated graphics boards.

With Qt/Embedded, developers benefit from exactly the same API that Qt/X11, Qt/Windows and Qt/Mac provide.

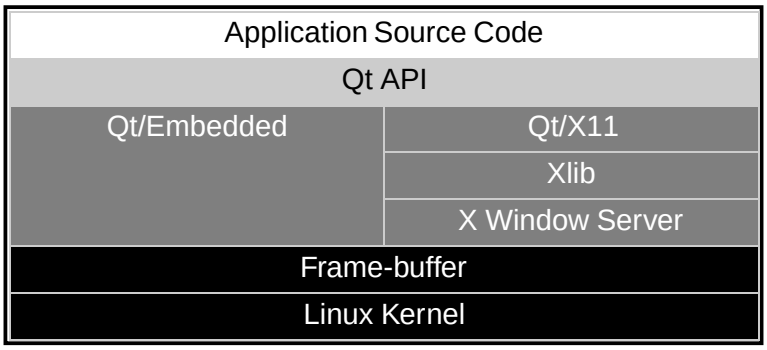

Figure 2. Qt/Embedded versus Qt/X11 on Embedded Linux

Using a single API across a variety of platforms offers many benefits. Companies that produce applications for both embedded devices and desktop computers can train their developers in a single toolkit. This makes it easier to share experience and knowledge, and gives managers more flexibility when allocating developers to projects. Furthermore, applications and

components developed for a particular platform can be sold for any of the other Qt platforms, expanding the products' market for a very low marginal cost.

3.1. Windowing System

A Qt/Embedded windowing system consists of one or more processes, one of which acts as a server. The server allocates regions to be displayed by clients, and generates mouse and keyboard events. The server process can also provide input methods and a user interface to launch client applications. The server process behaves like a client but has some additional privileges. Any program can be run as the server using the `-qws` command-line option.

Clients communicate with the server using shared memory. Communication is kept to a minimum; clients perform all drawing operations directly to the frame-buffer, without passing through the server, and are responsible for drawing their own title bars and other decorations. This is all handled transparently by the Qt/Embedded library.

Clients can exchange messages using QCOP channels. The server simply broadcasts QCOP messages to all applications listening to a given channel. Applications can respond in a slot connected to a received() signal. Messages can be accompanied by binary data, typically serialized using the **QDataStream** class, described in "Non-Graphical Classes" [p. 28].

The **QProcess** class provides another asynchronous inter-process communication mechanism. It is used to start external programs and to communicate with them by writing to their standard input and by reading their standard output and standard error.

3.2. Fonts

Qt/Embedded supports four different font formats: TrueType Fonts (TTF), PostScript Type1 Fonts, Bitmap Distribution Format (BDF) and Qt Pre-rendered Fonts (QPF). Support for other font formats can be added by subclassing **QFontFactory,** and can be made available as a plugin. Anti-aliased fonts are supported.

Each TTF or Type1 glyph is rendered at a given point size when it is first used in a drawing or metrics operation, and the result is cached. Memory and CPU time can often be saved by pre-rendering a TTF or a Type1 file at the required sizes (for example, 10 and 12 points) and saving the result in QPF format. QPF files that contain the necessary fonts can be obtained by using the `makeqpf` tool, or by running applications with the `-savefonts` option. If all the fonts are in QPF format, Qt/Embedded can be reconfigured to compile out support for TTF and Type1 fonts, which will cut down the size of the Qt/Embedded library, and considerably reduce the amount of memory used to store fonts. For example, a 10-point Times QPF font for ASCII uses about 1300 bytes, and is directly mapped into memory from physical storage.

Qt/Embedded fonts usually contain a small subset of Unicode, typically ASCII or Latin-1. A complete 16-point Unicode font uses over 1 MB of memory. It is possible to save custom subsets of a font, for example one that contains all the glyphs necessary to spell the name of your product in 24-point Cappuccino Bold.

3.3. Input Devices

Qt/Embedded 3.0 supports several mouse protocols out of the box: BusMouse, IntelliMouse, Microsoft and MouseMan. Qt/Embedded also supports the NEC Vr41XX touch-panel and the iPAQ touch-panel. Developers can support custom pointer devices by subclassing **QWSMouseHandler** or **QCalibratedMouseHandler**.

Qt/Embedded supports the standard 101-key keyboard and Vr41XX buttons. Custom keyboards and other non-pointer devices can be supported by subclassing **QWSKeyboardHandler**.

3.4. Input Methods

Input methods for non-Latin scripts (for example, Arabic, Chinese, Hebrew and Japanese) can be written to filter and convert keyboard input. Input method writers have the entire Qt API at their disposal.

On devices without a keyboard, input methods constitute the only means of entering characters. Qtopia provides four input methods: a handwriting recognizer, a graphical QWERTY keyboard, a Unicode keyboard and a dictionary-based pickboard.

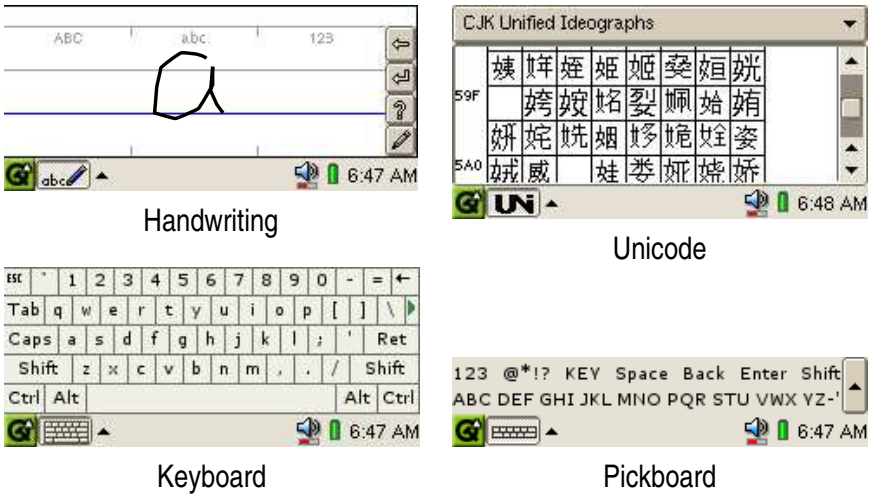

Figure 3. The standard input methods available on Qtopia

3.5. Screen Acceleration

Screen operations can benefit from hardware acceleration by subclassing **QScreen** and **QGfxRaster**. Trolltech provides example accelerated drivers for Mach64 and Voodoo3 cards, and can be contracted to write custom drivers.

4. Development Environment

Qt/Embedded development can take place using familiar Unix and Windows tools. Several multi-platform tools are provided to make development easier and faster, notably Qt Designer. Unix users additionally benefit from a virtual frame-buffer that duplicates, pixel for pixel, the screen of a device.

Applications for an embedded device can be compiled on any platform equipped with a cross-development tool chain. The most common option is to build a cross-platform GNU C++ compiler (g++) with libc and the binary utilities on a Unix system.

An alternative approach involves using a desktop version of Qt, such as Qt/X11 or Qt/Windows, for most of the development phase. This allows developers to use a familiar environment, for example, Microsoft Visual C++ or Borland C++. On Unix, many environments are available, such as KDevelop, which supports cross-development.

