Your First Qtopia Application

Linux Mobile Series

Author:Prashant, Rishi

Mindfire Solutions, www.mindfiresolutions.com

mindfire solutions
exceptional software delivery

Table of Contents

1. Introduction………………………………………………………………… 3

2. Target audience………………………………………………………………3

3. Architecture……………………………………………………………….....4

4. System Requirements……………………………………………………....4

5. Tools & SDK…....…………………………………………………………....5

6. Let's Start with Development…...…………………...……........5

7. Example…………………………………………..……......10

mindfire solutions
exceptional software delivery

1. Introduction

Welcome aboard!!!

So, finally you are ready to develop your first application for LinuxMobile on Qtopia. Let us help you in this endeavor by explaining the nuances of Qtopia programming so that you have a better platform to begin with.

As you must already know by now, Qtopia provides a graphical environment for the embedded devices and we can develop Qtopia application on Linux.

In this article, we assume that you have a working development environment nad have basic knowledge like building and running your application. If not, we suggest you go through our first article *"Integrated Development: KDevelop & Qtopia"*, that will arm you with the basic fundamentals and a proper setup

2. Target Audience

This document is intended as a starting point for the developers planning to start development on LinuxMobile using Qtopia SDK. We give some explanations about base classes and also present you with a HelloWorld example to give you a working program.

3. Architecture

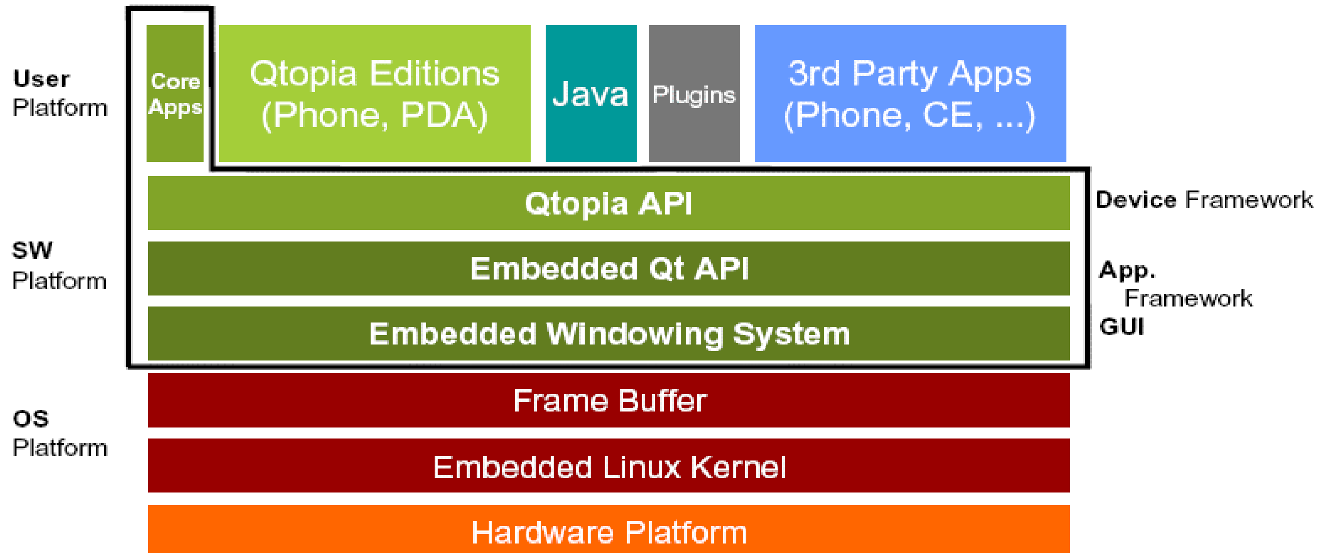

Figure 1: ***Qtopia Platform Architecture***

Note: For the detailed information about the architecture of Linux mobile u can refer to following document. http://www.trolltech.com/pdf/qtopia-wp-platform42.pdf

4. System Requirement

No hardware dependencies – can support any processor and graphics card supported by Linux

Requires Linux kernel with linear frame buffer support (4,8,16, or 32 bit +VGA16 supported)

5. Guidelines before Writing Qtopia Application

Following tools and sdks are required:

- OS: Linux
- IDE: KDevelop3
- Qtopia SDK
- QT Designer2 for UI designing

Along with these tools, you need to have a proper integrated development environment. Environment, where you can build, run and debug your applications. Please refer to our article *"Integrated Development: KDevelop & Qtopia"* for any help.

