Qt/Embedded

The C++ Embedded GUI Application Developer's Toolkit

Technical Overview

Abstract

This White Paper describes Qt/Embedded, an Application Framework and Windowing System for providing Graphical User Interfaces (GUI) on small computing devices. The system provides the same, rich functionality as the standard Qt Application Programming Interface (API) available on the X11 and Microsoft's Windows platforms; therefore, a Qt API overview is included for the reader's reference. Qt/Embedded is unique in offering this functionality without the memory and performance burden imposed by other platforms.

Contents

Introduction

Early on, embedded systems were used for servers, process control and other non-graphical applications. Manipulation and feedback were usually performed through a connection to a more powerful device such as a UNIX station or a PC. But as embedded hardware improved and applications became more sophisticated, it became increasingly necessary for the user to be able to interact directly with the embedded device. Still, due to limitations in hardware, the early graphical embedded applications were simple; typically only offering text-based in- and output on small black and white displays.

Early Embedded Operating Systems (EOS) were typically proprietary and expensive much like the early UNIX and Windows operating systems. However, as Linux matured, making steady gains in the server and desktop markets, embedded systems developers started to realize that open source and free resources could also be used to create EOSs. Linux quickly became a viable alternative to existing, closed-source embedded platforms.

Because Linux is open source, it is fairly simple for developers to make the changes required for the relatively strict memory and processor demands of embedded systems, and many companies have been able to produce some embedded version of Linux. As such, embedded Linux is steadily gaining momentum and is becoming one of the most popular embedded platforms.

Over time, production technologies and hardware performance also improved. Embedded devices such as mobile phones and Personal Digital Assistants (PDA) greatly benefited from this development. From a hardware point of view, the stage was set for rich, first class GUI applications on embedded devices.

During the fall of 1999, Trolltech realized that embedded Linux and improved embedded hardware had made it viable to put first class GUI solutions on embedded devices. In August 2000, less than a year later, Trolltech presented Qt/Embedded to the public. Being easily adaptable and configurable, major customers, like Ericsson of Sweden, and partners, like Mizi Research and PalmPalm Technologies of Korea, had soon employed Qt/Embedded on a multitude of hardware. On the low end, Mizi had Qt/Embedded running on its SmartPhone with 18 MHz ARM7 processors with only 2 MB of DRAM and 4MB flash. On the higher end, Trolltech made Qt/Embedded run on Compaq's iPAQ with 206 MHz strongARM processors and 32MB of RAM.

This document discusses some of the important features of Qt/Embedded and how they have been designed specifically to offer richness, speed and versatility with a reduced memory footprint. One hundred percent source code compatible with the proven Qt/Windows and Qt/X11 frameworks, Qt/Embedded provides a rich GUI API for creating embedded applications.

Qt is a product of Trolltech. It has been on the market since 1995, and has been used by leading companies such as HP, IBM, Intel, Siemens, Ericsson, and Pixar. Widely used on Linux, Qt is the basis for the popular KDE desktop environment included in every Linux distribution.

A detailed presentation of all the functionality provided by Qt is outside the scope of this document. Further technical information can be found in:

- *The Qt Reference Documentatio*n. Available on-line at www.trolltech.com/ and also available from www.amazon.com.

The most current list of available documentation can be found on-line at www.trolltech.com.

Qt and Qt/Embedded are continuously evolving toolkits. This document presents the main features of Qt/Embedded version 2.2.3. Further information about Qt and Qt/Embedded is available at the Trolltech web site at www.trolltech.com.

System Requirements

Although embedded system hardware is steadily advancing, embedded application developers must consider hardware limitations when designing and implementing solutions. The capacity and performance of memory, processors and screens on embedded devices are still a far cry from what is available even with most budget PCs.

Qt/Embedded has been designed to be memory efficient, to be responsive even on slow processors and to provide crisp and vivid graphics on a multitude of displays.

Memory requirements

One of the most critical components in an embedded device is memory, which heavily influences the production cost of the entire device. As this document is being written, 8 and 16Mb of ROM/RAM is still fairly common, while high-end devices are starting to employ 32Mb and even 64Mb. The Qt/Embedded design team made two major decisions that contribute to the low memory footprint of Qt/Embedded: memory customization and X graphics independence.

Customization of memory usage

Qt/Embedded is modular and customizable allowing the programmer to control the size of the library by leaving out selected features. For example, if an application does not need a particular feature normally offered by Qt/Embedded, the programmer omits that feature during compilation using a text-based configuration file (qconfig.h), thereby controlling the size of the resulting library.

By picking and choosing features, a user can reduce the memory demands of Qt/Embedded to 670 KB. At this level, while it is possible to create simple applications and the event mechanism is functional, most widgets necessary to form a rich GUI are not available. For first class GUI applications, a library size of 1.5 - 3Mb is more common. Due to the rich feature set of Qt/Embedded, applications are generally much smaller than those using other toolkits or X11 directly.

No need for X

The X system relies on an elaborate mechanism for handling displays. As advantageous and popular as this system has become on networked workstations and even some desktops, it makes little sense on most embedded devices. By offering an efficient windowing system and doing all graphics directly in the frame buffer, Qt/Embedded avoids the memory and time overheads of the X system.

Clearly, Qt/Embedded benefits from the same reference-counting and copy-on-write techniques found in all other Qt implementation. This means that many classes are implemented so that copies of the same object share the same data in memory. This saves unnecessary copying of the data, and it reduces the memory demands of the application as a whole. This technique is especially effective when applied to classes that contain large amounts of data, such as pixmaps and images, and for frequently used classes, such as strings. X independence also allows for a variety of display features that have been incorporated into Qt/Embedded. Screen Rotation, Anti-aliased fonts, and alpha-blended pixmaps are the key display advantages.

Screen requirements

Qt/Embedded has been designed to support a multitude of screen sizes, resolutions and bit depths. In fact, Qt/Embedded is independent of hardware and will work with any combination of processors and graphics cards supported by Linux.

Qt/Embedded requires frame buffer support on Linux. Currently, 1, 4, 8, 15/16, 24 and 32 bits per pixel (bpp) depths as well as VGA16 are supported, allowing Qt/Embedded to run on most embedded Linux devices. The Qt/Embedded applications access the video frame buffer directly.

Accelerated drivers for Qt/Embedded can also be used for improved performance. Such drivers access the graphics hardware directly to utilize any available hardware acceleration of graphics operations.

Many small displays do not have hardware acceleration, and although small screens benefit from it less than large ones, hardware acceleration capabilities are crucial to certain groups of graphically intense embedded applications. Trolltech can be contracted to create hardware acceleration drivers.

Processor requirements

All platforms supported by Linux can have Qt/Embedded ported to them easily, and may even work "out-of-the-box" if they use one of the supported frame buffer depths. This includes CPUs based on the ARM/strongARM, x86, Motorola 68000, MIPS, and PowerPC chips. Various groups and vendors are developing other embedded Linux platforms. Qt is highly portable, and Trolltech offers expert porting services to those with special requirements.

Architecture

Qt/Embedded is part of Qt, the cross-platform C++ application framework. Qt is implemented as a class library and provides a rich Application Programmer's Interface (API) for application developers. Qt provides a wide spectrum of generally useful functionality, but the main focus is on the Graphical User Interface (GUI). Thus, for application developers, Qt replaces Motif, MFC, and other GUI toolkits.

Cross-platform development

Qt is cross-platform, in the sense that the Qt class library is implemented for several different operating and window systems. The API is identical for all platforms. This means that an application written with Qt on one platform can be made to run on another by simply recompiling it on the new platform, and linking it with the Qt library for that platform. In fact, the Qt library is binary compatible on all supported Windows variants. Thus, with Qt, software producers can develop and maintain an application for multiple platforms by developing and maintaining a single application source code base. This means it is no longer necessary to learn yet another proprietary, embedded GUI API just to create embedded applications. Another benefit is that Qt/Embedded gives instant portability of existing Qt-based software to embedded Linux systems.