If the Qt/Embedded application is developed on Unix, it can be compiled to run on the development machine in a separate console or in the virtual frame-buffer, an X11 application that simulates a frame-buffer. By specifying the device's width, height and color depth, the simulated frame-buffer will match the physical device, pixel for pixel. This saves developers from continuously re-flashing the device, and accelerates the compile, link and run cycle. It also allows developers to use standard debuggers and profilers on the development machine. If desired, Qt/Embedded applications can act as VNC (Virtual Network Computing) servers and be run over a network.

4.1. Qt's Supporting Tools

Qt includes many tools to support embedded systems development, some of which are mentioned elsewhere in this document. The two most substantial tools (apart from the virtual frame-buffer mentioned above) are `qmake` and *Qt Designer*.

The `qmake` tool is a Makefile generator for the Qt/Embedded library and for applications. It generates Makefiles for multiple platforms from a project file (`.pro`). `qmake` supports cross-development and shadow builds, and makes it easy to switch between different configurations.

Developers can use *Qt Designer* to design dialogs visually instead of writing code. It uses Qt's layout managers to produce dialogs that resize smoothly, and is fully integrated with `qmake`. *Qt Designer* is covered in "Dialogs" [p. 19].

5. Signals and Slots

The signals and slots mechanism provides inter-object communication. It is easy to understand and use and it is fully supported by Qt Designer.

GUI applications respond to user actions. For example, when a user clicks a menu item or toolbar button, the application executes some code. More generally, we want objects of any kind to communicate with each other. The programmer must relate events to the relevant code. Older toolkits use mechanisms that are crash-prone, inflexible, and not object-oriented. Trolltech has

invented a solution called 'signals and slots'. Signals and slots is a powerful inter-object communication mechanism that can be used to completely replace the crude callbacks and message maps used by legacy toolkits. Signals and slots are fast, type-safe, flexible, fully object-oriented and implemented in C++.

To associate some code with a button using the old callback mechanism, it is necessary to pass a pointer to a function to the button. When the button is clicked, the function is then called. Old toolkits do not ensure that arguments of the right type are given to the function when it is called, which makes crashes more likely. Another problem with the callback approach is that it tightly binds the GUI element to the functionality, making it difficult to develop classes independently.

Qt's signals and slots mechanism is different. Qt widgets emit signals when events occur. For example, a button will emit a 'clicked' signal when it is clicked. The programmer can choose

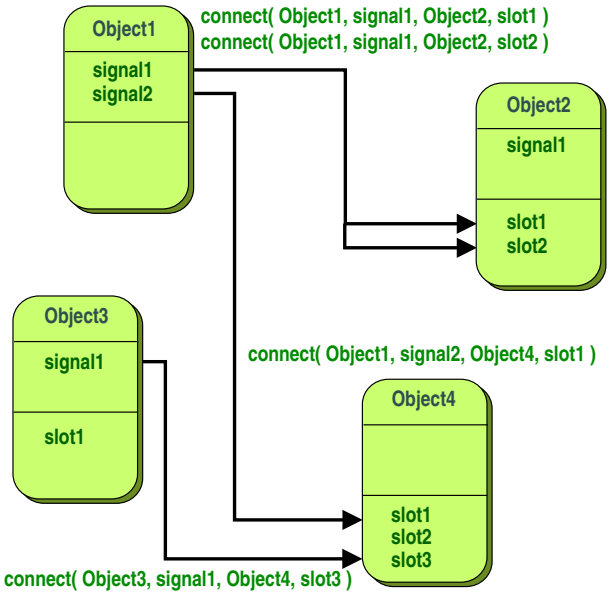

Figure 4. An abstract view of some signals and slots connections

to connect to a signal by creating a function (called a slot) and calling the connect() function to relate the signal to the slot. Qt's signals and slots mechanism does not require classes to have knowledge of each other, which makes it much easier to develop highly reusable classes. Signals and slots are type-safe, with type errors being reported by warnings rather than by crashes.

For example, if a Quit button's clicked() signal is connected to the application's quit() slot, a user's click on Quit makes the application terminate. In code, this is written as

```
connect( button, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

Connections can be added or removed at any time during the execution of a Qt application.

The signals and slots implementation smoothly extends C++'s syntax and takes full advantage of C++'s object-oriented features. Signals and slots can be overloaded or reimplemented and may appear in the public, protected or private sections of a class.

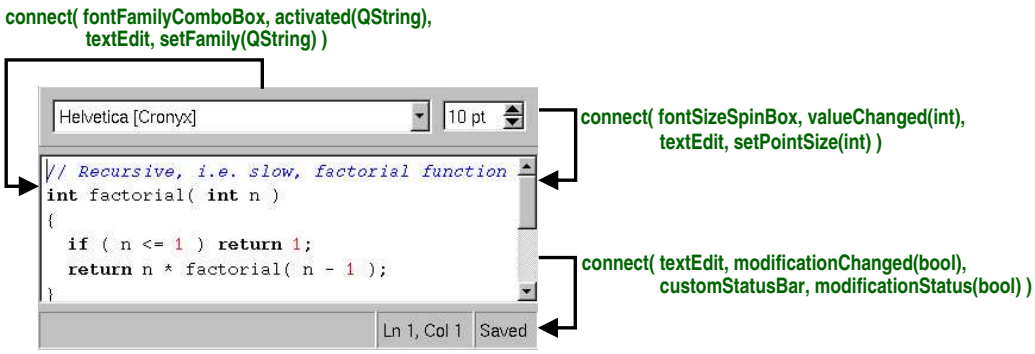

Figure 5. An example of signals and slots connections

5.1. A Signals and Slots Example

To benefit from signals and slots, a class must inherit from **QObject** or one of its subclasses and include the `Q_OBJECT` macro in the class's definition. Signals are declared in the `signals` section of the class, while slots are declared in the `public slots`, `protected slots` or `private slots` sections.

Here's an example **QObject** subclass:

```
class BankAccount : public QObject
{
    Q_OBJECT
public:
    BankAccount() { curBalance = 0; }
    int balance() const { return curBalance; }
public slots:
    void setBalance( int newBalance );

signals:
    void balanceChanged( int newBalance );

private:
    int curBalance;
};
```

In the style of most C++ classes, the class **BankAccount** has a constructor, a get function balance(), and a set function setBalance().

The class also has a signal balanceChanged(), which announces that the balance in the account has changed. Signals are not implemented; when a signal is emitted, the slots it is connected to are executed.

The set function is declared in the `public slots` section, so it is a slot. Slots are standard member functions with an implementation that can be called like any other function, and which can also be connected to signals.

Here's the implementation of the slot setBalance():

```
void BankAccount::setBalance( int newBalance )
{
```

```
        if ( newBalance != curBalance ) {
            curBalance = newBalance;
            emit balanceChanged( curBalance );
        }
    }
```

The statement