Once all of the above is in place, we are ready to start with actual development.

6. Let's Start With Development

First things first, creating a new project:

- **Start** Kdevelop3 IDE
- Select the **New Project** from the Project menu:
  Project-> New Project-> C++ -> Embedded-> Qtopia Application
- Give an appropriate **project name** and click OK.
- Set the **environment variables** for the project.
- Your new project is created with basic classes added automatically.

When we create any new Qtopia application, some files come along with it. For example, when we create a new project named **Mindfire,** we get some files automatically:

- MindfireBase.ui
- main.cpp
- Mindfire.cpp
- Mindfire.pro

MindfireBase.ui is used for user interface. You modify this file whenever any UI change is required. *Changes in the ui file should be done using QtDesigner2.*

main.cpp has the **main function** (entry point) from where your application starts execution.

Mindfire.cpp file contains a class named "Mindfire" (class name will be same as project name). Mindfire class is derived from "MindfireBase".

What is MindfireBase class?
Basically this class is generated from UI Compiler along with the .ui class and is derived from **QWidget.**

QWidget is the class that works as a base class for all user interface elements. A Widget is the user interface that receives all types of events, like mouse event, keyboard event, paint event, etc.

Qt has a rich set of widgets (buttons, scroll bars, etc.) that cater for most situations. We can easily subclass these widgets to create custom widgets when required.

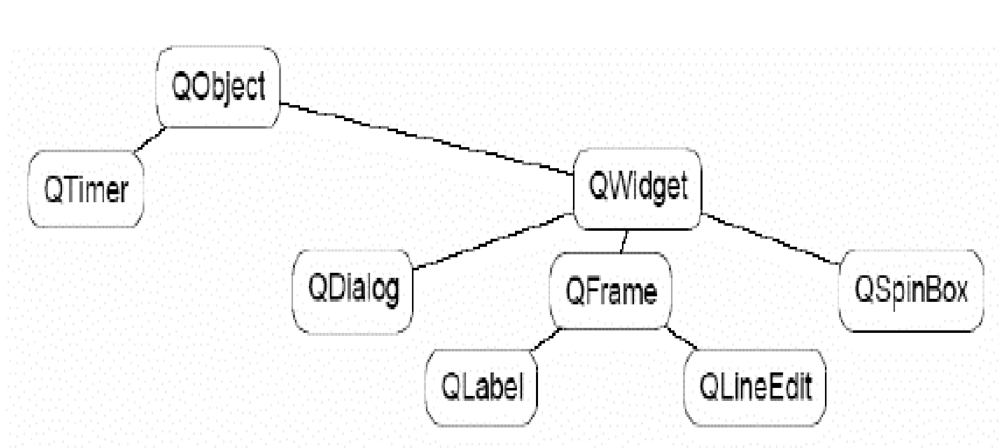

Figure 2: ***QWidget Class Hierarchy***

Mindfire.pro file is the project file usually we call ***project name dot pro***. This file contains the components that are needed to build the project. Following are the contents of the project file:

```
TEMPLATE = app
CONFIG   = qt warn_on release
HEADERS = Mindfire.h
SOURCES = Mindfire.cpp main.cpp
TARGET   = Mindfire
```

TEMPLATE = app, tells tmake to create a make file for an application.

CONFIG variable specifies what compiler option is used and what extra libraries used to link with.

warn_on tells that compiler will display all warnings. If you do not want to display warnings, set this to warn off.
release tells that compiler should compile with optimization enabled.

debug tells the compiler to compile with debug option enabled.

HEADERS contains list of all header files.

SOURCES lists all the source files.

TARGET specifies executable file of your project and it will give 32-bit LSB executable. LSB stands for Linux Standard Base.

When we build the application, the compiler needs to compile our code. Two types of compilers are required:

(a) UIC compiler: it compiles your .ui file and creates .h and .cpp file.
(b) MOC (Meta Object Compiler): It reads cpp file and and generates new file containing meta object code. For example, it reads Mindfire.cpp and generates moc_Mindfire.cpp

Finally, these ***moc*** files are used by gcc and g++ compiler to generate the object code. Both, MOC and UIC compilers come bundled with Qtopia SDK

Now we will be discussing about the entry point of the application and other important functions used in the application.

