한국정보통신대학교
Information and Communications University

RESL@ICU
Real-time and Embedded Systems Laboratory

Hand-Held devices and Embedded Systems Course (Spring 2004)

Porting Qt/Embedded to M9328MX1ADS board
&
Building a simple Qt/embedded application

Student: Tomás Sánchez López
Student nº: 20042116

Porting Qt/embedded and Qtopia to a M9328MX1ADS board

This report deals with the steps taken to port and install the Qt/embedded and Qtopia environments to a Motorola demonstration board M9328MX1ADS, based on an ARM microprocessor.

The general idea is that we have to cross-compile in a Linux OS (based in a i386 architecture) for an ARM microprocessor, so the software compiled can be executed in the target board, which architecture is different from the host machine. But the path to a successful port in not easy, and the severe lack of organized documentation in English has make this work tedious and sometimes frustrating (I must say that the Korean students, specially those who where provided with an EMPOS system, had a book specifying all the steps to be done for this and other configurations) Anyway, here it goes the result of my porting:

- **General Outlook**

As said before, the general idea is to cross-compile the software in the host machine for the target board. For this, we first, obviously, have to get the software to compile and the cross-compiler to be used. Trolltech (www.trolltech.com) offers in its web site the packages for Qt/embedded, Qt/x11 and Qtopia, all of them under a GPL license for free download. Of course, they don't say anything about he compilers, which we should look through other sources.

After finding the proper compiler and installing it, we should cross-compile, first Qt/embedded, then Qt/x11 and then Qtopia using the Qt/embedded libraries (compiled previously). We will need Qt/x11 for developing using the X server in a Linux host, as to build applications we will typically compile and test for the host machine and then cross-compile and port to the target board (this will save us a lot of time).

1.Getting the Software and the compilers

As said before, better if we get the last versions of Qt/embedded and Qtopia from the web site of Trolltech. In this case, we ge the following:

- Qt/embedded 2.3.7 (qt-embedded-2.3.7.tar.gz)
- Qt/X11 2.3.2 (qt-x11-2.3.2.tar.gz)
- Qtopia 1.7.0 (qtopia-free-1.7.0.tar.gz)

Copy to */opt*, and uncompress:

```
# tar -xvzf qt-embedded-2.3.7.tar.gz
# tar -xvzf qt-x11-2.3.2.tar.gz
# tar -xvzf  qtopia-free-1.7.0.tar.gz
```

About the compilers, we have to bear in mind that the C library to use should be the 2.1.3 version, as the root disk and kernel provided for the ADS are compiled with this version (we could always re-compile all the root disk, including applications, other libraries, etc, etc... couldn't we?) My option was to download the tool-chain (which includes the compilers g++ and gcc for ARM) and the libraries in an RPM format to install them easily (and avoid downgrading procedures of compilers including other version libraries, as described in the BSP documentation and the course material) So that, the RPM obtained were the following:

- arm-linux-glibc-2.1.3-2.i386.rpm
- arm-linux-gcc-2.95.3-2.i386.rpm
- arm-linux-binutils-2.11.2-2.i386.rpm

Note that the order of installation is important due to dependences. To install, just input:

```
# rpm -ivh arm-linux-glibc-2.1.3-2.i386.rpm
# rpm -ivh arm-linux-gcc-2.95.3-2.i386.rpm
# rpm -ivh arm-linux-binutils-2.11.2-2.i386.rpm
```

2.Compiling Qt/embedded

We now want to compile the library of Qt/embedded to proceed further with Qtopia and our own applications. Before compiling, we have to be very much aware that the environment variables are key for a proper compilation, as the makefiles will look for files to be included and etc in the proper directories. As we need to know the directory of Qtopia for proper tools compilation, the first thing to do is the following:

```
# export QPEDIR=/opt/qtopia-free-1.7.0
```

Now we can proceed with the configuration and compilation of Qt/embedded:

```
# export QTDIR=/opt/qt-2.3.7
# cd $QTDIR
# export QTEDIR=$QTDIR
# export PATH=$QTDIR/bin:$PATH
# export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
# cp $QPEDIR/src/qt/qconfig-qpe.h src/tools/
# ./configure -qconfig qpe -qvfb -depths 16 -xplatform linux-ipaq-g++
# make sub-src
```

Note that the platform we are compiling for is called *linux-ipaq-g++*. Although compilation can simply be done for the platform *linux-arm-g++*, some problems on deployment phase with the mouse on the ADS seem to be solved using the *ipaq* platform option.