```
emit balanceChanged( curBalance );
```

causes the balanceChanged() signal to be emitted with the new current balance as its argument. The keyword `emit`, like `signals` and `slots`, is provided by Qt and is transformed into standard C++ by the C++ pre-processor.

One object's signal can be connected to many different slots, and many signals can be connected to one slot in a particular object. Connections are made between signals and slots whose parameters have the same types. A slot can have fewer parameters than the signal and ignore the extra parameters.

5.2. Meta Object Compiler

The signals and slots mechanism is implemented in pure standard C++. The implementation uses the C++ pre-processor and the Meta Object Compiler (`moc`) included with the Qt toolkit.

The `moc` reads the application's header files and generates the necessary code to support signals and slots. Developers never edit or even need to look at the generated code. Makefiles generated by `qmake` include rules to run `moc` transparently, when required.

In addition to handling signals and slots, `moc` supports Qt's translation mechanism, its property system and run-time type information.

6. Widgets

Qt has a rich set of widgets (buttons, scroll bars, etc.) that cater for most situations. Qt's widgets are flexible and easy to subclass for special requirements.

Widgets are instances of **QWidget** or one of its subclasses, and custom widgets are created by subclassing.

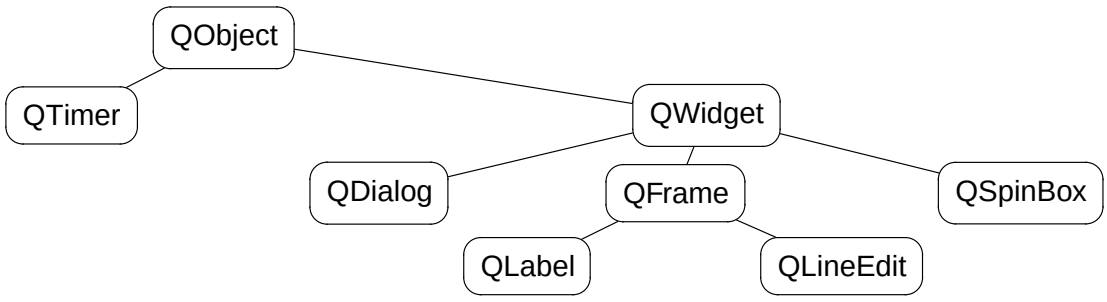

Figure 6. An extract from the **QWidget** class hierarchy

A widget may contain any number of child widgets. Child widgets are shown within the parent widget's area. A widget with no parent is a top-level widget (a 'window'), and is decorated with a configurable frame and title bar. Qt imposes no arbitrary limitations on widgets. Any widget can be a top-level widget; any widget can be a child of any other widget. The position of child widgets within the parent's area can be set automatically using layout managers [p. 19], or manually if preferred. When a parent widget is disabled, hidden or deleted, the same action is applied to all its child widgets recursively.

Labels, message boxes, tooltips, etc., are not confined to using a single color, font and language. Qt's text-rendering widgets can display multi-language rich text using a HTML subset.

6.1. A 'Hello' Example

The complete source code for a program that displays "Hello Qt/Embedded!" follows:

Figure 7. Hello Qt/Embedded!

```
#include <qapplication.h>
#include <qlabel.h>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel *hello = new QLabel( "<font color=blue>Hello"
                                " <i>Qt/Embedded!</i></font>", 0 );
    app.setMainWidget( hello );
    hello->show();
    return app.exec();
}
```

6.2. Common Widgets

The screenshots below present the main Qt widgets, shown using the Windows style.

Figure 8. A **QLabel** and a **QPushButton** laid out with a **QHBox**

Figure 9. Two **QRadioButton**s and two **QCheckBox**es laid out with a **QButtonGroup**

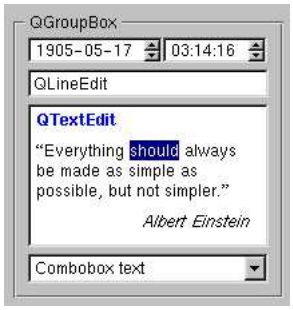

Figure 10. A **QDateTimeEdit**, a **QLineEdit**, a **QTextEdit** and a **QComboBox** laid out with a **QGroupBox**

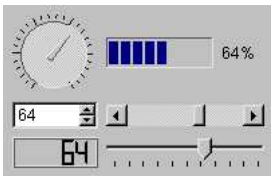

Figure 11. A **QDial**, a **QProgressBar**, a **QSpinBox**, a **QScrollBar**, a **QLCDNumber** and a **QSlider** laid out with a **QGrid**

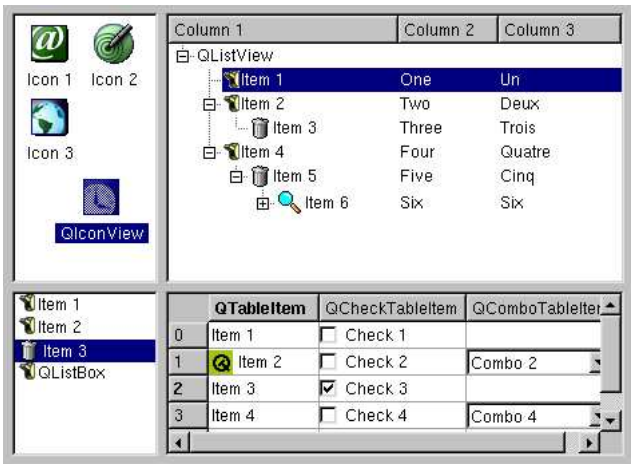

Figure 12. A **QIconView**, a **QListView**, a **QListBox** and a **QTable** laid out with a **QGrid**

QComboBox, **QLineEdit** and **QSpinBox**'s input can be constrained or validated using a **QValidator** subclass. Regular expressions can be used for validation.

QTable, **QListView**, **QTextEdit** and other widgets that can display large amounts of data inherit **QScrollView** and automatically provide scroll bars.

Many of Qt's built-in widgets can display images, for example, buttons, labels, menu items, etc. The **QImage** class supports the input, output and manipulation of images in several formats, including BMP, GIF*, JPEG, MNG, PNG, PNM, XBM and XPM.

6.3. Canvas

The **QCanvas** class provides a high-level interface to 2D graphics. It can handle a very large number of 'canvas items' that represent lines, rectangles, ellipses, texts, pixmaps, animated sprites, etc. Canvas items can easily be made interactive (e.g. user movable).

Figure 13. The Qtopia Asteroids game written with **QCanvas**

Canvas items are instances of **QCanvasItem** subclasses. They are more lightweight than widgets, and they can be quickly moved, hidden and shown. **QCanvas** has efficient support for collision detection, and can list all the canvas items in a given area. **QCanvasItem** can be subclassed to provide custom item types and to extend the functionality of existing types.

QCanvas objects are rendered by the **QCanvasView** class. Many **QCanvasView** objects can show the same **QCanvas**, but with different translations, scales, rotations and shears.

QCanvas is ideal for data visualization. It has been used by customers for drawing road maps and for presenting network topologies. It is also suitable for fast 2D games with lots of sprites.

6.4. Custom Widgets

Developers can create their own widgets and dialogs by subclassing **QWidget** or one of its subclasses. To illustrate subclassing, the complete code for an analog clock widget is presented. The **AnalogClock** widget displays the current time and updates itself automatically.

Figure 14. Analog clock widget

In `analogclock.h`, **AnalogClock** is defined like this:

*If you are in a country that recognizes software patents and where Unisys holds a patent on LZW decompression, Unisys may require you to license the technology to use GIF.