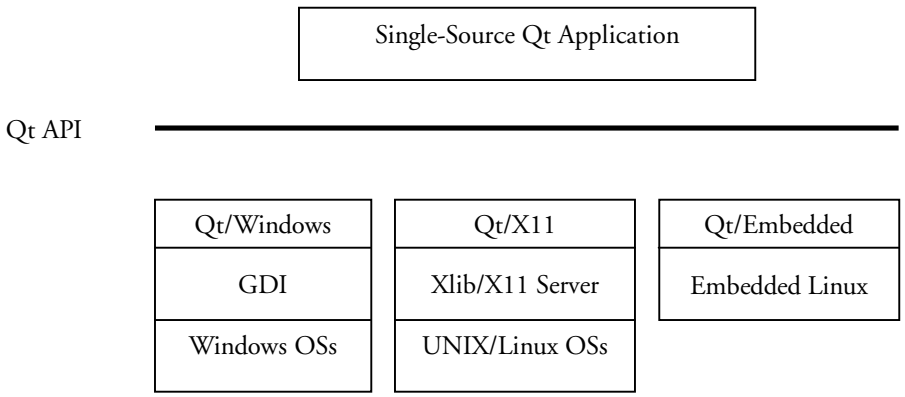
Qt is currently implemented for three main groups of operating systems, as shown in Figure 1:

- UNIX: this includes Linux, HP-UX, Sun Solaris, Digital UNIX, SGI Irix, IBM AIX, SCO UNIX, and several BSD variants. The Qt library is implemented using the X11 libraries, and uses the X Window system.

- Windows: this includes Windows 95, 98, NT and 2000. The Qt library is implemented using the Windows GDI API, and uses the Microsoft Windows window system.

- Qt/Embedded library includes a complete window system and can be easily targeted to any display and input hardware.

Figure 1: Qt Platforms



Implementations for other operating and window systems are being considered.

Look and feel

All visual elements in Qt are implemented with a dynamic look and feel. This means that their visual representation depends on whichever look and feel, or "theme", is currently selected for the application. Since Qt is an emulating toolkit, any Qt-based application can employ any look and feel

style on any platform. The style can even be changed at run-time. Qt provides the following default styles:

- Motif style: emulates the classic Motif look and feel. This is the default style of Qt-based applications running under X11.

- CDE style: a variation of the Motif style which emulates the lighter Motif look and feel that has become popular in the recent years.

- Windows style: emulates the Windows look and feel. This is the default style of Qt-based applications running under Windows.

- SGI: more closely emulates another popular and lighter Motif look and feel.

- MotifPlus: emulates the GTK look and feel.

- Platinum: emulates the GUI found on the MacOs.

In addition, an API is provided for implementing custom styles. This means that for applications that have special demands regarding visual appearance, e.g., a kiosk application, it is fairly straightforward for the programmer to implement a custom look and feel. All visual elements in Qt will then present themselves using this custom look and feel style.

Component programming

In object-oriented software development, it is desirable to structure the application code in independent, reusable components. This principle is known as *component programming*. Qt offers assistance with this task through a special inter-object communication mechanism called *signals and slot*s. This mechanism allows objects to emit anonymous signals that cause slot functions in other objects to be executed. This form of inter-object communication is similar to Motif callbacks and MFC message maps, with some important advantages.

The signals and slots mechanism consists of the following constructs:

- All classes defining either signals or slots must come from the Qt base class QObject.

- A QObject class may define any of its (otherwise normal) member functions to be slots.

- A QObject class may define that it is able to emit certain signals. Each signal has a name and a parameter list, like member functions.

- A signal of one QObject may be connected to a slot of another QObject. If the signals and slots are declared public, a third object can even do this connection.

- A QObject may at any time choose to emit a signal.

The resulting operation of these constructs is that every time a QObject emits a signal, the slot function of the QObject(s) it has been connected to will be executed immediately. Parameter values are passed from the emitting object to the slot functions. Thus, emitting a signal is like a function call, but with the very important difference that the emitting (calling) QObject does not need to know which slot functions (if any) of which QObjects (if any) will be executed. This makes it possible to design application-independent, reusable classes.

Note that all QWidgets are also QObjects with predefined signals and slots ready to be used. The logic of the application is controlled by the way the developer decides to connect signals and slots together. In addition, by sub-classing virtual functions found in Qt, the developer may customize the look and behavior of existing widgets.

A signal may be connected to any number of slot functions, and a slot function may have any number of signals connected to it. Connections can be established and removed at run-time. Any number and type of parameter may be passed with the signal, just as with a normal function call. The signal-slot mechanism provides full parameter type safety. If an application tries to connect a signal to a slot with mismatching parameter types, a warning message is issued and the connection is ignored. Superfluous signal parameters are silently ignored; for example a signal with an integer parameter followed by a string parameter may be connected to slot functions that take either no parameters, or only an integer parameter, or an integer parameter followed by a string parameter.

Qt's signal-slot mechanism replaces the traditional callback mechanisms of older toolkits. An important advantage of the signal-slot mechanism is that it is type-safe; mismatches between the parameter types of the signal and the slot are handled gracefully. Such mismatches in callback functions in other toolkits invariably lead to run-time failures (segmentation faults) and hard application termination.

The typical use of the signal-slot mechanism is best illustrated by an example. For example, assume an application design calls for a dialog box that gets closed when the user clicks its "OK" button. Using Qt, the programmer will implement this using the classes QDialog and QPushButton. The QPushButton class has a signal called `clicked()` that gets emitted when the user operates the button. The QDialog class has a slot function called `accept()` that closes the dialog. Thus, the programmer can achieve the desired functionality by simply connecting the `clicked()` signal of the QPushButton object to the `accept()` slot of the QDialog object. The code looks like this:

```
// create the objects
QDialog *d = new QDialog(...);
QPushButton *b = new QPushButton(...);

// connect the signal to the slot
connect( b, SIGNAL(clicked()), d, SLOT(accept()) );
```

Qt/Embedded Windowing System

A Qt/Embedded windowing system typically consists of a server process and one or more client processes. The server process allocates regions of the display to itself and client processes, generates mouse and keyboard events, and usually includes some form of user-interface for launching client processes.

Client processes communicate with the server to request regions of the display and to receive mouse and keyboard events. In the regions it is allocated, the client directly accesses the display to provide a GUI to the user.

The server and clients use shared memory to communicate the allocated regions, and to maintain a software cursor if required.

Server architecture

The server maintains a set of regions, which change per client request as windows are created, moved, resized, and destroyed. The set of regions resides in shared memory where it is read by the client as drawing operations are executed.

The server connects to system devices such as a mouse and keyboard that produce events that are sent to the appropriate clients. The mouse device can be a stylus or other pointer. The server generates a device-independent mouse event and sends it either to the client that created the window containing the mouse point, or the client, which has 'grabbed' the mouse. The server updates the hardware mouse cursor, and the client and server maintain software cursor cooperatively. Stylus-operated devices will generally not have a mouse cursor at all, but the gestures of the stylus are converted to device-independent mouse events for standard processing by clients.

Keyboard events are maintained by the server and are device-independent events that use Unicode and fixed keycodes. On some embedded devices there is no physical keyboard and so the server provides handwriting-recognition or displays a virtual keyboard with which the user interacts using the stylus.

User interface elements such as the virtual keyboard are possible because the server process is also a master client. The server therefore has access to all the GUI functionality available to clients. Usually the server will only use this functionality for windows related to keyboard input and perhaps some basic mechanism for launching client applications, but in very small embedded systems, the server may be the only process. Any client application can be the master by setting a flag as it starts, but there can of course be only one server for each display.

Client architecture
The API to which clients are written is the same standard Qt API found on Qt/X11 and Qt/Windows. When a Qt/Embedded client uses the Qt API to draw a line, the Qt/Embedded library accesses the display directly, whereas when a Qt/X11 client draws a line, the Qt/X11 library sends a message to the X server, which does the actual drawing. Similarly, when a Qt/Windows application draws a line, the Qt/Windows library calls the Win32 operating system to have the line drawn. In all instances, the application code is the same, calling up the same standard Qt API line drawing function.

The Qt/Embedded client library connects to the server process when the client starts and communicates with the server for all operations which have global consequences such as changing the display region covered by windows (which changes what regions are available to other clients), manipulating the global clipboard and drag-and-drop properties, and receiving mouse and keyboard events from the user.