After compiling, we get the following message:

```
The Qt library is now built in ./lib
```

```
The Qt examples are built in the directories in ./examples
The Qt tutorials are built in the directories in ./tutorial

Note: be sure to set $QTDIR to point to here or to wherever
you move these directories.

Enjoy!   - the Trolltech team
```

So that's it. Now in *$QTEDIR/lib* we have the library that we just compiled:

```
# ls -la
total 3980
drwxr-xr-x    3 518      519          4096 May 13 18:09 .
drwxr-xr-x   14 518      519          4096 May 13 18:18 ..
drwxr-xr-x    2 518      519          8192 Jul 17  2003 fonts
lrwxrwxrwx    1 root     root           15 May 13 18:09 libqte.so -> libqte.so.2.3.7
rwxrwxrwx    1 root     root           15 May 13 18:09 libqte.so.2 -> libqte.so.2.3.7
rwxrwxrwx    1 root     root           15 May 13 18:09 libqte.so.2.3 -> libqte.so.2.3.7
rwxr-xr-x    1 root     root      4054664 May 13 18:09 libqte.so.2.3.7
```

Our library is called *libqte.so.2.3.7* and is around 4Mb.

3.Compiling Qt/X11

So as said before, we need Qt/X11 for later development on the X windowing system of Linux. We can do this because we activated the support for the framebuffer on X11 when compiling Qt/embedded (-qvfb option when configuring). This is because Qt draws directly on the framebuffer to avoid overhead such as the one of X servers, so for developing in such X servers (that is what we are doing on Linux) we need to emulate the framebuffer.

The configuration and compilation procedure is as follows:

```
# export QTDIR=/opt/qt-2.3.2
# cd $QTDIR
# export PATH=$QTDIR/bin:$PATH
# export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
# ./configure
# make
# make -C tools/qvfb
# mv tools/qvfb/qvfb bin
# cp bin/uic $QTEDIR/bin
```

Note that we don't specify this time any compilation platform, as we are using the default one, that is *linux-i386* (we are not going to run this in the ADS, but in the host computer).

Apart from the compilation of some tools specified there, we want to compile also the Qt Designer. To do this, first issue the command “make clean”, and then proceed to the compilation

on the directory *tools/designer* (using the makefile found there). If you have installed RedHat Linux with full development support, most likely you will have a Qt Designer already installed in the OS, and if you have the latest versions of RedHat, most likely the version is version 3. We have to be careful with this, as the tools provided by our Qt/X11 are for version 2, and we wont be able to convert and compile our *ui* files generated by the Designer v.3 with those tools (so thats why we have to compile the version included with the Qt/X11 package)

4.Compiling Qtopia

Finally we got to the point of compiling Qtopia with the libraries we cross-compiled before. But what it doesn't say anywhere is that there are a couple of libraries, that are required by the installation, and are not included either by RedHat or by the packages we downloaded. We realize about this fact when, compiling Qtopia, we get this message:

```
arm-linux-gcc -L/opt/qtopia-free-1.7.0/lib -Wl,-rpath,/opt/qtopia-free-1.7.0/lib -L/opt/qt-2.3.7/lib -Wl,-rpath,/opt/qt-2.3.7/lib -o /opt/qtopia-free-1.7.0/bin/sysinfo .obj/linux-arm-g++//memory.o .obj/linux-arm-g++//graph.o .obj/linux-arm-g++//load.o .obj/linux-arm-g++//storage.o .obj/linux-arm-g++//versioninfo.o .obj/linux-arm-g++//sysinfo.o .obj/linux-arm-g++//main.o .obj/linux-arm-g++//moc_memory.o .obj/linux-arm-g++//moc_graph.o .obj/linux-arm-g++//moc_load.o .obj/linux-arm-g++//moc_storage.o .obj/linux-arm-g++//moc_versioninfo.o .obj/linux-arm-g++//moc_sysinfo.o -lqpe -lqtopia -lqte -lm
 /usr/local/arm-linux/arm-linux/bin/ld: cannot find -lqtopia collect2: ld returned 1 exit status
make[1]: *** [/opt/qtopia-free-1.7.0/bin/sysinfo] Error 1
make[1]: Leaving directory `/opt/qtopia-free-1.7.0/src/applications/sysinfo'
make: *** [applications/sysinfo] Error 2
```