```
#include <qwidget.h>

class AnalogClock : public QWidget
{
public:
    AnalogClock( QWidget *parent = 0, const char *name = 0 );

protected:
    virtual void timerEvent( QTimerEvent *event );
    virtual void paintEvent( QPaintEvent *event );
};
```

AnalogClock inherits **QWidget**. It has a constructor typical of widget classes, with optional `parent` and `name` parameters. (Testing and debugging are easier if `name` is set.) The timerEvent() function is inherited from **QObject** (a base class of **QWidget**) and is called at regular intervals by the system. The paintEvent() function is inherited from **QWidget** and is called automatically whenever the widget needs to be redrawn.

The timerEvent() and paintEvent() functions are two examples of 'event handlers'. Application objects receive system messages as Qt events (**QEvent** objects). There are over fifty types of event, of which the most commonly used are `MouseButtonPress`, `MouseButtonRelease`, `KeyPress`, `KeyRelease`, `Paint`, `Resize` and `Close`. Objects can respond to events sent to them, and filter events destined for other objects.

In `analogclock.cpp`, the functions declared in `analogclock.h` are implemented:

```
#include <qdatetime.h>
#include <qpainter.h>

#include "analogclock.h"

AnalogClock::AnalogClock( QWidget *parent, const char *name )
    : QWidget( parent, name )
{
    startTimer( 12000 );
    resize( 100, 100 );
}

void AnalogClock::timerEvent( QTimerEvent * )
{
    update();
}

void AnalogClock::paintEvent( QPaintEvent * )
{
    QCOORD hourHand[8] = { 2, 0, 0, 2, -2, 0, 0, -25 };
    QCOORD minuteHand[8] = { 1, 0, 0, 1, -1, 0, 0, -40 };
    QTime time = QTime::currentTime();

    QPainter painter( this );
```

```
    painter.setWindow( -50, -50, 100, 100 );
    painter.setBrush( black );

    for ( int i = 0; i < 12; i++ ) {
        painter.drawLine( 44, 0, 46, 0 );
        painter.rotate( 30 );
    }

    painter.save();
    painter.rotate( 30 * (time.hour() % 12) + time.minute() / 2 );
    painter.drawConvexPolygon( QPointArray(4, hourHand) );
    painter.restore();

    painter.save();
    painter.rotate( 6 * time.minute() );
    painter.drawConvexPolygon( QPointArray(4, minuteHand) );
    painter.restore();
}
```

The constructor tells the system to call timerEvent() every twelve seconds to refresh the clock, and sets the widget's default size to 100 x 100.

In timerEvent(), the **QWidget** function update() is called to tell Qt that the widget needs to be repainted. Subsequently, Qt will generate a paint event and call paintEvent().

In paintEvent(), a **QPainter** object is used to draw the twelve notches and the time and minute hands on the widget. The **QPainter** class provides an API for painting widgets, pixmaps, vector images and PostScript in a uniform way. It provides functions to draw points, lines, polygons, ellipses, arcs, Bezier curves, etc. The coordinate system of a **QPainter** can be translated, scaled, rotated and sheared; the objects drawn can be clipped according to a 'window', and positioned on the widget using a 'viewport'. Clipping can be used to reduce flicker when repainting. An area of the frame-buffer can be locked and accessed directly using the **QDirectPainter** subclass of **QPainter**.

The files `analogclock.h` and `analogclock.cpp` completely define and implement the **AnalogClock** custom widget. This widget can be used immediately:

```
#include <qapplication.h>

#include "analogclock.h"

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    AnalogClock *clock = new AnalogClock;
    app.setMainWidget( clock );
    clock->show();
    return app.exec();
}
```

6.5. Main Windows

The **QMainWindow** class lays out a set of related widgets to provide a framework for typical application main windows.

A main window contains a set of standard widgets. The top of the main window contains a menu bar, beneath which toolbars are laid out. The toolbars can be moved to any dock area; main windows have dock areas at the top, left, right and bottom. Toolbars can also be dragged out of a dock area and floated as independent tool palettes. The bottom of the main window, below the bottom dock area, is occupied by a status bar. The central area contains any widget. Tooltips and "What's this?" help provide balloon help for the user-interface elements.

For small screen devices, it can be preferable to define a standard **QWidget** template in *Qt Designer* and use that, rather than **QMainWindow**. The template typically has a menu bar and a toolbar side by side, and may not have a status bar at all. (Where necessary, status may be shown in the task bar or the title bar, for example.)

6.6. Menus

The **QPopupMenu** widget presents menu items to the user in a vertical list. Popup menus can be standalone (e.g. a context menu), can appear in a menu bar, or can be a sub-menu of another popup menu.

Each menu item can have an icon, a checkbox and an accelerator. Menu items usually correspond to actions (e.g. Save). Separator items are displayed as a line and are used to visually group related actions.

Here's an example that creates a File menu with **New**, **Open** and **Exit** menu items:

```
QPopupMenu *fileMenu = new QPopupMenu( this );
fileMenu->insertItem( "&New", this, SLOT(newFile()), CTRL+Key_N );
fileMenu->insertItem( "&Open...", this, SLOT(open()), CTRL+Key_O );
fileMenu->insertSeparator();
fileMenu->insertItem( "E&xit", qApp, SLOT(quit()), CTRL+Key_Q );
```

When a menu item is chosen, the corresponding slot is executed. As accelerators are rarely used on devices with no keyboard, Qt/Embedded is typically configured without accelerator support. This means that whereas "&New" would be rendered as **New** on a desktop machine, it will appear as **New** on an embedded device.

The **QMenuBar** class implements a menu bar. It automatically sets its geometry to the top of its parent widget. It splits its contents across multiple lines if the parent window is not wide enough. Qt's built-in layout managers automatically take the menu bar into consideration.

Qt's menu system is very flexible. Menu items can be enabled, disabled, added or removed dynamically. Menu items with customized appearance and behavior can be created by subclassing **QCustomMenuItem**.

6.7. Toolbars

The **QToolButton** class implements a toolbar button with an icon, a 3D frame and an optional label. Toggle toolbar buttons turn features on and off. Other toolbar buttons execute a command. Different icons can be provided for the active, disabled and enabled modes, and for the on and off states. If only one icon is provided, Qt automatically distinguishes the state using visual cues,

for example, graying out disabled buttons. Pressing a toolbar button can also be used to trigger a popup menu.

QToolButtons usually appear side-by-side within a **QToolBar**. An application can have any number of toolbars, and the user is free to move them around. Toolbars can contain widgets of almost any type, for example **QComboBox**es and **QSpinBox**es.

6.8. Balloon Help

Modern applications use balloon help to briefly explain the purpose of user-interface elements. Qt provides two mechanisms for balloon help: tooltips and "What's this?" help.

Tooltips are small, usually yellow, rectangles that appear automatically when the mouse pointer hovers over a widget. Tooltips are often used to explain a toolbar button, since toolbar buttons are rarely displayed with text labels. Here's how to set the tooltip of a 'Save' toolbar button:

```
QToolTip::add( saveButton, "Save" );
```

It is also possible to set a longer piece of text to be displayed in the status bar when the tooltip is shown.

Devices that do not use a mouse (for example, those that use a stylus), may not have a means of hovering the mouse pointer over a widget, which is the normal mechanism for raising a tooltip. Such devices may not support tooltips at all (relying on "What's this?" help instead), or may use a gesture, for example, press and hold, to signify hovering.

"What's this?" help is similar to tooltips, except that the user must request it. On a small screen device, "What's this?" help may be invoked by pressing a **?** help button that appears next to the application's **X** close button, and then pressing the relevant widget. "What's this?" help is typically longer than a tooltip. Here's how to set the "What's this?" text for a 'Save' toolbar button:

```
QWhatsThis::add( saveButton, "Saves the current file." );
```

The **QToolTip** and **QWhatsThis** classes provide virtual functions that can be reimplemented for more specialized behavior.

Qtopia doesn't use either of these mechanisms to provide help. Instead it provides a ? help button in each application's title bar, which launches the HTML help browser with the help contents page for the relevant application. It uses the press and hold gesture to invoke context (right click) menus and property dialogs.

6.9. Actions

Applications usually provide the user with several different ways to perform a particular action. For example, most applications provide a 'Save' action available from the menu (File | Save), from the toolbar (the 'floppy disk' toolbar button) and as an accelerator (Ctrl+S). The **QAction** class encapsulates this concept. It allows programmers to define an action in one place and then add that action to a menu or toolbar. Actions that only make sense as menu options can be added to menus directly.

The following code implements a 'Save' menu item and a 'Save' toolbar button. Balloon help and an accelerator could easily be added, but are not included because they are rarely used for small devices.

