



Running a QML HMI on an ARM11 without OpenGL

by [Burkhard Stubert](#) / 2016/02/05

Recently, I brought up Qt 5.5 on a Freescale i.MX35, which has an ARM11 CPU but no OpenGL support. Despite the missing OpenGL, I wanted to write the HMI with QML. The additional challenge was that the cross-compilation toolchain was 32-bit, but I wanted to use my standard 64-bit Ubuntu. I'll show in this post how to set up the 32-bit toolchain and rootfs on my 64-bit Ubuntu machine, how to configure and build Qt 5.5 from the sources, and how to run a hello-world application written in QML on the i.MX35.

The Challenges

I was recently tasked to bring up Qt 5.5 on [Wachendorff's display computer OPUS A3s](#) and build a QML demo for it. The OPUS A3s is based on the [Freescale i.MX35 system on chip](#) (SoC), which has an ARM11 CPU but no OpenGL acceleration.



My
first



Harvester HMI (QML) running on Wachendorff's display computer OPUS A3 (i.MX35, no OpenGL).

thought – and the first challenge – was that QML requires OpenGL to work and that the QPA plugin for software rendering is only available under the commercial Qt license. But then my personal Qt historian [Dario Freddi](#) reminded me that in the olden days QML (Qt 4.8) used to run just with the software renderer and that this feature still lives on in Qt 5.5 with QtQuick 1 and QtDeclarative. In short, QtQuick 2 requires OpenGL but QtQuick 1 does not. Phew, no Qt commercial license needed and lots of euros saved!

The second challenge was that Wachendorff provided a toolchain with 32-bit tools and a 32-bit Ubuntu 10.10. As Ubuntu 10.10 is not supported any more, I wanted to use at least the latest 64-bit Ubuntu with long-term support (v14.04) or best the very latest 64-bit Ubuntu (v15.10). Unfortunately, the 32-bit versions of the gcc compilers do not simply run on a 64-bit system. This can be solved by installing some libraries including the standard C and C++ libraries on the 64-bit Ubuntu development machine. I describe the installation of the toolchain and root file system in the section [Setting Up the Development Environment](#).

The third challenge is to create an “mkspec” or “make spec” for a new device, the i.MX35, and to configure and build Qt 5.5 from the GitHub sources for this device. The make specs of the Raspberry Pi (another ARM11 SoC) and of the i.MX53 (another Freescale SoC) will be a good reference for the make spec. Section [Configuring and Building Qt 5.5 from Sources](#) gives the details.

The fourth and final challenge is to run a sample QML app on the target device, the OPUS A3s. We must tell the app, which device files send the events for the function keys, the rotary knob and the touches and whether to rotate the touch coordinate system. I explain these things in section [Building and Running a QML App on the Target](#).

Although my target device is a Wachendorff OPUS A3s with an i.MX35 SoC, most of my explanations apply to every SoC. It does not make much of a difference whether we bring up Qt on a Freescale i.MX35/i.MX53/i.MX6, Texas Instruments Jacinto 5/6 or an Nvidia Tegra 2/3. Once we have understood the essential steps, it just small adjustments like the path to the C++ compiler, the sysroot path, the Qt modules that can or cannot be built for the target device or the environment variables needed for running the QML app.

Above is a photo of the home screen of the harvester HMI I have been building for the OPUS A3s (i.MX35). The needles of the two dials change every 50ms (20 times per second). All the other information – the diesel gauge, the gear info, the speed and the up to five warning indicators at the top – change once a second. Although the i.MX35 does not have OpenGL acceleration, the CPU load comes in at an average of only 27% – with peaks up to 35%. This is pretty good for a low-end device like the i.MX35.

Setting Up the Development Environment

Installing Packages Needed for Building Qt

My development machine is a 64-bit Ubuntu 15.10 virtual machine hosted by VmWare Fusion on my Macbook Pro. A well-equipped Windows laptop or PC with VmWare or VirtualBox would do as well. When starting with a fresh Ubuntu installation, many packages needed for development are missing.