What is happening here? Well, if we take a look up, we will see that for some reason the Qtopia library couldnt be compiled. This is because we lack a library, as I said, that is called *libuuid*, which can be found and compiled from the package *e2fsprogs*. To repair this problem, we should download the package from the Internet and cross-compile it to get the needed library. Provided that the package is called *e2fsprogs-1.35.tar.gz*:

```
# tar -xvzf  e2fsprogs-1.35.tar.gz
# cd  e2fsprogs-1.35
# ./configure --with-cc=arm-linux-gcc –disable-nls
# make
```

This way we get the library *libuuid.a* on the directory *lib.* Note that for a proper cross-compilation, we need to specify which compiler to use with the option *–with-cc=arm-linux-gcc*. Note also that, as we don't need native language support, we use the option *–disable-nls* to reduce the size of the library.

Nest step is to copy the library to the proper location. To find out where our compiler looks for the libraries, we can use the option *-print-search-dirs* when calling it. So:

```
# arm-linux-g++ -print-search-dirs
```

In our case, the output is the following:

```
install: /usr/local/arm-linux/lib/gcc-lib/arm-linux/2.95.3/
programs: /usr/local/arm-linux/lib/gcc-lib/arm-linux/2.95.3/:/usr/local/arm-linux/lib/gcc-lib/arm-linux/:/usr/lib/gcc/arm-linux/2.95.3/:/usr/lib/gcc/arm-linux/:/usr/local/arm-linux/arm-linux/bin/arm-linux/2.95.3/:/usr/local/arm-linux/arm-linux/bin/
libraries:  /usr/local/arm-linux/lib/gcc-lib/arm-linux/2.95.3/:/usr/lib/gcc/arm-linux/2.95.3/:/usr/local/arm-linux/arm-linux/lib/arm-linux/2.95.3/:/usr/local/arm-linux/arm-linux/lib/
```

We decide to put the new library in */usr/local/arm-linux/arm-linux/lib/*. But this is not enough. For some reason, the compiler looks for the library *libuuid.so* and not *libuuid.a*. Well, just we need to change the name with the command *mv*.

About the other library, we get a similar error while compiling, this time with *-ljpeg*. Repeating in a similar way the previous procedure, we download the package *jpegsrc.v6b.tar.gz*:

```
# tar -xvzf  jpegsrc.v6b.tar.gz
# cd  jpeg-6b
# ./configure
```

Before making, we realize a problem. There is no option in this ./configure for choosing an alternative compiler. Well, to solve this, we just need to get into the Makefile and change the line "CC= gcc" by "CC= arm-linux-gcc". Then we can already type "make", copy the library libjepg.a to the same location as the other one and also change the name.

So now we are ready to configure and compile Qtopia:

```
# export QTDIR=$QTEDIR
# export QPEDIR=/opt/qtopia-1.7.0
# cd $QPEDIR
# export PATH=$QPEDIR/bin:$PATH
# cd src
# ./configure -xplatform linux-ipaq-g++
# make
```

No success message, but all the stuff is compiled as it should be.