It should be noted that for drawing operations, the client does not need to interact with the server because it has an allocated region of the display to which it can freely write to display user-interface controls, video clips, and other graphics. The client drawing functions are built on an architecture that easily supports hardware accelerated operations, and has been test-targeted to Mach64 and Voodoo3 devices. The base drawing functions draw directly to the Linux frame buffer.

The Qt/Embedded client library handles all drawing operations, including text display and font handling. It also handles window decorations (title bars, etc), which can be customized on a per-window basis.

The client library has built-in support for Windows .FON files; TrueType, Type1, and BDF fonts; and a memory-efficient bitmap font format for storing pre-rendered versions of such fonts. TrueType and Type1 fonts can be displayed using anti-aliasing whereby text readability is improved by smoothing.

Bitmap font files are formatted in such a way that they can be memory-mapped, unlike on X11, where the server reads in the entire font. Similarly, TrueType fonts are rendered as required for each single character, whereas on X11, many extra characters are rendered. For example, on X11, the character set is divided into chunks of at least 16, so a heading such as "Welcome to Alcatel" renders almost all 52 upper and lower case characters, rather than nine.

Some functionality in the Qt API is implemented in a superior way on Qt/Embedded, such as the anti-aliased text already mentioned, and full-color mouse cursors. These improvements are possible because the client library has direct access to the display, unlike with X11 where the only drawing operations that can be implemented efficiently are those defined in the X11 standard, which changes very slowly. End-user applications for which portability to Qt/X11 and Qt/Windows is not required can also leverage the direct access to implement very fast application-specific drawing.

Internationalization

Qt is designed to allow applications to use any language and character set. It is easy to switch languages, even at run-time. The internationalization feature allows the application developer to

present the user with a choice of languages. This is another way Qt empowers not only the application developer, but also the end users.

Unicode
Qt allows applications to use international (i.e. non-ASCII) character sets. For text operations, Qt provides the QString class, which contains a text string in the 16 bit Unicode standard encoding. Qt is 16 bit clean throughout. The Qt kernel uses the QString class for all internal text operations, and it is used in all API functions that take or return text parameters. This includes all text labels of widgets such as labels on push buttons, menu items, content of line edits, etc.

QString is highly optimized, and in tests of moving real-world applications from 8 bit to 16 bit strings, no significant performance penalty has been observed. This is not unexpected, since text manipulation is not among the most demanding tasks performed by typical GUI applications.

Qt supports keyboard input and screen output of Unicode text, as provided by the underlying window system. Screen output requires the appropriate font(s) to be installed. These fonts need not be Unicode encoded as Qt provides codecs between Unicode and many of the common font encodings. Custom codecs can also be added.
All application text I/O (e.g., to/from files) may be passed through a text codec, which translates between the preferred local format and the Unicode standard format used internally. Codecs for a number of commonly used locales are provided, as well as an API for implementation of custom codecs.

Localization
Qt provides support for creating localized applications, such as applications that can choose at run-time which language to display for all the user-visible texts. The choice may be made automatically based on the user's locale setting, or explicitly by the application. This can be achieved by presenting the user with a language selection dialog on start-up. Building a Qt application that is prepared for localization is straightforward: The programmer simply passes all user-visible texts through Qt's `tr()` (translate) function before passing them to Qt for display. For example, the non-localized application code to make a push button display the text label "proceed" would be:

```
myPushButton->setText( "Proceed" );
```

while the localization-prepared version would be:

```
myPushButton->setText( tr( "Proceed" ) );
```

The translate function will do a lookup in the current translation table, and return the text string (translation) corresponding to the argument.

A handy aspect of the translate function is that if no translation table is installed, it will simply return its argument. This means that a localization-prepared application will run just fine even if the translation tables are not present. The behavior will be the same as if the application were not prepared for localization. This is practical during application development when translation tables often have not yet been produced, or for releasing the first version(s) of an application which is planned to be localized in later versions.

Qt provides tools which assist the application developer to build and maintain the translation tables. One tool, `findtr()`, searches the application source code for strings that need translation, and produces a formatted text file with empty areas for the application translator to simply fill in the required translations. Another tool, `msg2qm()`, converts these text files to the binary, hashed translation table files that are used by Qt for lookup at run-time. A third tool, `mergetr()`, assists in merging existing translation files when the application has been extended or modified so that new strings that need translation have been added.

An example showing a translation to Japanese illustrates how easy Qt makes offering applications to customers using any language. After having written the source using the `tr()` function for all

visible texts, the `findtr()` tool is used to generate the original translation files. The translation file template for mywidget.cpp would be mywidget.po and part of it might look like this:

```
#: mywidget.cpp:28
msgid "MyWidget::E&xit"
msgstr ""
#: mywidget.cpp:41
msgid "MyWidget::Perspective"
msgstr ""
```

mywidget.po is copied to mywidget_jp.po and the Japanese signs inserted as appropriate, making it look like this:

```
#: mywidget.cpp:28
msgid "MyWidget::E&xit"
msgstr "エグジット"
#: mywidget.cpp:41
msgid "MyWidget::Perspective"
msgstr "遠近法"
```

The only step needed now is to use the `msg2qm()` tool to generate the binary translation file, mywidget_jp.qm, used run-time by Qt. The very same approach can naturally be repeated for any language.

With the appropriate translation files and font sets installed, Qt allows the application's language to be changed run-time. The developer simply creates a QTranslator object, loads the appropriate translation file and applies the translator to the current QApplication. In code, it looks like this:

```
QTranslator *translator = new QTranslator( 0 );
QString lang("mywidget_jp.qm");
translator->load( lang, "." );
qApp->installTranslator( translator );
```

All visible text will now be displayed using the Japanese character set, to the delight of the application's Japanese users. Localization is another example of a powerful, but often difficult to implement concept made simple by Qt.

Layout management

When implementing the visual appearance of a GUI, one of the main tasks is to decide the positions and sizes of the child widgets within their parent's area. Although it is possible to hard-code static coordinate values for all widgets, this approach is usually not satisfactory for anything but the simplest applications, for the following reasons:

- Most applications will want to allow the user to resize the application window while still keeping the window contents. This might produce unwanted results if the coordinates are static.

- For localized applications, or applications where the contents of otherwise static widgets can change dynamically at run time, suitable coordinate values cannot be known in advance.

- Similarly, applications that want to honor the user's preferred font setting cannot know in advance how much space will be required to display its widgets using that font.

- It is a time-consuming and tedious task for the programmer to tune the widgets' positions and sizes so that they align and give the desired aesthetic impression. Maintenance is also demanding, since the whole layout must be manually re-implemented whenever widgets are added or removed.

To overcome these problems, Qt provides a mechanism for automatic widget layout management. By providing an API, a widget may create a layout manager object that will take care of assigning positions and sizes to the child widgets. The layout manager does this by dividing the widgets' available area into virtual cells (as many as there are child widgets), and placing one child widget in each cell. When the widget gets resized, or a child widget's size requirements change, the layout manager will automatically recalculate the layout, and move and resize all the child widgets to fit.

Qt provides two basic layout manager classes:

- QBoxLayout divides the available space into a stack of cells (horizontal or vertical).
- QGridLayout divides the available space into an n x m grid of cells.

In addition, custom layout manager classes may also be added. Note that instead of a child widget, a cell may contain another layout manager object, which in turn manages other child widgets. Thus, by building a nested structure of layout managers, automatic layout of very complex user interfaces can be readily achieved.

Each widget class specifies its own layout requirements:

- A widget may specify a preferred size for itself.
- A widget may specify a minimum size it needs to display itself in a satisfactory manner. For example, a push button will ask to not be made so small that it cannot paint its label and the surrounding button frame.
- A widget may specify that it should not be stretched out more than its preferred size, or that it should be stretched in only one direction. For example, a vertical scroll bar will specify that it can be stretched vertically, but not horizontally, since the latter would ruin its visual appearance.

All of Qt's standard widgets provide sensible, run-time default values calculated using the widgets' current contents and state for all the above-mentioned constraints. If the contents of a widget change while the program is running, the layout will be automatically recalculated to fit the new size of the widget.

The layout algorithm may be tuned as follows:

- A stretch factor can be assigned to each cell to determine what ratio of the available, superfluous space the layout manager will assign to it.
- The widths of the blank borders around and between the cells can be changed. Extra blank space (stretching or non-stretching) may be added.
- The alignment (left/right/ top/bottom/center) of the child widget within the cell may be specified.
- The maximum and/or minimum size of the child widget may be set explicitly.