```
QAction *saveAct = new QAction( this );
saveAct->setText( "Save" );
saveAct->setIconSet( QPixmap("save.png") );
connect( saveAct, SIGNAL(activated()), this, SLOT(save()) );
saveAct->addTo( fileMenu );
saveAct->addTo( toolbar );
```

In addition to avoiding duplication, using a **QAction** ensures that the state of menu items stays in sync with the state of toolbar buttons, and that tooltips are displayed when necessary. Disabling an action will disable any corresponding menu items and toolbar buttons. Similarly, if the user clicks a toggle toolbar button, the corresponding menu item will be checked or unchecked accordingly.

7. Dialogs

Developers can build their own dialogs using the Qt Designer visual design tool. Qt uses 'layouts' to automatically size and position widgets in relation to one another. This ensures that dialogs make the best use of the available screen space. The use of layouts also means that buttons and labels automatically resize to show their text in full regardless of language.

7.1. Layouts

Qt provides layout managers for organizing child widgets within the parent widget's area. They feature automatic positioning and resizing of child widgets, sensible minimum and default sizes for top-level widgets, and automatic repositioning when the contents or the font changes.

Using layouts, developers can write applications independently of the screen size or orientation, without wasting space or duplicating code. For internationalized applications, layouts ensure that buttons and labels take as little space as possible without cutting off the text, regardless of the language.

Layouts also make it easy to accommodate certain user-interface components such as input methods and task bars. For example, when Qtopia users are entering text, the input method takes up screen space, and the application should adapt accordingly.

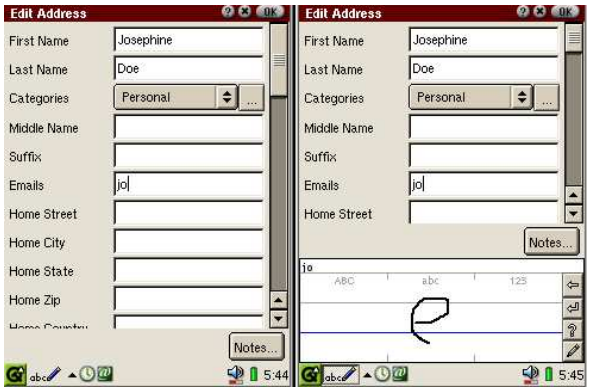

Figure 15. Layout management on Qtopia

Qt provides three built-in layout managers: **QHBoxLayout**, **QVBoxLayout** and **QGridLayout**.

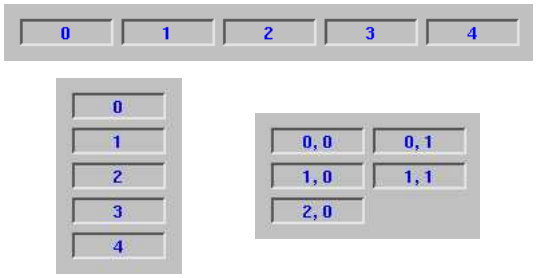

Figure 16. **QHBoxLayout, QVBoxLayout** and **QGridLayout**

QHBoxLayout organizes the managed widgets in a single horizontal row from left to right. **QVBoxLayout** organizes the managed widgets in a single vertical column, from top to bottom. **QGridLayout** organizes the managed widgets in a grid of cells; widgets may span multiple cells.

In most cases, Qt's layout managers pick optimal sizes for managed widgets so that windows look good and resize smoothly. Developers can refine the layout using the following mechanisms:

1. *Setting a minimum size, a maximum size or a fixed size for some child widgets.*

2. *Adding stretch items or spacer items.* Stretch or spacer items fill empty space in a layout.

3. *Changing the size policies of the child widgets.* Programmers can fine tune the resize behavior of a child widget. Child widgets can be set to expand, contract, keep the same size, etc.

4. *Changing the child widgets' size hints.* QWidget::sizeHint() and QWidget::minimumSizeHint() return a widget's preferred size and preferred minimum size based on the contents. Built-in widgets provide appropriate reimplementations.

5. *Setting stretch factors.* Stretch factors allow relative growth of child widgets, e.g. two thirds of any extra space made available should be allocated to widget A and one third to widget B.

Layouts can also run right-to-left and bottom-to-top. Right-to-left layouts are convenient for internationalized applications supporting right-to-left languages such as Arabic and Hebrew.

Layouts can be nested to arbitrary levels. Here's an example of a dialog box, shown at two different sizes:

Figure 17. Small dialog and large dialog

The dialog uses three layouts: a **QVBoxLayout** that groups the push buttons, a **QHBoxLayout** that groups the country listbox with the push buttons and a **QVBoxLayout** that groups the "Now please select a country" label with the rest of the widget. A stretch item maintains the gap between the < Prev and Help buttons.

The dialog's widgets and layouts are created with the following code:

```
QVBoxLayout *buttonBox = new QVBoxLayout( 6 );
buttonBox->addWidget( new QPushButton("Next >", this) );
buttonBox->addWidget( new QPushButton("< Prev", this) );
buttonBox->addStretch( 1 );
buttonBox->addWidget( new QPushButton("Help", this) );

QListBox *countryList = new QListBox( this );
countryList->insertItem( "Canada" );
/* … */
countryList->insertItem( "United States of America" );

QHBoxLayout *middleBox = new QHBoxLayout( 11 );
middleBox->addWidget( countryList );
middleBox->addLayout( buttonBox );