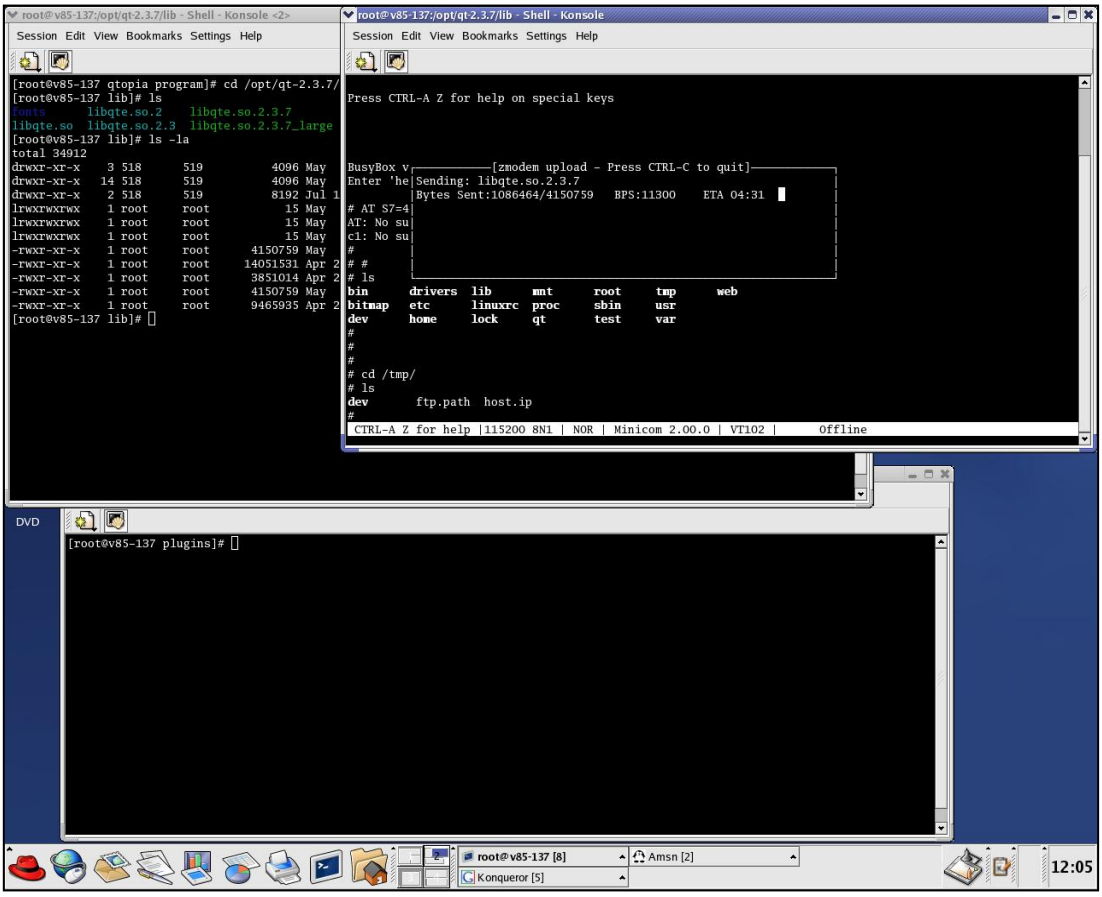
5.Testing our Qt compilation

OK, is compiled... but does it work? Before moving the whole thing to the ADS (for what, as we will see afterward, we have to mount the root file system, change it, and make the binary image of it again), lets try to run the examples that where compiled when we compiled Qt/embedded, to see if the library, the heart of Qt, is working properly. For this, we make a self-made package containing the library, the font directory and our example. We build a typical structure:

root
  |----- examples
  |----- lib
        |------fonts

In the *examples* directory, we include a cross-compiled example such as the typical “hello world”. In the *lib* directory, we include the *lib* directory of our */opt/qt-2.3.7* directory. As the amount of fonts is too much, we take a look to the *fonts* library in the ADS version, and include just the same ones. Finally, we *tar* the tree with “*tar -cf test.tar root”*.

To send this package to the ADS, we can use several methods, such as NFS, USB or Zmodem protocol. I chose this last one (although a bit slow, is simple) For this, we have to set-up the *minicom* to connect to the ADS, the same way we did in the previous homework. Once connected, we should change the directory to one with writing permissions (such as */tmp*), and with CTRL-Z A, and S, we chose the package and send it (about 5Mb)



Next step is to *untar* it on the ADS. For that, just need to type “tar -xf test.tar”. We also need to set some eviroment variables for our “hello world” binary to find the chared library that

we want (our new compiled *libqte.so*):

```
# export QTDIR=/tmp/test
# export QWS_MOUSE_PROTO=TPanel:/dev/h3600_ts
# export LD_LIBRARY_PATH=/tmp/test/lib:$LD_LIBRARY_PATH
```

We just need to go to the */tmp/test/examples* directory and execute the *hello* binary (don't forget to use the *-qws* option to specify that we want to run it as a server):

It works!

6.Installing Qtopia on the ADS

For using our version of Qtopia, we need to replace the old one with our files. The strategy followed by the system in the ADS is to provide a gzipped file, which would be un-gzipped and run executing the script called *startQT* at */usr/sbin:*

```
cd /tmp

echo "extract QTE desktop..."
cat /qt/QtPalmtop.tgz | gunzip - | tar x
```

```
export HOME=/tmp/QtPalmtop
export QTDIR=/tmp/QtPalmtop
export QPEDIR=/tmp/QtPalmtop
export QWS_MOUSE_PROTO=TPanel:/dev/h3600_ts
export LD_LIBRARY_PATH=/tmp/QtPalmtop/lib:$LD_LIBRARY_PATH

/tmp/QtPalmtop/bin/qpe -qws &
```