By applying the above adjustments to the standard layout algorithm, virtually any layout behavior can be obtained.

Customizing widgets

A central design feature of Qt's widget system is extensibility. This is important since experience shows that a fixed set of static widget classes cannot cover all the requirements of a real-world application. GUI toolkit designers may attempt to foresee the various demands that applications may

have, and try to provide the necessary functionality in the widget classes. Indeed, Qt's standard widget classes are designed like this. But that can never be a satisfactory replacement for enabling the application developer to easily customize the widget classes or design widget classes from scratch.

Qt is designed to make it very easy to create custom widget classes. The application programmer simply makes a new C++ class that inherits QWidget (directly or indirectly). There are no resource files to be edited, or mandatory methods to implement (except for the constructor, as the C++ syntax demands). Depending on what the widget shall do, the programmer can choose to implement any of QWidget's virtual methods, in order to receive events or customize look and feel.

Custom widgets have several powerful uses, including presentation of application data, tuning of standard controls, and creation of custom controls.

Qt Virtual Framebuffer

An important utility available with Qt/Embedded is the Qt Virtual Framebuffer (QVFb). QVFb is currently available on the X11 platform and makes it possible to run, test and debug embedded applications right on the desktop. QVFb simulates the Linux Framebuffer and the user can specify height, width and bit depths. In a marked where time to marked is crucial, QVFb is giving Qt developers and important advantage.

Qt Virtual Network Computer Server

For some embedded applications, the ability to display the GUI on arbitrary, external displays is required. For this purpose, Trolltech added a Virtual Network Computer (VNC) server that can be added to the toolkit at compile time. Since VNC clients are freely available for virtually any desktop and operating system, this capability ads invaluable flexibility and makes it possible to develop whole new types of embedded applications.

Qt Palmtop Environment

The Qt Palmtop Environment (QPE) available from Trolltech is the first Personal Information Management (PIM) package for embedded Linux released under the GPL. With the release of Qt/Embedded and the QPE under the GPL, programmers can port their favorite open source Qt-based applications over to embedded devices, such as PDAs.

QPE uses Qt/Embedded, so there is no need for an X11 server, extra client libraries, separate window manager, or layer-upon-layer of toolkits. Although memory requirements vary somewhat on different systems, the entire Qt Palmtop Environment requires about 2.5 MB of memory on x86 based systems.

QPE is a complete Windowing System with Window Manager and an Application Launcher. In the latest release, QPE 1.1, a simplified build system and improved development documentation allow any developer to add their own applications to the QPE.

Several input methods are included making it easier for the end user to enter information:

- QWERTY layout virtual keyboard
- Opti layout virtual keyboard
- Pickboard – a popular mobile phone text input method
- Handwriting – a trainable handwriting input mechanism
- Unicode – displays all Unicode characters

QPE comes with a set of Personal Information Management applications:

- Address Book
- Date Book
- Todo List

a multitude of useful applications and utilities:

- Text Editor
- Spreadsheet
- File Browser
- HTML Help Browser
- MPEG Player
- Calculator
- Clock
- Memory and Load Meter
- Screen Rotator
- Terminal

and games:

- Mine Hunt
- Word Game
- Tetr*x
- Solitaire
- Tux

Trolltech currently has QPE running on Casio's Cassiopeia and Compaq's iPAQ as well as on different development hardware. QPE is also available as a bootable floppy disk for PC. All are available under the GPL license. To download the source or binaries for any of these platforms, please, refer to http://www.trolltech.com/products/qt/embedded/qpe.html.

Visual Development

Although the Layout Management offered by the Qt API does aid the user in overcoming most of the problems discussed in the previous chapter, it is still cumbersome and time consuming to develop a large number of dialogs and widgets working only with the C++ API. Developers need better tools to speed up the development and maintenance of user interfaces.

IDE and RAD

Integrated Development Environments (IDE) and Rapid Application Development (RAD) tools such as Borland Delphi and Microsoft Visual Basic try to fulfill this need. But each new IDE or RAD puts new burdens onto the shoulders of application developers. There is a new, and often very complex user interface to become familiar with. Time and effort must be invested in understanding the logic and operation of the application in order to set up the environment and preferences properly.

What many developers dislike is that most of these applications try to be too clever and lock their users into their own technology and products. This might make sense from a strict business point of view, but it is often not the optimal solution for the developer. For instance, in Microsoft Studio, programmers are expected to use the accompanying text editor as well as the compiler. Traditionally, IDE and RAD tools tend to offer poor support for integrating with the developer's favorite text editor or compiler.

Qt designer

With the release of Qt 2.2, Trolltech introduced the Qt Designer; an application designed specifically to speed up the mundane and laborious tasks when constructing and maintaining graphical user interfaces. The following is a list of what Qt Designer does:

- offers the user a point and click, Drag and Drop, WYSIWYG interface to user interface creation and layout.
- allows easy access to widget properties.
- provides easy access to all the standard widgets as well as the Layout Management mechanisms available in Qt.
- employs a vendor neutral, fully documented XML-format for persistent storage.
- facilitates editing and re-generation of forms by employing a two-layered C++ class hierarchy. Since the application developer only works with sub-classes, re-generating the forms will not interfere with previous work.
- allows different groups to work concurrently on design and implementation as is often a requirement in organizations with dedicated design departments.

Qt Designer does not:

- generate C++ code. A separate tool, the User Interface Compiler, is offered which generates C++ files from the User Interface files Qt Designer uses for persistent storage.
- force the developer into using a certain text editor, compiler or debugger. It gracefully integrates with whatever tools the developer prefers working with.

From this it is clear that Qt Designer is not just another IDE. It is a specialized tool capable of integrating with the application developer's preferred development tools. This is done in a visual manner, greatly speeding up the specific task of creating and maintaining graphical user interfaces. To the developer, Qt Designer is a simple, yet very powerful tool.

Graphics

Qt has a mature 2D graphics API. The QPainter class offers the basic graphics functionality supported by Qt. It provides a high-level drawing engine with commands for drawing lines, polygons, ellipses, splines, images, pixmaps, texts, etc.

Device independent graphics

Qt supports graphics drawing to screen (i.e. widgets), printers, pixmaps and pictures (known as a meta-file in Windows terminology). Qt hides the intrinsic differences between these devices from the application programmer. In Qt, they are all paint devices.

A QPainter operates on a paint device, and the application using the QPainter need not be concerned about whether this QPainter is currently drawing on a widget or on a printer. The drawing API is totally device independent. This is practical for many tasks. For instance, applications may use the same drawing routine for screen output as for print output. This can be achieved by simply making the drawing routine take the QPainter as a parameter, and then passing one QPainter operating on a widget for screen drawing, and one QPainter operating on a printer for printer drawing.

Special paint devices

In addition to widgets and printers, QPainter supports drawing to two special devices:

QPixmap
A pixmap is an off-screen memory frame area, i.e. a two-dimensional array of pixel values. If an application is going to display complex static graphics on screen, it makes sense to draw the graphics into a pixmap, and then just draw the pixmap to screen later. This technique, known as double-buffering, is more efficient since the complex drawing need only be performed once. It can also be used to eliminate screen flicker. Qt provides fast `bitblt()` (bit block transfer) operations for moving pixels between widgets and pixmaps.

Pixmaps are also handy for storing graphics to file for later retrieval, or other transfer of image data.

QPicture
A picture is a stored sequence of drawing operations. Pictures are very handy for storing graphics for re-display at a different magnification level. Zooming in on a pixmap will only magnify the individual pixels, but zooming in on a picture will recreate the drawing as if it had been drawn at that scale originally.

The 2D graphics API

QPainter is implemented using the drawing operations of the underlying window system (e.g., Xlib on UNIX/X11 and Windows GDI on Windows). Features lacking in the underlying system, such as drawing transformations on Windows 95/98 and X11, are implemented within Qt itself. Qt/Embedded provides the drawing operations through the client because there is no underlying window system.

