Software Development for Linux PDAs

Zaurus Fodder

If you are familiar with the Qt Toolkit, you may well have been surprised to discover how little skill it takes to write programs for the free PDA desktop, OPIE. The project allows even inexperienced programmers to work constructively.

BY CARSTEN NIEHAUS AND PATRICIA JUNG

No matter whether you run the standard Qtopia desktop for Linux on your PDA or have made the move to OpenZaurus (see page 26) with the Open Palmtop Integrated Environment, OPIE [1], there seems to be no lack of software for the handheld at first sight. After all, hordes of GUI programs for the Linux desktop also use Trolltech's [2] Qt library. It is simply a question of replacing the library with Qt/e, the Qt edition for embedded devices, linking and cross-compiling for the Zaurus PDA's ARM processor – or is it?

In fact, it is not that simple. For one thing the PDA has nothing like the computing power or memory resources of a PC or Mac. For another, most GUI programs for the Linux desktop only make sense when displayed on a PC screen. Displaying them in the same scale on the minute PDA display involves irritating and confusing scrolling operations and severely impacts the usability of the application. Additionally, there are bound to be some applications that make sense for a PDA, but not for a PC.

So, application development for the PDA cannot simply be dismissed as a side effect of desktop software development. Although the output device requires a different approach to software design than a PC or Mac, you should not need to familiarize yourself with new development techniques.

Setting Up the Desktop

Because of the PDA's restricted resources, programmers will tend to use a desktop computer (a PC or Mac) for their development work. This means setting up a suitable developer environment first. An emulator allows you to test your PDA programs as part of the development process. The Qt/e [3] package provides an emulator called *qvfb* ("Qt Virtual Framebuffer") (see Figures 1–3).

As soon as your program is stable and provides the required functionality, you can cross-compile it for the PDA and install the binary on your handheld. This approach allows you to use typical C++ development tools like *gcc*, *gdb*, and *valgrind* (to check for buffer overflows and other programming errors) [4], and means you will not need to install and test the software on the PDA until the rest of the development process has been completed.

Of course you can use a standard editor like *vi* or *Emacs* to write your source code, but KDevelop 3 [5,6], which will probably be released at the same time as KDE 3.2 (although it is already extremely stable), provides an integrated developer environment with OPIE support.

As is the case for many other projects, you can get the latest OPIE version from the CVS repository (see the "OPIE Latest" box). It contains patches for Qt/e and will mean a bit more work. If you simply want to take a quick look at OPIE (not to mention avoiding the frustration of a CVS version that does not compile), you can simply use the *opie-devel-g++-3_x86_0.9.1-ml2.tar.gz* from [7].

OPIE Latest

If you decide to start programming for OPIE, you should consider using the latest version of OPIE and Qt/e. The following commands

```
cvs -d:pserver:anoncvs@cvs.➥
handhelds.org:/cvs login
```

(password: *anoncvs*) and

```
cvs -d:pserver:anoncvs@cvs.➥
handhelds.org:/cvs co opie
export OPIEDIR=/path/to/opie/
```

allow you to create a local copy of the OPIE CVS repositories; and *cd $OPIEDIR; cvs up* will keep your copy up to date. The *$OPIEDIR/qt* directory contains a number of patches for Qt 2.3.x, which can be installed as follows, after unzipping the *qt-embedded-2.3.5.tar.gz* archive

```
cd qt-2.3.5; export QTDIR=`pwd`
cat $OPIEDIR/qt/qte234-for-opie➥
091-gfxraster.patch | patch -p0
```

After copying the OPIE specific configuration file to the Qt tree using the following command

```
cp $OPIEDIR/qt/qconfig-qpe.h ➥
$QTDIR/src/tools
```

you can compile the library (after invoking *configure* as shown in the text).
As soon as the user interface compiler *uic* has taken up residence in *$QTDIR/bin* (cf. the relevant text passage), you can start to compile the latest OPIE version:

```
export PATH=$PATH:$QTDIR/bin
cd $OPIEDIR
make clean
make
```

Again, before launching *make*, you can use *make menuconfig* to change the footprint, target platform and other OPIE characteristics. In fact, this is often essential, as not all the program components will compile. You can use *make menuconfig* to remove the misbehaving components from the compiler pool. More information is available from [13].