1. Entry point in the application

Entry point function is generally part of **main.cpp**. It has a **main** function from where your application starts its execution.

```
#include "Mindfire.h"
#include <qpe/qpeapplication.h>

int main( int argc, char **argv )
{
   QPEApplication a( argc, argv );
   Mindfire mw;
   a.showMainWidget( &mw );
   return a.exec();
}
```

In the above function the first line creates an object of class *QPEApplication*. This class models a complete application, and have the logic for event handling, starting up and shutting down, among other things.

The class Mindfire has a main window that can have a frame, menu & other things. Now we use the two functions of class QPEApplication

a. showMainWidget()
By using showMainWidget function we set the newly created window to main Widget.

b. exec()
exec() method is used to run the application. The exec() method does not return until the application is shut down, either because the code calls a method to shut it down, or the user takes an action to shut it down (such as clicking the close icon in the frame)..

2. Connect, Signal & Slot:

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

Qtopia use signals for sending messages between widgets. Every widget has some specific kind of signal it emits and according to signals we call the specific function, using a **Slot**. We can use the **connect** macros to bind up the functionality according to our requirement. **connect** macro is used to initiate the connection to a socket.

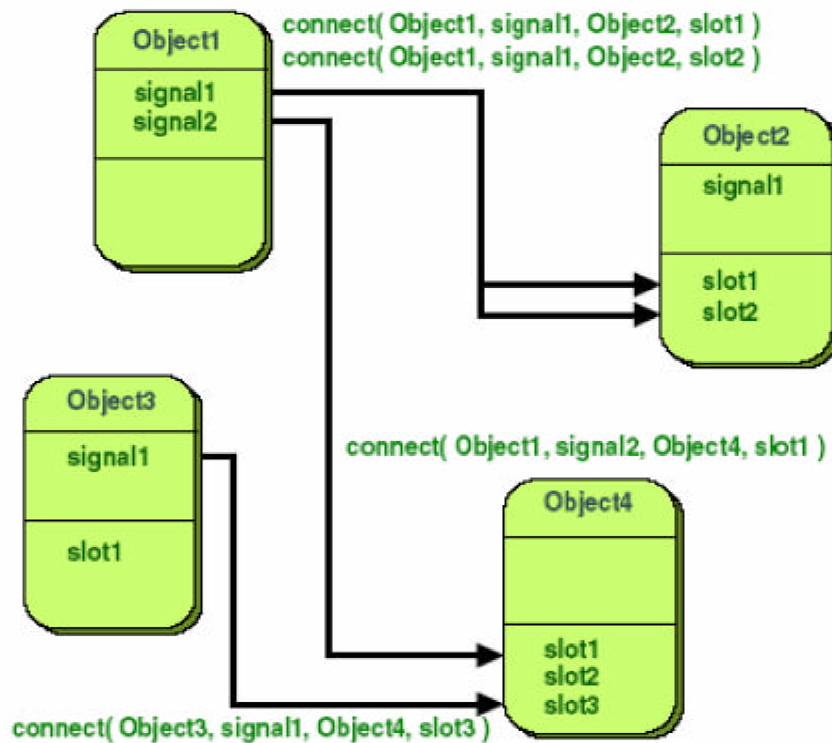

Figure 3: ***Signal & Slot***

For Example:

```
Mindfire:: Mindfire ( QWidget* parent,  const char* name, WFlags fl )
    : MindfireBase( parent, name, fl )
{
    connect(quitButton, SIGNAL(clicked()), this, SLOT(Bye()));
    connect(ComboBox, SIGNAL(activated(const QString &)), this,
              SLOT(comboBoxSelected(const QString &)));
}
```

In the above example we tried to **connect** quitButton's activated signal, and ComboBox's clicked signal to our slots.

Note: while using connect we need to give same argument for the signal & slot of the specific widget.

Image Drawer Example

This sample project is a typical "Hello World" type of example which is no rocket science but will give you an incite into few utility functions and help you start with Qtopia development.

You may find examples on Qtopia’s site but they are not Qtopia phone edition based (QPE). We have taken up the example using QPE so that you do the development for actual platform.

There are few samples in the SDK too but they are not compatible with KDevelop. This example aims to help the beginners by giving the detailed description.