Color handling
Qt provides a separate class for specifying color for drawing operations. Colors can be specified as RGB or HSV values, or as a name from the web standard, such as “steelblue”, “green4”, etc. On systems with limited color ranges like 8 bit displays, Qt automatically handles the allocation of colors in the system palette so Qt-based programs need not do anything special to be prepared for running on such systems.

The drawing style
For specifying the desired graphics attributes of lines, polygon outlines, etc., Qt provides the QPen class. QPen lets the application developer control the line color, width, and stipple pattern.

For the fill style of polygons, ellipses etc., Qt provides the QBrush class. With this, the fill color and pattern can be specified. QBrush provides a set of predefined patterns, but in addition, custom fill pattern (specified as a pixmap) can also be used.

Transformations
QPainter provides full support for transformations, such as scaling and rotating. A QPainter's world transformation specifies how the world coordinates (e.g., the parameter values given to the `drawRect()` method) will be transformed into logical coordinates. The view transformation specifies how these logical coordinates in turn will be transformed into the physical coordinates of the paint device.

For the world transform, Qt supports a general transformation matrix. That is, all forms of coordinate translation, scaling, rotation and shearing can be performed. A separate transformation matrix class is provided, but QPainter also has convenience functions for specifying the most common transformations directly.

For the view transform, Qt allows setting the origin and extent of the drawing window and the drawing viewport. The drawing viewport determines the logical coordinate system, and specifying this to, for example, 1000 x 1000, gives the application programmer a 1000 x 1000 drawing area independent of the size of the underlying physical device. The drawing window, on the other hand, determines the rectangle of the physical device that the logical coordinates will be mapped down into.

All drawing operations provided by QPainter may also be performed with world and/or view transformation applied, including text and pixmap drawing.

Clipping
QPainter allows clipping to a rectangle or a more general region composed of a set of rectangles, polygons, ellipses, and bitmaps. The composition can be made as unions, intersections and/or subtractions.

Text drawing and fonts
QPainter provides two text drawing methods: a simple function for drawing a given text at a given x, y coordinate, and a more complex function allowing the specification of :

- A rectangle Qt should fit the text into;
- How to align the text within the rectangle (top, bottom, flush left, center, flush right); and
- Whether Qt should break the text into lines to fit the width of the rectangle.

A separate class, QFont, is provided for specifying the font. All the fonts installed in the underlying window system may be used for text drawing in Qt. A font may be selected by specifying any or all of its name, size, weight (bold), slant (italic), and character set. Qt will provide the closest matching available font. Font sizes can be given as logical (dpi) or pixel sizes. A number of international character sets are supported, including ISO_8859-1 to ISO_8859-15 (Latin1 to Latin9), KOI8R, and Japanese and various other Asian character sets.

Image handling

Qt supports input, output, and manipulation of images in several formats, including PNG, BMP (Windows bitmap), XBM (X11 bitmap), XPM (X11 pixmap), PNM (P1-P6), and optionally GIF (note that including GIF support may require patent licensing from Unisys). All these image formats are supported on all platforms (e.g., BMP on both Windows and UNIX). Image formats are auto-detected on reading.

The Qt ImageIO Extension library adds support for the JPEG format, and also allows application programmers to add support for custom formats. The image formats added with the ImageIO Extension become fully integrated with Qt's image handling system, just like the internally supported formats.

Once read into an application, an image is stored in a QImage object. This class provides an API that allows manipulating the image data in a hardware-independent manner. This means applications

using QImage for image manipulation can easily be designed to function independently of the screen depth and byte-ordering (endianess) properties of the hardware they run on. QImage also provides direct access to the image data (memory block), for speed-critical operations.

QImage supports images of 32-, 8-, or 1-bit depth. Images with other depths are automatically converted to the next higher supported depth. For 8- or 1-bit deep images, a color palette is provided, which also may be manipulated. Depth conversion methods are provided, including optional dithering when converting to a lower depth.

For each pixel in a 32-bit deep QImage, an 8-bit value is stored for each of the red, green, blue and alpha components. The optional alpha component may be used for custom image operations relating to image transparency, blending, etc.

Qt also supports reading of animation image formats, with asynchronous (e.g. frame-by-frame) reading for interleaved reading and display. The QMovie class provides easy handling of animations.

Canvas

The QCanvas class contains any number of QCanvasItems and has any number of CanvasView widgets observing some part of the canvas. QCanvasItems represent drawing primitives such as pixmaps, rectangles, lines and texts, and can be moved, hidden, and tested for collision with other items. They have selected, enabled, and active state flags which subclasses may use to adjust appearance or behavior.

A canvas containing many items is different than a widget containing many sub-widgets:

- Items are drawn much faster than widgets, especially when non-rectangular.
- Items use less memory than widgets.
- Efficient item-to-item hit tests ("collision detection") can be performed with items in a canvas.
- Finding items in an area is efficient.
- There can be multiple views of a canvas

Widgets of course offer richer functionality with respect to hierarchies, events, and layout.

Each canvas has a solid background and foreground, and all the items are drawn in between. Qt provides a rich API, offering control of the appearance of each layer. QCanvas also simplifies animation by enabling the user to control the speed and direction of movement of each QCanvasItem. Finally, QCanvas offers collision testing. Using the API, it can easily be determined which, if any, items collide with a point, a rectangle or with other items.

3D graphics

Qt does not itself offer 3D graphics functionality, but integration with 3rd party 3D libraries is provided. Many Qt users employ these extensions to be able to offer exciting 3D solutions to their clients and customers.

It should be noted that these integration packages do not depend on special support within Qt itself; the ordinary Qt API provides the necessary general low-level access functions. Thus, it is possible for application programmers to build custom integration packages to other libraries.

OpenGL

The Qt OpenGL Extension library provides integration with OpenGL. It allows the application developer to build data display widgets where the contents are drawn using the native OpenGL library

instead of Qt's 2D graphics code. The Qt OpenGL Extension also provides a platform-independent C++ wrapper API around the platform-specific C APIs GLX and WGL. More information on OpenGL can be found at www.opengl.org.

HOOPS
HOOPS is a high-level, cross-platform, object oriented graphics subsystem that simplifies the design, development and maintenance of high-performance, interactive 2D and 3D graphics applications. It is a product of Tech Soft America, which offers a HOOPS-Qt integration package. More information on HOOPS 3D can be found at www.hoops3d.com.

Tool Classes

Qt is more than a GUI toolkit; it is an application framework. In addition to the GUI functionality, Qt provides application developers with a comprehensive set of generally useful tool classes.

Some of Qt's tool classes provide similar functionality to the C++ standard library and the STL. However, Qt's tool classes are preferable for most Qt-based applications, for the following reasons:

- Portability: the Qt classes are portable to a wide range of platforms and compilers. Many of these platforms lack a functional and standard-conformant STL implementation. By using the Qt classes, application programmers are relieved from relating to such portability issues.

- Cross-platform data exchange: Qt's classes for data I/O provide platform- and architecture-independence, so that even binary data can be successfully exchanged between one platform and another. This is not the case with the standard I/O.

- Internationalization: Qt's classes for text handling and I/O are Unicode-based and thus fully prepared for internationalization. Again, this is not the case with the standard classes.

However, application developers may freely choose to use the standard library and/or the STL instead of, or in combination with, the Qt tool classes; they may coexist in the same application without problems, and data conversion between the tool sets is straightforward.

Operating system services

The task of making an application truly portable involves more than giving it a cross-platform GUI. Real-world applications will always need to access various operating system services, which typically have different, incompatible APIs on the different operating systems.

Qt provides OS-independent encapsulations of the most commonly used OS services. Thus, by using the Qt API instead of the native OS API, Qt-based applications can be immediately re-compiled and successfully executed on new platforms. This relieves the programmer from maintaining large amounts of different, conditionally compiled (#ifdef'ed) code for the various platforms. It has the added advantage of providing the programmer with a clean, object-oriented C++ API to the OS services, instead of the native C API.

Files and directories
Qt provides an API that allows Qt-based applications to query and manipulate the files and directories of the local file system in an OS-independent manner. Files and directories may be created, deleted and renamed, their access rights may be queried and modified, etc. The programmer is relieved from having to relate to such platform-specific details such as whether to use a forward or backward slash for the directory separator character.