QVBoxLayout *topLevelBox = new QVBoxLayout( this, 6, 11 );
topLevelBox->addWidget( new QLabel("Now please select a country", this)
);
topLevelBox->addLayout( middleBox );
```

Alternatively, the dialog can be designed using *Qt Designer* with just 17 mouse clicks.

Figure 18. Laying out a form in *Qt Designer*

7.2. *Qt Designer*

Qt Designer is a visual user-interface design tool. Qt applications can be written entirely in source code, or using *Qt Designer* to speed up development. Designing a form with *Qt Designer*

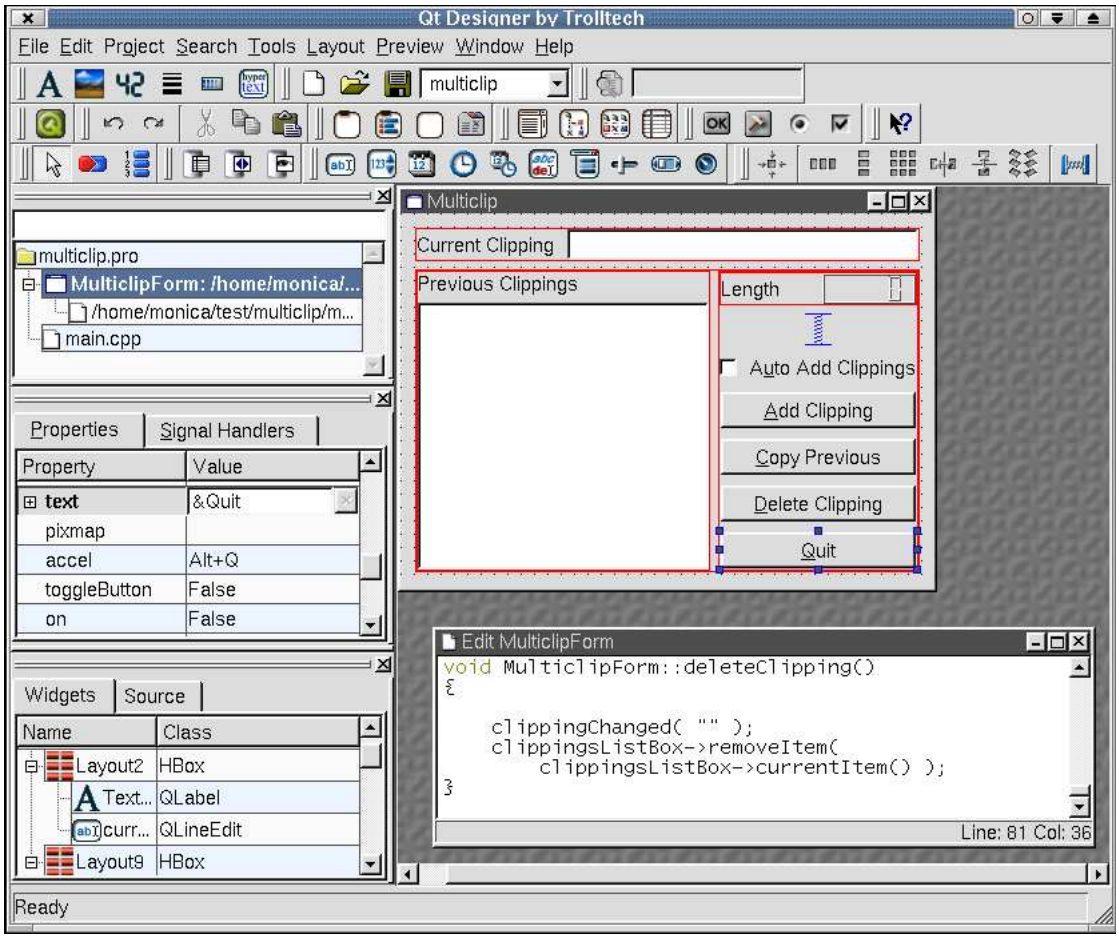

Figure 19. *Qt Designer*

is a simple process. Developers click a toolbar button representing the widget they want, then click on a form to place the widget. The widget's properties can then be changed using the property editor. The precise positions and sizes of the widgets do not matter. Developers select widgets and apply layouts to them. For example, some button widgets could be selected and laid out side-by-side by choosing the 'lay out horizontally' option. This approach makes design very fast, and the finished forms will scale properly to fit whatever window size is available.

Qt Designer eliminates the time-consuming compile, link and run cycle for user interface design. This makes it easy to correct or change designs. *Qt Designer*'s preview options let developers see their forms in any style, including custom styles. *Qt Designer* provides live preview and editing of database data through its tight integration with Qt's database classes.

Developers can create both 'dialog' style applications and 'main window' style applications with menus, toolbars, balloon help, etc. Several form templates are supplied, and developers can create their own templates to ensure consistency across an application or family of applications. *Qt Designer* uses wizards to make creating toolbars, menus and database applications as fast and

easy as possible. Programmers can create their own custom widgets that can easily be integrated with *Qt Designer*.

Form designs are stored in human-readable `.ui` files, and converted into C++ header and source files by the `uic` (User Interface Compiler). The `qmake` build tool automatically includes build rules for `uic` in the Makefiles it generates, so developers do not need to invoke `uic` themselves.

Alternatively, `.ui` files can be loaded at run-time by applications, and converted into fully functional forms. This allows customers to modify the look of an application without recompiling, and can also be used to reduce the size of applications.

7.3. Built-in Dialogs

Qt includes ready-made dialog classes with static convenience functions for the most common tasks. Screenshots of some of Qt's standard dialogs are presented below.

QMessageBox is used to provide the user with information or to present the user with simple choices (e.g. 'Yes' or 'No').

Figure 20. A **QMessageBox**

QProgressDialog displays a progress bar and a 'Cancel' button.

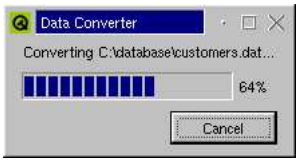

Figure 21. A **QProgressDialog**

QWizard provides a framework for wizard dialogs.

Figure 22. A **QWizard**

Qt also includes **QColorDialog**, **QFileDialog**, **QFontDialog** and **QPrintDialog**. These classes are more suitable for desktop applications and are usually compiled out of Qt/Embedded.

8. Look and Feel

Qt desktop applications adopt the style, or look and feel, of their execution environment, e.g. Windows XP, Mac OS X, Linux. Qt/Embedded applications can use any of these styles, or can use custom styles, statically or as plugins. Developers can customize both the widget style and the window decorations.

8.1. Widget Style

A style is a **QStyle** subclass that implements the look and feel of Qt's widgets. Qt/Embedded programmers are free to use and modify existing styles or implement their own styles using Qt's style engine. The built-in styles available on Qt/Embedded are Windows, Motif, MotifPlus, CDE, Platinum and SGI. The style can be set dynamically on a per-application basis, and even on a per-widget basis.

Figure 23. Comboboxes in the different built-in styles

A family of applications can be given a distinctive look by writing a custom style. Custom styles can be defined by subclassing **QStyle**, **QCommonStyle** or any descendent of **QCommonStyle**. It is easy to make small modifications to existing styles by reimplementing one or two virtual functions from the appropriate base class.

A style can be compiled as a plugin. With plugins, developers can preview a form in their device's custom style in *Qt Designer*. Style plugins also give users the opportunity to change the look of the device without recompiling.

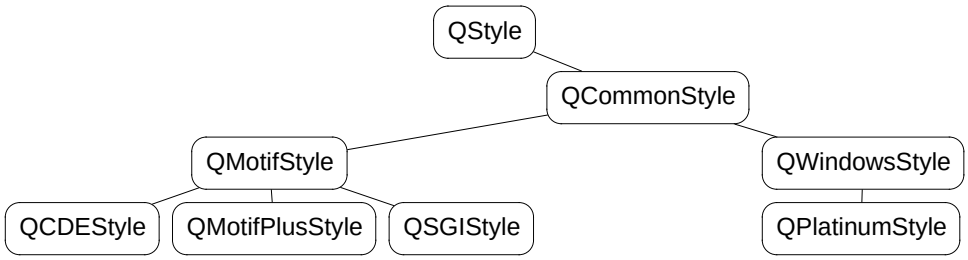

Figure 24. The **QStyle** class hierarchy

Qt's built-in widgets are style-aware and will automatically repaint themselves when the style changes. Custom widgets and dialogs are almost always combinations of built-in widgets and layouts, and are automatically style-aware. On the rare occasions that it is necessary to write a custom widget from scratch, developers can use QStyle to draw primitive user-interface elements rather than drawing raw rectangles directly.

8.2. Window Decorations

Top-level windows are decorated by a title bar and a frame. Qt/Embedded includes these window manager styles: BeOS, Hydro, KDE and Windows.

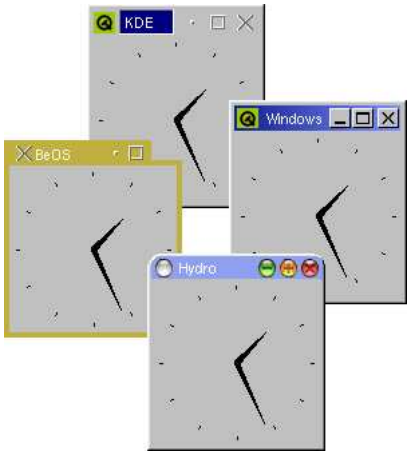

Figure 25. Windows with different window decorations

Decorations can be configured on a per-window basis, if required. Custom styles are created by subclassing **QWSDecoration**, and distributed as plugins. For more control over the window manager's behavior, developers can subclass **QWSManager**.

9. Internationalization

Qt/Embedded fully supports Unicode, the international standard character set. Developers can freely mix Arabic, English, Hebrew, Japanese, Russian, and every other language supported by Unicode, in their applications. Qt/Embedded also includes tools to support application translation to help companies reach international markets.

9.1. Unicode

Qt uses the **QString** class to store Unicode strings. **QString** replaces the crude `const char *`; constructors and operators are provided to handle conversion between **QString** and `const char *`. Programmers can copy **QString**s by value without penalty, since **QString** uses implicit sharing (copy on write) to reduce memory use. Qt also provides **QCString** to efficiently store ASCII strings.

Qt provides a powerful Unicode text rendering engine for all text that is displayed on screen, from the simplest label to the most sophisticated rich-text editor. The engine supports advanced features such as special line breaking behavior, bidirectional writing and diacritical marks. It renders most of the world's writing systems, including Arabic, Chinese, Cyrillic, English, Greek, Hebrew, Japanese, Korean, Latin and Vietnamese. The engine is optimized for the common case: a single line of plain text with an optional accelerator (e.g. **File**).

Conversion to and from different encodings and charsets is handled by **QTextCodec** subclasses. Qt 3.0 supports 37 different encodings, including Big5 and GBK for Chinese, EUC-JP, JIS and

Shift-JIS for Japanese, KOI8-R for Russian and the ISO 8859 series. They can be compiled as part of the library or as plugins, or compiled out using the 'feature' mechanism.

9.2. Translating Applications

Qt provides tools and functions to help developers provide applications in their customers' native languages.

To make a string translatable, simply wrap it in a call to tr() (read 'translate'):