Qmake instead of tmake

OPIE developers tend to prefer the more powerful *qmake* tool rather than the *tmake* Perl script to generate makefiles from project files (see text). As Trolltech will not be including *qmake* until Qt version 3.0, you will find it in the OPIE CVS tree.
If you are using *qmake* instead of *tmake*, set the *QMAKESPEC* variable instead of *TMAKEPATH* in Listing 1:

```
export QMAKESPEC=$OPIEDIR/➥
mkspecs/linux-g++/
```

The extended syntax of *qmake* allows you to add the following line to Listing 2:

```
include ( $(OPIEDIR)/include.pro )
```

The included project file contains generic settings for OPIE projects and is only available in the OPIE CVS version.

It contains a customized version of Qt/e for OPIE and the OPIE libraries, although neither of them are exactly spring chickens. Again this will involve some work on your part: If you have not installed *distcc* [11] to provide distributed compilation of C/C++ code, you will need to edit the *qt-2.3.4-beta2/configs/linux-x86-g++-shared* file and replace the word *distcc* in four sections (while setting the *SYSCONF_CXX*, *SYSCONF_CC*, *SYSCONF_LINK*, and *SYSCONF_LINK_SHLIB* variables). Then assign the path to the *qt-2.3.4-beta2* directory to the *QTDIR* environment variable, change to that directory, and type the following to start compiling:

```
./configure -qconfig qpe -depths
 8,16,24 -qvfb -vnc -system-
jpeg -system-libpng -system-zlib
-gif -no-xft make
```

Link *uic-qt2* from [7] to *$QTDIR/bin/uic* (and while you are at it, *qvfb-qt2* from [7] to *$QTDIR/bin/qvfb*) before going on to create the OPIE libraries. You will first need to set the *LD_LIBRARY_PATH* to point at *$QTDIR/lib* to ensure that the Qt/e binary you are compiling will be found. After doing so, follow the instructions in the *opie/README.NEWBUILD* file. As the basic configuration, which you can copy from *opie/def-configs/opie* to *opie/.config* is perfectly ok for the time being, you can leave out the last step, *make menu config*, and simply type *make*.

After completing these steps, you will of course have compiled libraries and tools for the PC. As the PDA does not have an Intel compatible processor, but an ARM processor instead, you cannot link software for the PDA against the PC libraries. In other words, you will need to re-compile the Qt/e and OPIE libraries for the PDA using a cross-compiler [10].

To do so, you should copy the whole *qt-2.3.4-beta2* directory to *qt-2.3.4-beta2_pda*, for example.

A number of environment variables, specifically *QTDIR*, need to be set to reflect the task in hand, that is, compiling a test version of an OPIE program for the PC, or cross-compiling a version for the PDA. A script like the one shown in Listing 1 will help you avoid confusion while doing so. Running the script without any arguments will set the variables for a PC compilation. Providing any argument will set up the environment for cross-compilation. As a side effect, the script also sets the *OPIEDIR* variable that points to the OPIE root directory, and *TMAKEPATH* that points to the configuration directory for the *tmake* tool that also ships with *opie-devel-g++3_x86_0.9.1-ml2.tar.gz*.

The Framework

At last you can start creating an OPIE program designed to display information about a file – to be more precise, the file size, the last modification date, the owner and the access privileges. All of these functions would be useful for a file manager program.