Times and dates
Classes for querying the system date and time are provided. Dates and times may be operated on with millisecond resolution. The time span between two different dates/times can be computed. Conversion to and from various date formats (Gregorian, Julian, seconds since the 1.1.1970 epoch, etc.) is provided.

Low-level I/O
Qt provides an API for OS-independent file I/O. The file I/O class is a specialization of Qt's general I/O device encapsulation class. It provides low-level I/O, i.e., reading and writing of raw blocks of bytes. Another specialization class provides I/O to a memory area. Custom encapsulations of other I/O devices may be added in the same way. Thus, an application may use the same code for doing I/O to files, memory buffers, and custom devices.

High-level I/O
Qt provides OS-independent, high-level, stream-based I/O. Both binary and text streams are provided. The streams use Qt's low-level I/O system, so they may be read from and written to files, memory buffers, and custom devices.

All the fundamental types (various precisions of integers and floating point values) and text strings may be read and written. The stream format is independent of the OS and the CPU byte-ordering (endianess), so the streams written on one OS architecture may be read on any other.

Most of Qt's non-widget classes provide functionality for serializing their data to and from a binary stream, so they can be stored efficiently for later retrieval.

The text stream can be set to use a specific encoding / codec in order to read or write text in a format compatible with non-Qt applications.

Text classes

Qt provides two string classes. QString is a powerful string class for all kinds of text operations. It operates with 16 bit Unicode characters. But for non-international applications, Qt provides the QCString class. Seamless integration with the traditional C "char*" string is provided through automatic conversions.

QString
QString uses sharing, meaning that copies of a QString object will share the same string data in memory. The application programmer need not be concerned about the data sharing; if the application modifies the contents of one of the copied objects, QString automatically makes a deep copy of the string data, so that the contents of the other copies remain unchanged. Sharing saves much memory and unnecessary copying.

QString provides all the usual string class functionality, like searching, replacing, conversion to and from integer / floating-point values and various textual representations, comparison operators, truncation, insertion, etc. It automatically allocates enough memory space for the contents, so the programmer is relieved from managing this.

For advanced text searching, Qt provides a regular expression class. Strings can be matched against regular expressions, and the position and length of the match is returned, so it is straightforward to implement, for example, regular expression search and replace functionality.

QCString
For easy manipulation of non-internationalized text and classic C strings, in cases where the conversion to and from QString's 16 bit representation could become a performance issue, Qt includes the QCString class which provides most of the same functionality as QString.

Collection Classes

Qt includes a full set of generic, template-based collection classes. These allow the programmer to easily make things such as a stack class that operates on any Qt or programmer-defined class. The major collection classes are:

- Array: provides an ordered list of objects, with constant-time indexed access.

- Dictionary: stores a key value along with each object, and provides fast (hashed) lookup based on the key values.

- Cache: a dictionary with a programmer-defined limit to the total number and/or cost of stored objects. When the limit is exceeded, the least recently accessed objects are discarded from the collection.

- Map: a sorted list stored in a tree structure for efficient searching.

- List: provides a double-linked list. For convenience, specialized List classes are provided for commonly used collection types, such as Sorted List and String List.

- Queue: a first-in, first-out list.

- Stack: a last-in, first-out list.

For all collection classes, corresponding Iterator classes are provided. The Iterators allow traversal of the entire collection independently of the collection's normal access method.

Network Classes

Qt includes a set of classes offering cross-platform support for non-blocking socket-based programming. The most important classes are briefly discussed below.

QSocket
This class can be employed to establish a non-blocking socket for TCP, UDP (a UNIX-domain socket) or any other protocol family your operating system supports. It supports both binary and ASCII mode and offers a rich API for communicating state and accessing its content.

QSocketNotifier
Once a QSocket is created, a QSocketNotifier can be created to monitor the socket. The `activated()` signal is then connected to the slot to be called when a socket event occurs. There are three types of socket notifiers and the type specifies when the `activated()` signal is emitted:

- QSocketNotifier::Read: there is data to be read (socket read event).

- QSocketNotifier::Write: data can be written (socket write event).

- QSocketNofifier::Exception: an exception has occurred (socket exception event).

For example, if both reads and writes must be monitored for the same socket, two socket notifiers must be created. Socket action is detected in the QApplication::exec() main event of Qt.

QServerSocket
This class is a convenience class for accepting incoming TCP connections. The port can be specified, or QSocketServer can pick one, listening on just one address or on all the addresses of a machine.

QDns
Both Windows and UNIX provide synchronous DNS lookups; Windows provides some asynchronous support too, but neither OS provides asynchronous support for anything other than hostname-to-address mapping.

QDns rectifies that, by providing asynchronous caching lookups for the record types that modern GUI applications are expected to need in the near future. The aim of QDns is to provide a correct and small API to the DNS.

Threading

Qt offers cross-platform threading support.

QThread
A QThread represents a separate thread of control within the program It shares all data with other threads within the process but executes independently in the way that a separate program does on a multitasking operating system. Each thread starts executing in QThread::run() and dies whenever that function returns.

QMutex
The purpose of a QMutex is to protect an object, data structure or section of code so that only one thread can access it at a time. In Java terms, this is similar to the *synchronized* keyword.

Trolltech Embedded Partnerships

Trolltech is continually gaining new partners interested in applying Qt/Embedded to a variety of embedded devices.

MIZI Research, Inc., a leading Linux solution provider specializing in mobile Internet appliances such as SmartPhoneT, PDAT, and WebPhoneT technology. MIZI is including Qt/Embedded with their LINUETTE Linux for embedded devices.

PalmPalm Technology Inc., specializing in Embedded Linux, and **Opera Software**, a privately held company headquartered in Oslo, Norway, have formed a strategic alliance with Trolltech for the Asian wireless Linux market. The three companies are jointly developing the "Linux Total Solution for Wireless Internet Appliance" for hardware manufacturers in the wireless Internet space. Linux Total Solution for Wireless Internet Appliance consists of Opera's "Opera for Linux" Web browser and Qt/Embedded integrated with PalmPalm's "Tynux," a Linux Operating System optimized for the wireless Internet.

Lineo, Inc., a leading embedded Linux solution provider, is collaborating with Trolltech to offer a complete embedded Linux software solution to manufacturers of embedded devices. The combination of Lineo's Embedix Linux software with Qt/Embedded provides customers with a valuable solution for their embedded Linux software requirements at a low-cost while achieving high performance and rapid time-to-market. Trolltech is a certified partner with Lineo's Partner Connect program.

NEC Electronics is adapting Qt/Embedded to run on NEC's evaluation boards featuring the VR4100™ family of 64-bit MIPS® RISC microprocessors. The VR4100 processors' high speed, low power consumption and high integration make them ideal for use in portable handheld systems. With Qt/Embedded, NEC can offer attractive Linux-based GUIs to their customers.

Ericsson Home Communications is using Qt/Embedded as the GUI solution for their Cordless Screen Phone, the Linux-based HS210. Ericsson made the decision to use Qt/Embedded due to its low memory footprint, dedicated product architecture for embedded systems, and strong level of support. In addition, Ericsson's familiarity with desktop Qt allowed an easy transition to Qt/Embedded.