```
saveButton->setText( tr("Save") );
```

tr() attempts to replace a string literal (e.g. "Save") with a translation if one is available; otherwise it uses the original text. For example, English could be used as the source language and Chinese as the translated language, or vice versa. The argument to tr() is converted to Unicode from the application's default encoding.

tr()'s general syntax is

```
Context::tr("source text", "comment")
```

The 'context' is the name of a **QObject** subclass. It is usually omitted, in which case the class containing the tr() call is used as the context. The 'source text' is the text to translate. The 'comment' is optional; along with the context, it provides additional information for human translators.

Translations are stored in **QTranslator** objects, which use memory-mapped `.qm` files (Qt Message files). Each `.qm` file contains the translations for a particular language. The language can be changed at run-time; any dialog created using *Qt Designer* can retranslate itself on the fly with no special provisions.

Qt provides three tools for preparing `.qm` files: `lupdate`, *Qt Linguist* and `lrelease`.

1. `lupdate` extracts all the (context, source text, comment) triples from the source code, including *Qt Designer* `.ui` files, and generates a `.ts` file (Translation Source file). The `.ts` files are human-readable.

2. Translators use *Qt Linguist* to provide translations for the source texts in the `.ts` files.

3. Highly compressed `.qm` files are generated by running `lrelease` on the `.ts` files. The `.qm` files are used on the embedded device.

These steps are repeated as often as necessary during the lifetime of an application. It is perfectly safe to run `lupdate` frequently, as it reuses existing translations and marks translations for obsolete source texts without eliminating them.

9.3. Qt Linguist

Qt Linguist is a Qt application that helps translators translate Qt applications. Translators can edit `.ts` files by hand, or more conveniently using *Qt Linguist*. The `.ts` file's contexts are

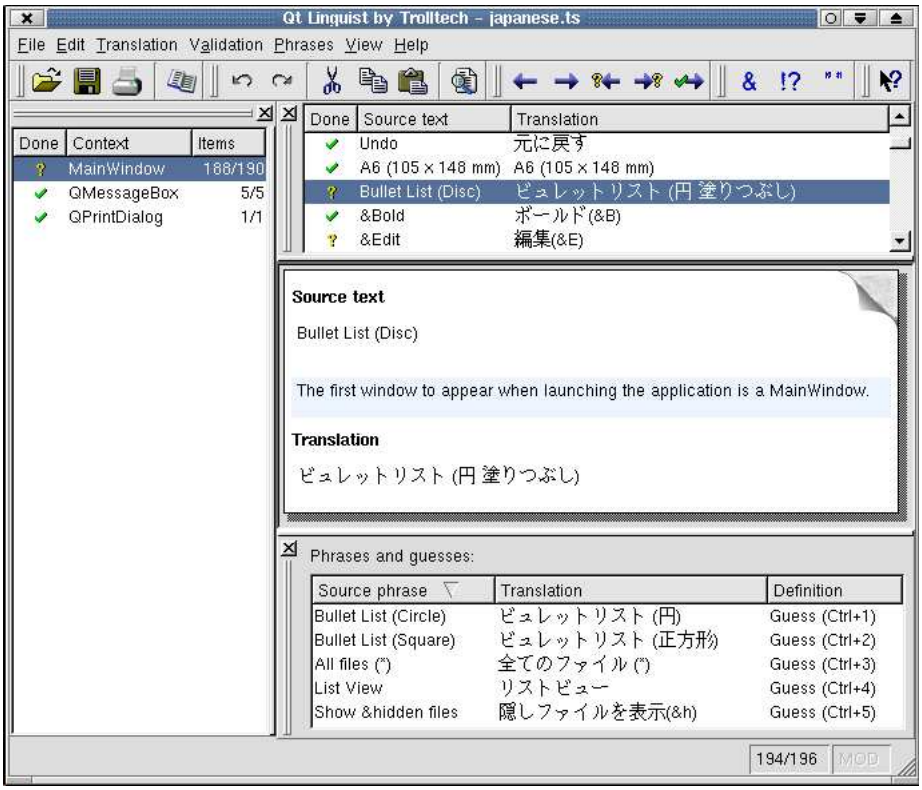

Figure 26. *Qt Linguist*

listed in the left-hand side of the application's window. The list of source texts for the current context is displayed in the top-right area, along with translations. By selecting a source text, the translator can enter a translation, mark it done or unfinished and proceed to the next unfinished translation. Keyboard shortcuts are provided for all the common navigation options: Done & Next, Next Unfinished, etc. The user interface's dockable windows can be reorganized to suit the translators' preferences.

Applications often use the same phrases many times in different source texts. *Qt Linguist* automatically displays intelligent guesses based on previously translated strings and predefined translations at the bottom of the window. Guesses often serve as a good starting point that helps translators translate similar texts consistently. *Qt Linguist* can optionally validate translations to ensure that accelerators and ending punctuation are translated correctly. *Qt Linguist* also detects slight changes in source texts and automatically suggests appropriate translations. These translations are marked as unfinished so that a translator can easily find them and check them.

10. Non-Graphical Classes

Qt/Embedded provides a full range of non-graphical classes that provide data containers (collection classes), input/output, networking, database interaction and threading.

10.1. Collection Classes

Collection classes are used to store groups of items in memory. Qt/Embedded provides two sets of collection classes: pointer-based collections and value-based collections.

The pointer-based collection classes are **QDict<Key,T>**, **QPtrList<T>**, **QPtrQueue<T>**, **QPtrStack<T>**, **QPtrVector<T>** and **QCache<T>**. These classes are often used for storing pointers to **QWidget**s and **QObject**s, and Qt/Embedded's internals make heavy use of them. The pointer-based collection classes can optionally take ownership of the objects they contain and automatically delete them when the collection is destroyed, simplifying memory management.

The value-based collection classes are **QMap<Key,T>**, **QValueList<T>**, **QValueStack<T>**, **QValueVector<T>** and **QStringList**. They have an interface very similar to the STL containers. Qt/Embedded also provides the low-level **QMemArray<T>** class with its subclasses **QBitArray**, **QByteArray** and **QPointArray**. These classes are very efficient for handling basic 'plain old data' types.

To avoid the problem of code bloat associated with templates, Qt/Embedded uses private non-template classes to implement the functionality of template classes. The template classes are only a thin layer that converts special types to generic pointers, and results in very little binary code. Another technique, implicit sharing, is used in the value-based containers to avoid needless duplication of data. These optimizations make Qt's collection classes suitable to embedded development.

10.2. Input/Output

Qt provides **QTextStream** and **QDataStream** to read and write text and binary data in a file, a buffer, a socket or a custom device. **QDataStream** can be used to serialize basic C++ types and many Qt types.

Directories are manipulated using **QDir**. The **QFileInfo** class provides more detailed information about a file, such as its size, permissions, creation time and last modification time.

Transparent access to remote files is provided by **QUrlOperator**. In addition to local file system access, Qt supports the the FTP and HTTP protocols and can be extended to support other protocols. For example, files can be downloaded using FTP like this:

```
QUrlOperator op;
op.copy( QString("ftp://ftp.trolltech.com/qt/INSTALL"),
         QString("file:/tmp") );
```

URLs can easily be parsed and recomposed using **QUrl**.

Image files are usually read by creating a **QImage** with the file name as argument. Printing text and images is handled by **QPainter**. These classes are described in "Widgets" [p. 11].

User settings and other application settings can easily be stored on disk using the **QSettings** class. Settings are stored in text files under hierarchical keys,

e.g. `/Tools/Zoomer/RecentFiles`. Booleans, numbers, Unicode strings and lists of Unicode strings are supported.

Qt's XML module provides a SAX parser and a DOM parser, both of which read well-formed XML and are non-validating. The SAX (Simple API for XML) implementation follows the design of the SAX2 Java implementation, and is especially suitable for simple parsing requirements and for very large files. The DOM (Document Object Model) Level 2 implementation follows the W3C recommendation and includes namespace support.

10.3. Networking

Qt provides an interface for writing TCP/IP clients and servers. The **QSocket** class provides an asynchronous buffered TCP connection. Functions such as QSocket::connectToHost() and QSocket::writeBlock() can be called at any time without freezing the application's user interface. Sockets emit the readyRead() signal when there is data available to read.

The **QSocketDevice** provides an abstraction for the underlying functionality for **QSocket** and **QServerSocket**, and can be used for UDP.

10.4. Database

The Qt SQL module provides a uniform interface for accessing SQL databases. Qt includes native drivers for Oracle, Microsoft SQL Server, Sybase Adaptive Server, PostgreSQL, MySQL and ODBC. Programs can access multiple databases using multiple drivers simultaneously.

Programmers can easily execute any SQL statements. Qt also provides a high-level C++ interface that programmers can use to generate the appropriate SQL statements automatically.

Any Qt widget, including custom widgets, can be made data-aware. Qt also includes some database-specific convenience widgets, to simplify the creation of dialogs and windows that present records as forms or in tables. Data-aware widgets automatically support browsing, updating and deleting records. Most database designs require that new records have a unique key that cannot be guessed by Qt, so insertion usually needs a small amount of code to be written. The programmer can easily force the user to confirm actions, e.g. deletions.

Using the facilities that the Qt SQL module provides, it is straightforward to create database applications that use foreign key lookups, present master-detail relationships, and support drill-down.

Qt's SQL module is fully integrated with *Qt Designer*. *Qt Designer* can preview database forms and tables using live data if desired, allowing developers to browse, delete and update records. *Qt Designer* has templates and wizards to make creating database forms fast and simple.

10.5. Multi-Threading

GUI applications often use multiple threads: one thread to keep the user interface responsive, and one or many other threads to perform time-consuming activities such as reading large files and performing complex calculations. Qt/Embedded can be configured to support multi-threading, and provides four threading classes: **QThread**, **QMutex**, **QSemaphore** and **QWaitCondition**.

11. Qt/Embedded in the Wider World

Qt/Embedded makes Linux a viable platform for embedded GUI applications. It is an implementation of a mature, consistent, object-oriented toolkit that includes many tools to ease and speed development. Qt/Embedded is already used by major companies and is attracting software developers from both the commercial sector and from the open source community.

Qt/Embedded became commercially available for the first time in September 2000. It is a port of the Qt toolkit which has been powering both commercial and open source applications since 1995. Qt/Embedded is already used by enterprises and individuals across the world.

Organizations that wish to make use of a ready-made software environment for specialized devices such as PDAs and WebTVs, can license Qtopia, an environment created by Trolltech that is built with Qt/Embedded. Qtopia is used in the Sharp Zaurus device (shown on the cover-page) and includes a PIM (Personal Information Management) application suite. Qtopia is also available in open source form at http://qpe.sourceforge.net. Qtopia is described in the *Qtopia Whitepaper*.

Insigna Solutions offers a Java Virtual Machine for Qt/Embedded. The Qt API is used to implement the Java AWT, resulting in a look and feel that is consistent with C++ applications.

IBM and OTI (Object Technology International) also provide a Java solution for Qt/Embedded. Their Simple Widget Toolkit is implemented using the Qt API.

Qt has an active and helpful user community who communicate using the `qt-interest` mailing list. See http://qt-interest.trolltech.com to subscribe or to browse the archive.

Qt's extensive documentation is available on-line at http://doc.trolltech.com.

Developers can evaluate Qt/Embedded, with support, for 30 days. See http://www.trolltech.com/products/qt/evaluate.html for details.

For further information, email info@trolltech.com.

A small sample of the applications that have been developed with Qt/Embedded are shown below.

Opera Software has developed a fast Qt/Embedded web-browser that supports HTML 4.0, CSS1, JavaScript 1.3 and cookies.

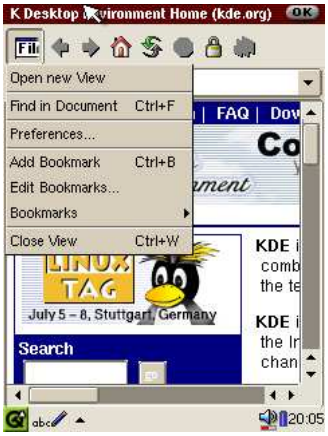

Figure 27. Konqueror/Embedded by the KDE team · Port of NetHack by Warwick Allison

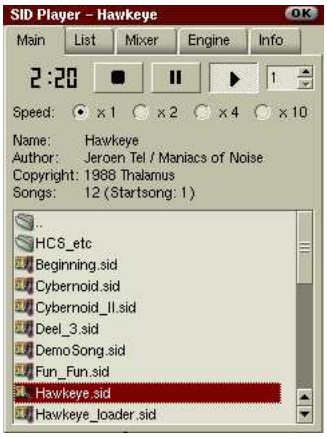

Figure 28. Port of KDE's Sokoban game by Steve Dunham · A SID player by Markus Gritsch

Index