So all we have to do is to substitute the */qt/QtPalmtop.tgz* by our version. The problem comes when we realized that a big part of the file system is read-only, including the directory */qt*. This mean that the only way to modify it is modifying its image in the host PC and installing again totally to the ADS. To do this, we can mount the image provided by issuing the command "mount -t cramfs -o loop root.cramfs disk", where *root.cramfs* is the image provided with the ADS and *disk* is an empty directory where we are mounting. After this, we should copy the whole thing to other directory, unmount he image, modify whatever we want to and create another image with the command "mkcramfs newdisk newroot.cramfs", where *newdisk* is the directory of our modifications and *newroot.cramfs* is the new image of the file systems.

But what should we include in the gzipped file? Well, of course, the library files and fonts we included when testing the Qt/embedded, plus all the necessary files from Qtopia. We should be careful about this last thing, because the total uncompressed package shouldn't exceed the 20Mb. We can easily compare the directories and files from the package included with the ADS and remove the unnecessary stuff, such as the documentation directories and the source directories.

To transfer the new root file system to the ADS, we can follow the same steps described in the previous homework, using the USB cable in Windows OS. Finally, we connect again to the ADS and check that everything went OK:

Building a Qt/embedded application

Once installed Qt/embedded in the ADS and tested one of the examples included with the package, building a simple application is quite straight forward (provided you know how to program in Qt, of course)

This report exposes the code of a simple application which does nothing but showing up to 10 different widgets and some connections between them. This way, it pretends to prove that the student understands the basics of Qt/embedded and the cross-compilation for the target board

1. Code

The code of the application is shown following:

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>
#include <qgrid.h>
#include <qradiobutton.h>
#include <qcheckbox.h>
#include <qbuttongroup.h>
#include <qlayout.h>
#include <qlistbox.h>
#include <qprogressbar.h>
#include <qlabel.h>

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

Grids::Grids( QWidget *parent, const char *name )
        : QVBox( parent, name )
{

  QButtonGroup *group1 = new QButtonGroup(2, Vertical,"Group of Grids",this);


    QGrid *gridb = new QGrid( 2, group1 );
    QGrid *gridc = new QGrid( 2, group1 );

    for( int c = 0 ; c < 2 ; c++ ) (void)new QRadioButton("radio button", gridb);
    for( int c = 0 ; c < 2 ; c++ ) (void)new QCheckBox("check box", gridc);
```

```
}

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

LCD::LCD( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd"  );
    QSlider * slider = new QSlider( Horizontal, this, "slider" );
    slider->setRange( 0, 99 );
    slider->setValue( 0 );
    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}

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

Progress::Progress( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QProgressBar *progress  = new QProgressBar(this, "progress");
    QSlider *slider2 = new QSlider( Horizontal, this, "slider2" );
    slider2->setRange( 0, 99 );
    slider2->setValue( 0 );
    connect( slider2, SIGNAL(valueChanged(int)), progress, SLOT(setProgress(int)) );

}

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

LabelList::LabelList( QWidget *parent, const char *name )
        : QVBox( parent, name )
{
    QLabel *label = new QLabel("Next Widget is a List", this);

    QListBox *list = new QListBox(this, name);
    list->insertItem("Item number 1",-1);
    list->insertItem("Item number 2",-1);
    list->insertItem("Item number 3",-1);
}

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

```
};

MyWidget::MyWidget( QWidget *parent, const char *name )
       : QWidget( parent, name)
{

   Grids *grid1 = new Grids(this);
   LCD *lcd = new LCD(this);
   LabelList *label = new LabelList(this);
   Progress *progress = new Progress(this);

   QPushButton *quit = new QPushButton( "Quit", this, "quit" );
   quit->setMinimumSize( 100, 50 );
   quit->setMaximumSize( 100, 50 );
   quit->setFont( QFont( "Times", 18, QFont::Bold ) );

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

   QGridLayout *gridl = new QGridLayout( this, 3, 2, 10 );

   gridl->addWidget(grid1,0,0);
   gridl->addWidget(lcd,1,0);
   gridl->addWidget(quit,2,0);
   gridl->addWidget(label,0,1);
   gridl->addWidget(progress,1,1);
   gridl->setColStretch( 1, 10 );

}

int main( int argc, char **argv )
{
   QApplication a( argc, argv );

   MyWidget w;
   w.setCaption("Homework");
   a.setMainWidget( &w );
   w.show();
   return a.exec();
}
```