Appendix 1: Complete API Class List

QAccel	Handles keyboard accelerator and shortcut keys
QAction	Abstracts a user interface action that can appear both in menus and tool bars
QActionGroup	Combines actions to a group
QApplication	Manages the GUI application's control flow and main settings
QArray	Template class that provides arrays of simple types
QAsciiCache	Template class that provides a cache based on char* keys
QAsciiCacheIterator	Iterator for QAsciiCache collections
QAsciiDict	Template class that provides a dictionary based on char* keys
QAsciiDictIterator	Iterator for QAsciiDict collections
QAsyncIO	Encapsulates I/O asynchronicity
QBitArray	Array of bits
QBitmap	Monochrome (1 bit depth) pixmaps
QBitVal	Internal class, used with QBitArray
QBoxLayout	Lines up child widgets horizontally or vertically
QBrush	Defines the fill pattern of shapes drawn by a QPainter
QBuffer	I/O device that operates on a QByteArray
QButton	The abstract base class of button widgets, providing functionality common to buttons
QButtonGroup	Organizes QButton widgets in a group
QByteArray	Array of bytes
QCache	Template class that provides a cache based on QString keys
QCacheIterator	Iterator for QCache collections
QCanvas	2D graphic area upon which QCanvasItem objects exist
QCanvasEllipse	An ellipse with a movable center point
QCanvasItem	The QCanvasItem is an abstract graphic object on a QCanvas
QCanvasLine	A lines on a canvas
QCanvasPixmap	Pixmap with an offset
QCanvasPixmapArray	An array of QCanvasPixmap to have multiple frames for animation
QCanvasPolygon	A polygon with a movable reference point
QCanvasPolygonalItem	A QCanvasItem which renders itself in a polygonal area
QCanvasRectangle	A rectangle with a movable top-left point
QCanvasSprite	Animated moving pixmap on a QCanvas
QCanvasText	A text object on a QCanvas
QCanvasView	A QWidget which views a QCanvas
QCDEStyle	CDE Look and Feel
QChar	Light-weight Unicode character
QCharRef	Helper class for QString
QCheckBox	Check box with a text label
QCheckListItem	Implements checkable list view items
QChildEvent	Event parameters for child object events
QClipboard	Access to the window system clipboard
QCloseEvent	Parameters that describe a close event
QCollection	The base class of all Qt collections
QColor	Colors based on RGB
QColorDialog	Dialog widget for specifying colors
QColorDrag	Drag-and-drop object for transferring colors

QColorGroup	Group of widget colors
QComboBox	Combined button and popup list
QCommonStyle	Encapsulates common Look and Feel of a GUI
QConstString	A QString which uses constant Unicode data
QCopChannel	This class provides communication capabilities between several clients
QCString	Abstraction of the classic C zero-terminated char array (char*)
QCursor	Mouse cursor with an arbitrary shape
QCustomEvent	Support for custom events
QCustomMenuItem	Abstract base class for custom menu items in popup menus
QDataPump	Moves data from a QDataSource to a QDataSink during event processing
QDataSink	Asynchronous consumer of data
QDataSource	Asynchronous producer of data
QDataStream	Serialization of binary data to a QIODevice
QDate	Date functions
QDateTime	Date and time functions
QDial	Rounded rangecontrol (like a speedometer or potentiometer)
QDialog	The base class of dialog windows
QDict	Template class that provides a dictionary based on QString keys
QDictIterator	Iterator for QDict collections
QDir	Traverses directory structures and contents in a platform-independent way
QDns	Asynchronous DNS lookups
QDomAttr	Represents one attribute of a QDomElement
QDomCDATASection	Represents an XML CDATA section
QDomCharacterData	Represents a generic string in the DOM
QDomComment	Represents an XML comment
QDomDocument	The representation of an XML document
QDomDocumentFragment	Tree of QDomNodes which is usually not a complete QDomDocument
QDomDocumentType	The representation of the DTD in the document tree
QDomElement	Represents one element in the DOM tree
QDomEntity	Represents an XML entity
QDomEntityReference	Represents an XML entity reference
QDomImplementation	Information about the features of the DOM implementation
QDomNamedNodeMap	Collection of nodes that can be accessed by name
QDomNode	The base class for all nodes of the DOM tree
QDomNodeList	List of QDomNode objects
QDomNotation	Represents an XML notation
QDomProcessingInstruction	Represents an XML processing instruction
QDomText	Represents textual data in the parsed XML document
QDoubleValidator	Range checking of floating-point numbers
QDragEnterEvent	The event sent to widgets when a drag-and-drop first drags onto it
QDragLeaveEvent	The event sent to widgets when a drag-and-drop leaves it
QDragMoveEvent	Event sent as a drag-and-drop is in progress
QDragObject	Encapsulates MIME-based information transfer
QDropEvent	Event sent when a drag-and-drop is completed
QDropSite	Provides nothing and does nothing
QEucJpCodec	Provides conversion to and from EUC-JP character sets
QEucKrCodec	Provides conversion to and from EUC-KR character sets
QEvent	Base class of all event classes. Event objects contain event parameters

QFile	I/O device that operates on files
QFileDialog	Dialog widget for inputting file names
QFileIconProvider	Icons for QFileDialog to use
QFileInfo	System-independent file information
QFilePreview	Abstract preview widget for the QFileDialog
QFocusData	Maintains the list of widgets which can take focus
QFocusEvent	Event parameters for widget focus events
QFont	Font used for drawing text
QFontDatabase	Provides information about available fonts
QFontDialog	Dialog widget for selecting a text font
QFontInfo	General information about fonts
QFontMetrics	Font metrics information about fonts
QFrame	The base class of widgets that can have a frame
QFtp	Implements the FTP protocol
QGArray	Internal class for implementing the QArray class
QGbkCodec	This class provides conversion to and from the Chinese GBK encoding
QGCache	Internal class for implementing QCache template classes
QGCacheIterator	An internal class for implementing QCacheIterator and QIntCacheIterator
QGDict	Internal class for implementing QDict template classes
QGDictIterator	An internal class for implementing QDictIterator and QIntDictIterator
QGL	Namespace for miscellaneous identifiers in the Qt OpenGL module
QGLayoutIterator	The abstract base class of internal layout iterators
QGLContext	Encapsulates an OpenGL rendering context
QGLFormat	The display format of an OpenGL rendering context
QGList	Internal class for implementing Qt collection classes
QGListIterator	Internal class for implementing QListIterator
QGLWidget	Widget for rendering OpenGL graphics
QGrid	Performs geometry management on its children
QGridLayout	Lays out widgets in a grid
QGroupBox	Group box frame with a title
QGuardedPtr	Template class that provides guarded pointers to QObjects
QGVector	Internal class for implementing Qt collection classes
QHBox	Performs geometry management on its children
QHBoxLayout	Lines up widgets horizontally
QHButtonGroup	Organizes QButton widgets in a group with one horizontal row
QHeader	Table header
QHGroupBox	Organizes widgets in a group with one horizontal row
QHideEvent	The event sent after a widget is hidden
QHostAddress	IP address
QIconDrag	The drag object which is used for moving items in the iconview
QIconDragItem	The internal data structure of a QIconDrag
QIconSet	Set of differently styled and sized icons
QIconView	Area with movable labeled icons
QIconViewItem	Implements an iconview item
QImage	Hardware-independent pixmap representation with direct access to the pixel data
QImageConsumer	An abstraction used by QImageDecoder
QImageDecoder	Incremental image decoder for all supported image formats
QImageDrag	Drag-and-drop object for transferring images

QImageFormat	Incremental image decoder for a specific image format
QImageFormatType	Factory that makes QImageFormat objects
QImageIO	Parameters for loading and saving images
QInputDialog	A convenience dialog to get a simple input from the user
QIntCache	Template class that provides a cache based on long keys
QIntCacheIterator	Iterator for QIntCache collections
QIntDict	Template class that provides a dictionary based on long keys
QIntDictIterator	Iterator for QIntDict collections
QIntValidator	Range checking of integers
QIODevice	The base class of I/O devices
QIODeviceSource	QDataSource that draws data from a QIODevice
QJisCodec	Provides conversion to and from JIS character sets
QJpUnicodeConv	Implementation support for QJisCodec, QSjisCodec, and QEucJpCodec
QKeyEvent	Parameters that describe a key event
QLabel	Static information display
QLayout	The base class of geometry specifiers
QLayoutItem	The abstract items which a QLayout manipulates
QLayoutIterator	Iterators over QLayoutItem
QLCDNumber	Displays a number with LCD-like digits
QLineEdit	One-line text editor
QList	Template class that provides doubly linked lists
QListBox	List of selectable, read-only items
QListBoxItem	This is the base class of all list box items
QListBoxPixmap	List box items with a pixmap and an optional text
QListBoxText	List box items with text
QListIterator	Iterator for QList collections
QListView	Implements a list/tree view
QListViewItem	Implements a list view item
QListViewItemIterator	Iterator for collections of QListViewItems
QLNode	Internal class for the QList template collection
QLocalFs	Implementation of a QNetworkProtocol which works on the local filesystem
QMainWindow	Typical application window, with a menu bar, some tool bars and a status bar
QMap	Value based template class that provides a dictionary
QMapConstIterator	Iterator for QMap
QMapIterator	Iterator for QMap
QMenuBar	Horizontal menu bar
QMenuData	Base class for QMenuBar and QPopupMenu
QMessageBox	Displays a brief message, an icon, and some buttons
QMetaObject	Meta information about Qt objects
QMetaProperty	Stores meta data about a property
QMimeSource	An abstract piece of formatted data
QMimeSourceFactory	An extensible supply of MIME-typed data
QMotifPlusStyle	More sophisticated Motif-ish look and feel
QMotifStyle	Motif Look and Feel
QMouseEvent	Parameters that describe a mouse event
QMoveEvent	Event parameters for move events
QMovie	Incrementally loads an animation or image, signaling as it progresses
QMultiLineEdit	Simple editor for inputting text