Few things which we will accomplish using this example are:
- Paint a String on screen
- Draw an Image on screen
- Implement Timer functionality
- Trap mouse events.

You can also download the full source zip for the sample project.

We will take up the main classes and discuss their salient features.
Details follow …

Note: *In order to increase the readability and to make it easy to understand, we are explaining each class(QTimers,QImage, QPainter etc) separately.*
You can find the complete collective code in the attached zip file.

mindfire solutions
exceptional software delivery

Main.cpp

```
#include "imagedrawer.h"
#include <qpe/qpeapplication.h>

int main( int argc, char **argv )
{
   QPEApplication  a( argc, argv );
   imagedrawer  mw;
   a.showMainWidget( &mw );
   return a.exec();
}
```

Details

The `main()` function is the entry point to the program. Almost always when using Qt, `main()` only needs to perform some kind of initialization before passing the control to the Qt library, which then tells the program about the user's actions via events.

The `argc` parameter is the number of command-line arguments and `argv` is the array of command-line arguments. This is a standard C++ feature.

The first line in the method creates an object of class ***QPEApplication***. This class models a complete application, and have the logic for event handling, starting up and shutting down, among other things.

By using **showMainWidget** function we set the newly created window to main Widget.

exec() method is used to start application execution and shows the Mindfire window.

imagedrawer.cpp

```
// Constructs a imagedrawer which is a child of 'parent', with the
//  name 'name' and widget flags set to 'fl'
imagedrawer:: imagedrawer ( QWidget* parent,  const char* name, WFlags fl )
    : imagedrawerBase( parent, name, fl )
{
        connect(PushButton1, SIGNAL(clicked()), this, SLOT(goodBye()));
}

//   Draw an image.
{
        QImage mindfire = QImage("mindfire.png")
        QPainter paint (object);
        QPoint p (40,120);
        paint.drawImage (p, mindfire);

        //paint.drawText(QPoint(0,0), "Mindfire Solutions");
}

// Timer implementation.
{
        QTimer *timer = new QTimer( object );
        connect( timer, SIGNAL(timeout()), object, SLOT(imageupdate()) );
        timer->start(4000 ); // 4 seconds single-shot timer
}

// to handle mouse clicks
void imagedrawer::mousePressEvent( QMouseEvent *e)
{
        //do something
}

// A simple slot... not very interesting.
void imagedrawer::goodBye()
{
   // quitting from the application
    close();
}
```

Details

Constructor
The first line shows the constructor of imagedrawer class.The constructor basically, connects the **PushButton1**(Labelled as “Quit” in attached application) to the **goodBye()** function using Signal and Slots.

Drawing Text
Here comes the GUI related part. We have created an object of QPainter class which performs the low level painting on widget and other painting devices. Using the QPainter object we have called the drawText function to display the string at desired location.

Drawing Image
The QImage class provides a image representation that allows direct access to the pixel data, and it can be used as a paint device.
Basically, Qt provides four types of classes to handle the image

QImage - designed and optimized for I/O, and for direct pixel access and manipulation.
QPixmap - designed and optimized for showing images on screen.
QBitmap - is only a convenience class that inherits QPixmap, ensuring a depth of 1.
QPicture - class is a paint device that records and replays QPainter commands.

Because QImage is a QPaintDevice subclass, QPainter can be used to draw directly onto images.

The above example displays mindfire.png at coordinates 40,120.

Handling Timers
The QTimer class provides repetitive and single-shot timers
To implement timers, we need to create a QTimer object, connect its timeout() signal to the appropriate slots, and call start() method to start the timer. Then on it will call the timeout() signal at constant intervals again & again.

The above example will call the imageupdate*()* function, after every 4 seconds.

Handling Mouse Events
If you want to do your programming with mouse events, we need to use QMouseEvent class. QMouseEvent class contains parameters that describes a mouse event.
Mouse events occur whenever a mouse button is pressed, double clicked or released inside a widget, or when the mouse cursor is moved.

Signals & Slots
As described previously, the goodBye() function is called on button press which in turns calls the close function to close the application.

What you see

When you compile the code and run it, it will display "Mindfire Solutions" in character by character typing motion starting from the co-ordinate 75,40 and an image painted at co -ordinate 40,120

Developing for LinuxMobile using Qtopia sound simpler now, doesn't it?