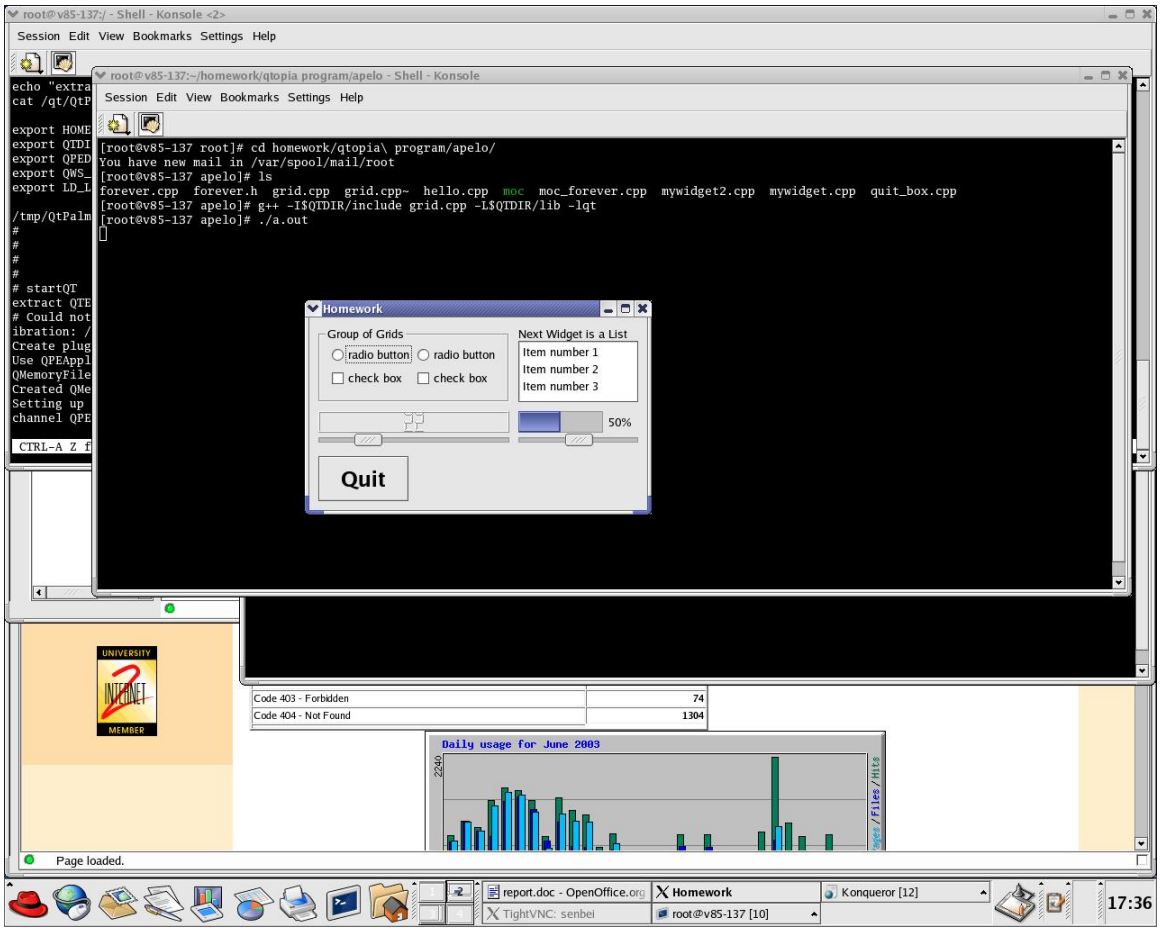
To understand quickly what this code is going to produce, lets compile it first for the X11 and then run it on our host PC. Previous to compilation, we should set up the environment variables for Qt/X11, this is it:

```
# export QTDIR=/opt/qt-2.3.2
# export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

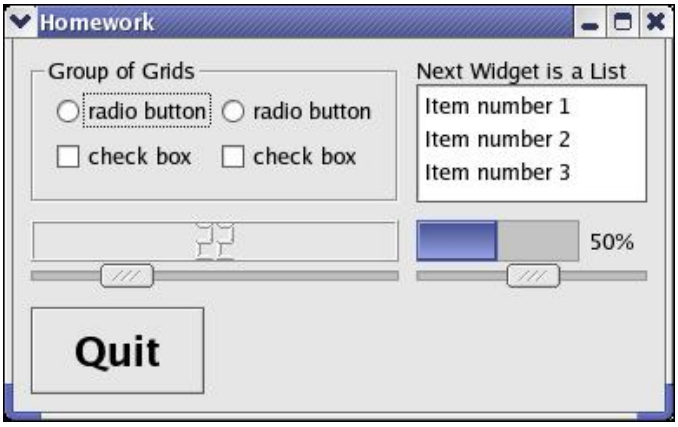
Now, knowing that the code file is called *grid.cpp* we can cross-compile using the *arm-ipaq-g++* compiler:

```
# g++ -I$QTDIR/include grid.cpp -L$QTDIR/lib -lqt
```

Luckily this will result in a file *a.out* (as we didn't specify the name of the output file), that we can run just with *./a.out*:



To distinguish better our application:



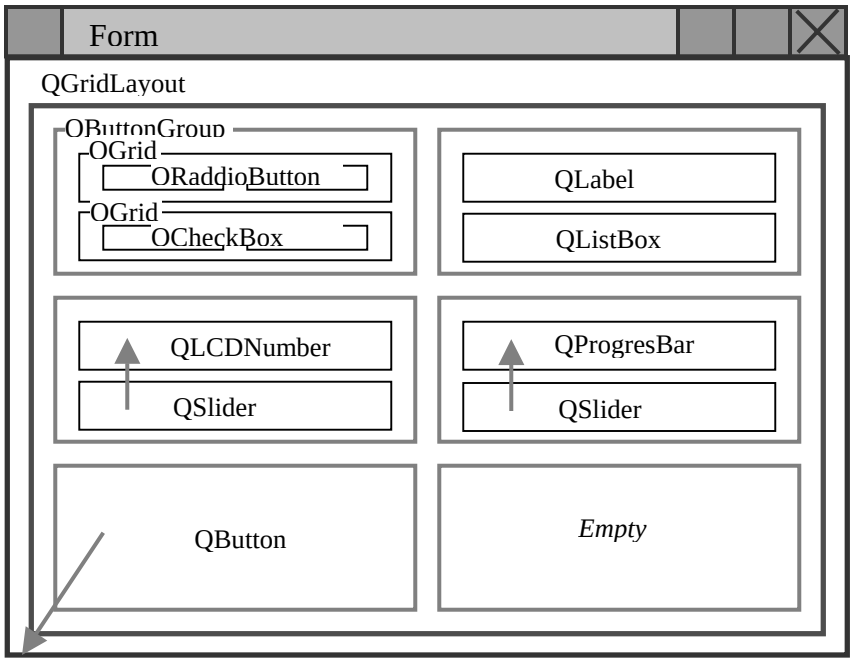
The 11 widgets used are:

- QButton
- QButtonGroup
- QGrid
- QgridLayout
- QRadioButton
- QCkeckBox
- QListBox
- QLabel
- QLCDNumber
- QProgressBar
- QSlider

5 classes are used: *Grids* (that builds the ButtonGroup with the radio buttons and the check boxes), *LCD* (which builds the LCD display and the slider and connects them), *Progress* (which does the same as LCD but with a progress bar), *LabelList* (that builds the label and the list) and *MyWidget* (which puts objects from the other classes in a GridLayout and adds a "Quit" button)

3 connections are done. First two are done between the sliders and the LCD number and progress bar. The last one connects the Quit button and the main form so it will close the program when pressed.

The *QGridLayout* arranges the other widgets in a nice way. An schema of the application layout is shown following. Arrows mean widget connections:



To compile it, we have to change the compiler. But Trolltech also offers a utility called *tmake* for compilation through different platforms. So we should download and uncompress the utility and configure it to compile for out platform:

```
# export QTDIR=/opt/qt-2.3.7
# export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
# export TMAKE=/opt/tmake1.8/lib/qws/linux-arm-g++
# export PATH=$PATH:/opt/tmake1.8/bin
```

And then compiles it:

```
# progen –o grid.pro grid.cpp
# tmake –o Makefile grid.pro
```

With this process we get a binary *grid*, which we need to copy to the ADS as explained before and execute it there the same way we did with the “hello world” example. If we did it correctly, the LCD panel will show this:

Well, the position in the screen is not as accurate as we would have liked, but it works!!. Job done.