QMutex	Access serialization between threads
QNetworkOperation	This class is used to define operations for network protocols
QNetworkProtocol	This is the base class for network protocols which provides a common API for network
QNPInstance	A QObject that is a Web-browser plugin [Qt NSPlugin Extension]
QNPlugin	The plugin central factory [Qt NSPlugin Extension]
QNPStream	A stream of data provided to a QNPInstance by the browser [Qt NSPlugin Extension]
QNPWidget	A QWidget that is a Web-browser plugin window [Qt NSPlugin Extension]
QObject	The base class of all Qt objects
QPaintDevice	Of objects that can be painted
QPaintDeviceMetrics	Information about a paint device
QPainter	Paints on paint devices
QPaintEvent	Event parameters for paint events
QPalette	Color groups for each widget state
QPen	Defines how a QPainter should draw lines and outlines of shapes
QPicture	Paint device that records and replays QPainter commands
QPixmap	Off-screen pixel-based paint device
QPixmapCache	Application-global cache for pixmaps
QPlatinumStyle	Platinum Look and Feel
QPNGImagePacker	Creates well-compressed PNG animations
QPoint	Defines a point in the plane
QPointArray	Array of points
QPopupMenu	Popup menu widget
QPrinter	Paint device that paint on a printer
QProgressBar	Horizontal progress bar
QProgressDialog	Provides feedback on the progress of a slow operation
QPtrDict	Template class that provides a dictionary based on void* keys
QPtrDictIterator	Iterator for QPtrDict collections
QPushButton	Command button
QQueue	Template class that provides a queue
QRadioButton	Radio button with a text label
QRangeControl	Integer value within a range
QRect	Defines a rectangle in the plane
QRegExp	Pattern matching using regular expressions or wildcards
QRegion	Clip region for a painter
QResizeEvent	Event parameters for resize events
QScrollBar	Vertical or horizontal scroll bar
QScrollView	Scrolling area with on-demand scrollbars
QSemaphore	Robust integer semaphore
QSemiModal	The base class of semi-modal dialog windows
QServerSocket	TCP-based server
QSessionManager	Access to the session manager
QSGIStyle	SGI Look and Feel
QShared	The QShared struct is internally used for implementing shared classes
QShowEvent	The event sent when a widget is shown
QSignal	Can be used to send signals without parameters
QSignalMapper	Bundles signals from identifiable senders
QSimpleRichText	A small displayable piece of rich text
QSize	Defines the size of a two-dimensional object

QSizeGrip	Corner-grip for resizing a top level window
QSizePolicy	A layout attribute describing horizontal and vertical resizing
QSjisCodec	Provides conversion to and from Shift-JIS
QSlider	Vertical or horizontal slider
QSocket	Buffered TCP connection
QSocketDevice	Platform-independent low-level socket API
QSocketNotifier	Support for socket callbacks
QSortedList	List sorted by operator< and operator==
QSound	Access to the platform audio facilities
QSpacerItem	The QLayoutItem class that represents blank space
QSpinBox	Spin box widget, sometimes called up-down widget, little arrows widget or spin button
QSplitter	Implements a splitter widget
QStack	Template class that provides a stack
QStatusBar	Horizontal bar suitable for presenting status information
QStoredDrag	Simple stored-value drag object for arbitrary MIME data
QStrIList	Doubly linked list of char* with case insensitive compare
QString	Abstraction of Unicode text and the classic C null-terminated char array (char*)
QStringList	A list of strings
QStrList	Doubly linked list of char*.
QStrListIterator	Iterator for the QStrList and QStrIList classes
QStyle	Encapsulates common Look and Feel of a GUI
QStyleSheet	A collection of styles for rich text rendering and a generator of tags
QStyleSheetItem	Encapsulates a text format
Qt	Namespace for miscellaneous identifiers that need to be global-like
QTab	The structures in a QTabBar
QTabBar	Tab bar, for use in e.g. tabbed dialogs
QTabDialog	Stack of tabbed widgets
QTable	A flexible and editable table widget
QTableItem	Content for one cell in a QTable
QTableSelection	The QTableSelection provides access to the selected area in a QTable
QTableView	This is an abstract base class for implementing tables
QTabWidget	Stack of tabbed widgets
QTextBrowser	A rich text browser with simple navigation
QTextCodec	Provides conversion between text encodings
QTextDecoder	State-based decoder
QTextDrag	Drag-and-drop object for transferring plain and Unicode text
QTextEncoder	State-based encoder
QTextIStream	A convenience class for input streams
QTextOStream	A convenience class for output streams
QTextStream	Basic functions for reading and writing text using a QIODevice
QTextView	A sophisticated single-page rich text viewer
QThread	Platform-independent threads
QTime	Clock time functions
QTimer	Timer signals and single-shot timers
QTimerEvent	Parameters that describe a timer event
QToolBar	Tool bar
QToolButton	Quick-access button to specific commands or options, usually used inside a QToolBar
QToolTip	Tool tips (sometimes called balloon help) for any widget or rectangular part of a widget

QToolTipGroup	Collects tool tips into natural groups
QTranslator	Internationalization support for text output
QTranslatorMessage	Translator message and its properties
QTsciiCodec	Conversion to and from the Tamil TSCII encoding
QUriDrag	Provides for drag-and-drop of a list of URI references
QUrl	Mainly an URL parser and simplifies working with URLs
QUrlOperator	Common operations on URLs (get() and more)
QValidator	Validation of input text
QValueList	Value based template class that provides doubly linked lists
QValueListConstIterator	Iterator for QValueList
QValueListIterator	Iterator for QValueList
QValueStack	Value based template class that provides a stack
QVariant	Acts like a union for the most common Qt data types
QVBox	Performs geometry management on its children
QVBoxLayout	Lines up widgets vertically
QVButtonGroup	Organizes QButton widgets in a group with one vertical column
QVector	Template collection class that provides a vector (array)
QVGroupBox	Organizes widgets in a group with one vertical column
QWaitCondition	Allows waiting/waking for conditions between threads
QWhatsThis	Simple description of any widget, e.g. answering the question "what's this?"
QWheelEvent	Parameters that describe a wheel event
QWidget	The base class of all user interface objects
QWidgetItem	A QLayoutItem that represents widget
QWidgetStack	Stack of widgets, where the user can see only the top widget
QWindowsStyle	Windows Look and Feel
QWizard	Framework for easily writing wizards
QWMatrix	2D transformations of a coordinate system
QWorkspace	Workspace window that can contain decorated windows, e.g. for MDI
QXmlAttributes	XML attributes
QXmlContentHandler	Interface to report logical content of XML data
QXmlDeclHandler	Interface to report declaration content of XML data
QXmlDefaultHandler	Default implementation of all XML handler classes
QXmlDTDHandler	Interface to report DTD content of XML data
QXmlEntityResolver	Interface to resolve extern entities contained in XML data
QXmlErrorHandler	Interface to report errors in XML data
QXmlInputSource	The source where XML data is read from
QXmlLexicalHandler	Interface to report lexical content of XML data
QXmlLocator	The XML handler classes with information about the actual parsing position
QXmlNamespaceSupport	Helper class for XML readers which want to include namespace support
QXmlParseException	Used to report errors with the QXmlErrorHandler interface
QXmlReader	Interface for XML readers (i.e. parsers)
QXmlSimpleReader	Implementation of a simple XML reader (i.e. parser)
QXtApplication	Allows mixing of Xt/Motif and Qt widgets [Qt Xt/Motif Extension]
QXtWidget	Allows mixing of Xt/Motif and Qt widgets [Qt Xt/Motif Extension]