To do so, the program launches a dialog box that displays the file and directory tree. When the user selects a

Translation

Most software projects opt to display desktop texts (such as menu items or error messages) in English. Programs designed for non English speakers need to be internationalized and localized if these messages are to be displayed in the local language. Qt provides the *QApplication::translate()* function for this purpose, allowing the program to display the required desktop language at runtime. This is typically the language defined by the appropriate system variable. Programmers only need to watch out for two things when internationalizing a program – any messages the user may be confronted with at any time must be wrapped in the *tr()* function. The following example shows the *File* menu:

```
menubar->insertItem( tr( "File" 
), file );
```

The program will then pass this string to *QApplication::translate()* at runtime. The function locates a translation for "File" in the selected language within the translation file and simply replaces the string. Assuming the translation is available Qt – thanks to Unicode (UTF8) support – can even show GUI elements with Japanese or traditional Chinese texts.

Programmers should also get used to handling arguments as follows:

```
label.setText( "The City %1 has 
%2 inhabitants").arg(cityName).
arg(cityInhabitants);
```

The following code can cause a problem:

```
QString city_text = tr( "The 
City" ) + cityName + tr( "has" )
+ cityInhabitants + tr
( "inhabitants" );
label.setText( city_text );
```

The reason is that this assumes that English syntax (and thus the order of the arguments) can be translated word for word to any other language. This does not even apply to languages closely related to English and is particularly disastrous in the case of languages written from right to left.

The Qt Linguist [9], which is also available from [7], provides some useful tools for localization, such as *lupdate* and *lrelease*.

Listing 1: Compiling or cross-compiling?

```
#!/bin/bash
export OPIEDIR=$HOME/opie/opie
if [ "x${1}x" = "xx" ]; then
    export QTDIR=$HOME/qt-2.3.4-beta2
    export TMAKEPATH=$HOME/tmake/1.8/lib/qws/linux-generic-g++
 else
     export QTDIR=$HOME/qt-2.3.4-beta2_pda
     export TMAKEPATH=$HOME/tmake/1.8/lib/qws/linux-sharp-g++
 fi
 export PATH=$QTDIR/bin:$HOME/tmake/1.8/bin:$PATH
 export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

Listing 2: The project file

```
TEMPLATE     = app
CONFIG       = qt warn_on release
DESTDIR      = $(OPIEDIR)/bin
HEADERS      = mainwindow.h
SOURCES      = main.cpp mainwindow.cpp
INCLUDEPATH += $(OPIEDIR)/include 
$(OPIEDIR)/libopie
DEPENDPATH  += $(OPIEDIR)/include 
$(OPIEDIR)/libopie
LIBS        += -lqpe -lopie
TARGET       = testapp
```

file a new dialog box appears displaying the file information.

The program is called *testapp* and the source code is available from [12]. We will be creating a project directory of the same name in which the project file *testapp.pro* (Listing 2) and the source code will be stored.

testapp.pro uses the *HEADERS* and *SOURCES* lines to specify the project files. As it is an OPIE program, the binary we will be creating (*TARGET*), *testapp* must be linked against *libopie*; the *LIBS += lopie* directive tells the linker this.

We can now pass the project file as an argument to the *tmake* tool to create a makefile. If you have previously installed a developer environment for a desktop version of Qt, you can alternatively use the newer *qmake* tool, which is available with Qt 3.0 or later. If you have any trouble with this step, the *testapp.tar.bz2* tarball at [12] includes a functional makefile, although you will need to modify a few paths.

Listing 3 finally contains some programming. The application itself is created as a *QPEApplication* type object and displays a *MainWindow* type object. This is the only self-written class the program uses. It is implemented in the *mainwindow.cpp* file; extracts of this file follow.

The Main Thing

MainWindow derives from the Qt *QMainWindow* class, which implements the main window in a GUI application. Objects belonging to this class can contain toolbars, a status bar, and a menu bar. To keep the code readable, our sample program will only have a menu bar:

```
QPEMenuBar *menubar = new ➥
QPEMenuBar( this );
```

Listing 3: main.cpp

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

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

Figure 1: The sample program in qvfb

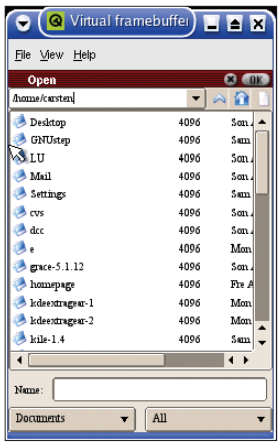

Figure 2: The File dialog box being created in our example

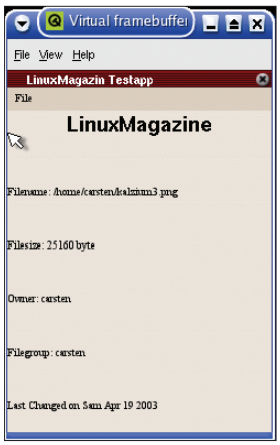

Figure 3: The file information is shown

QPEMenuBar is a class that was specifically extended for PDAs. It inherits from *QMenuBar*. (To understand the name, recall that Trolltech's Qtopia desktop for PDAs was formerly called QPE – "Qt Palmtop Environment".)

The lines that follow (Listing 4) are responsible for defining the menu and the actions it invokes. First, a new menu is created, and inserted into the menu bar in line two. This is also the point where the name *File*, the text displayed to users later (see Figure 1), is assigned to the menu. The "Translations" box explains the *tr()* command, which is used here and at other positions in the source code.

When selected, a menu should display sub-items that in turn invoke some kind of action. To do so, an instance, *a*, of the class *QAction* is created. A *QAction* represents an action in a menu or toolbar. In our example, the menu is called *New*. When selected, it opens a dialog box where the user can select a file.

A picture says more than a thousand words; this is why OPIE provides a whole range of icons that allow you to enhance your menu entries. *Resource::loadPixmap("new")* places an icon next to the *New* entry to illustrate the action invoked by this entry, that is, selecting a file.

Signals and Slots

The second to last line in Listing 4 shows one of the most important tricks for Qt programmers, the signal-slot mechanism. Every object derived from the *QObject* class (these include every GUI element, for example) can use the *emit* function (often automatically) to generate a signal when an event occurs. For example action *a* emits the signal *activated()* when the relevant menu item has been selected. Signals are collected by slots linked to each individual signal.

In our example the *slotfileNew()* slot of the same (*this*) class, *MainWindow*, reacts to the *activated()* signal. The slot is a normal function that can also be called directly. To allow the function to be linked to a signal in Listing 4, the slot is declared as such in the *mainwindow.h* header file:

```
private slots:
    void slotfileNew();
```

The implementation of the *MainWindow::slotfileNew()* function in *mainwindow.cpp* specifies what happens when a user enables the *New* menu item: in

Listing 4: A menu

```
QPopupMenu *file = new QPopupMenu( this );
menubar->insertItem( tr( "File" ), file );
QAction *a = new QAction( tr( "New" ), Resource::loadPixmap( "new" ),➥
 QString::null, 0, this, 0 );
connect( a , SIGNAL( activated() ), this, SLOT( slotfileNew() ) );
a->addTo( file );
```

this case a file dialog box is created (see Figure 2).

The signal-slot mechanism simplifies communications between objects as the objects involved do not need to see each other. Additionally, the mechanism is typesafe, that is, it can handle all types and any quantities of data.

A Special OPIE Class

Up to this point – with the exception of the embedded icon – we have not used any classes and functions from the OPIE API itself. The file dialog displayed by the *slotfileNew()* function changes this. This dialog that provides convenient file selection functions is implemented by the *OFileDialog* class. If the user then clicks on *OK*, the following line of code will store the return value of the *OFileDialog* function *getOpenFileName*, that is, the name of the selected file with its full path, in a private class variable of the *MainWindow* class:

```
QString fileName=OFileDialog::➥
getOpenFileName( OfileSelector➥
::EXTENDED , Qdir::➥
homeDirPath() );
```

The next step is to parse the required information for this file. Qt/e uses the *QFileInfo* class for this purpose. The class allows for simple parsing of the file size and privileges, for example.

The *MainWindow* class saves the file size in the private class variable *fSize*, which is an *unsigned int* type, the owner in *QString fOwner*, the group in *QString fGroup*, the file name in *QString fileName* and the last modification date and time in *fLastMod*, which is an instance of the Qt *QDateTime* class, specifically designed for this purpose.

Listing 5: Parsing file information

```
void MainWindow::getFileInfos➥
( QString fName )
{
   fileInfo.setFile( fName );
   fSize = fileInfo.size();
   fOwner = fileInfo.owner();
   fGroup = fileInfo.group();
   fLastMod = fileInfo.➥
   lastModified();
}
```

Listing 5 shows how these variables can be filled with the values pertaining to the selected file. The *MainWindow getFileInfos()* function is responsible for this; it expects to be passed a file name when called.

We still need to display this information. Lines 64 through 68 in the *mainwindow.cpp* file take care of this by writing the contents of the variables (and an explanation) to five *QLabel* type objects created for this purpose:

```
fileNameLabel->setText( tr( ➥
"Filename: %1").arg( ➥
fileName ) );
```

To distribute the five labels evenly over the amount of space available for the display, as shown in Figure 3, they are placed in a *QVBoxLayout* one after another. *QVBoxLayout* arranges any widgets assigned to it vertically.

```
QVBoxLayout *v_layout = new ➥
QVBoxLayout( mainWidget );
[...]
v_layout->addWidget➥
( fileNameLabel );
```

The *QLabel* class provides various ways of manipulating label texts, changing the font size, type or justification, for example. The sample program utilizes this feature in the *Linux Magazine* title shown in Figure 3; the title is described by a label object called *topicLabel*:

```
topicLabel->setAlignment➥
( AlignHCenter )
```

will center the text;

```
QFont f1 (  "times", 18, Qfont➥
::Bold );
topicLabel->setFont( f1 );
```

will display the title in 18 point Times bold. There are also ways of enabling or disabling line wrapping.

Ready, steady, go!

However, the prettiest of programs is useless, if you cannot test it. To do so, we finally get round to launching the *qvfb* emulator. Do not worry about the black window that appears – the test application should be up a short time later. If required, you can type the following command

```
QWS_DISPLAY=QVFb:0
```

to ensure that the display gets to the right spot. If you used *tmake* or *qmake* to create a makefile for the test program, you can simple enter *make* in the project directory to start compiling. But even if everything works out fine, you might be surprised to discover that the project directory does not contain a single binary executable. The executable will be in the *$OPIEDIR/bin* directory instead. If you launch the program with the *-qws* option, *qvfb* will suddenly spring to life (see Figure 1).

If you have enjoyed this excursion, and are looking to embark on your own OPIE projects, you will be pleased to hear that help is at hand if you encounter difficulties. The OPIE homepage provides more information and you can add your name to the *opie-devel* mailing list, which OPIE developers use to exchange ideas and help each other out. ■

INFO

[1] OPIE: *http://opie.handhelds.org/*

[2] Trolltech AS: *http://www.trolltech.com/*

[3] Qt/e download: *ftp://ftp.trolltech.com/qtopia/source/qt-embedded-2.3.5.tar.gz*

[4] Valgrind: *http://developer.kde.org/~sewardj*

[5] KDevelop 3 download: *http://www.kdevelop.org/index.html?filename=download.html*

[6] KDevelop 3 Tutorial: *http://www.kdevelop.org/doc/tutorial_qtopia/*

[7] Developer tools for OPIE: *http://opie.net.wox.org/tools/*

[8] Compiling Qt/e for OPIE: *http://opie.handhelds.org/docs/opie-cookbook/x22.html#AEN81*

[9] Qt Linguist Manual: *http://doc.trolltech.com/3.1/linguist-manual.html*

[10] Preparing to cross-compile: *http://opie.handhelds.org/docs/opie-cookbook/x116.html*

[11] Distcc: *http://distcc.samba.org/*

[12] *http://www.handhelds.org/~cniehaus/*

[13] *http://opie.handhelds.org/wiki/index.php/SourceCode*