A good starting point is to install all the packages needed to build Qt from sources. As Qt is quite a kraken, we are done more often than not. And – we need all the Qt dependencies anyway, because we will build Qt both for our Ubuntu machine and for the target hardware. The documentation page [Building Qt 5 from Git](#) lists the required packages neatly.

Ubuntu has a nifty command to install all the packages needed to build a package – *qt5-default* in our case:

```
$ sudo apt-get build-dep qt5-default
```

The next command pulls in tools needed for building software. The *perl* and *python* packages are typically already installed in a standard Ubuntu image. Then, they get updated by this command.

```
$ sudo apt-get install build-essential perl python git
```

Especially if you have been around Qt for some years, it is hard to believe that these two innocuous commands do all the heavy lifting. But they do! No more cumbersome figuring out which packages are needed.

If we want to build some special Qt modules like WebKit, WebEngine and Multimedia or if we need the latest version of some packages like XCB (the default QPA plugin for X11), we must install some more packages. I don't care about web things at the moment, but the other two could be interesting.

```
// For XCB:
$ sudo apt-get install "^libxcb.*" libx11-xcb-dev libglu1-mesa-dev
libxrender-dev libxi-dev

// For Multimedia:
$ sudo apt-get install libasound2-dev libgstreamer0.10-dev libgstreamer-
plugins-base0.10-dev
```

Installing the Toolchain and Root File System

Wachendorff, the manufacturer of the OPUS A3s display computer, gives us a tarball `linux-opusa3-2.0.3.tgz`. When we unpack this tarball, we find a tarball for the toolchain in the subdirectory `linux-opusa3-2.0.3/toolchain` and a tarball for the root file system in `linux-opusa3-2.0.3/rootfs`. These two tarballs are all we need.

First, we unpack the tarball for the toolchain.

```
$ cd /
$ sudo tar xzf /path/to/linux-opusa3-2.0.3/toolchain/tc_arm_gcc-4.6.2-
glibc-2.13-linaro-multilib-2011.12-1.tar.gz
```

This unpacks the toolchain into the directory `/opt/freescale`. As I work with different SoCs, different toolchains and even different versions of the same toolchain, I renamed this directory `/opt/imx35-gcc-4.6.2-glibc-2.13` and created a symbolic link `/opt/imx35` to it for convenience.

```
$ cd /opt
$ sudo mv freescale imx35-gcc-4.6.2-glibc-2.13
$ sudo ln -s imx35-gcc-4.6.2-glibc-2.13 imx35
```

From now on, the toolchain with the C++ compiler and the linker is located at `/opt/imx35`.

Second, we unpack the tarball for the root file system.

```
$ mkdir -p ~/Wachendorff/OpusA3
$ cd ~/Wachendorff/OpusA3
$ tar xzf /path/to/linux-opusa3-2.0.3/rootfs/rootfs-opusa3_2.0.3.tar.gz
```

This unpacks the root file system into the directory `~/Wachendorff/OpusA3/rootfs`. The root file system contains all the files that are needed to run our applications on the target device.

By the way, it does not matter where we install the toolchain and the root file system. We only need to pass these two locations to Qt's `configure` command.

Running a 32-bit GCC on a 64-bit Ubuntu Machine

The installed toolchain comes with 32-bit executables made for cross-compiling from an i386 development PC to an ARM11 target device. If we try to cross-compile a simple hello-world app

```
#include <iostream>

int main(int argc, char** argv) {
    std::cout << "Hello World!" << std::endl;
}
```

with the command

with the command

```
$ /path/to/arm-fsl-linux-gnueabi-g++ -o hello main.cpp
```

the answer will be something along the lines:

```
bash: /path/to/arm-fsl-linux-gnueabi-g++: No such file or directory
```

The error message is not really helpful. It took me some time to understand that I had run into a 32-bit versus 64-bit problem. Running `ldd` on the executable of the cross-compiler finally revealed that the libraries `libc.so.6` and `/lib/ld-linux.so.2` were missing. The second one gives away that the linker was looking for a 32-bit version, because the 64-bit version is `/lib64/ld-linux-x86-64.so.2`.

After a bit of “duckduckgoing”, I came across [the answer](#) on the AskUbuntu forum. We must install `libc6`, `libstdc++6` and `libncurses5` for the `i386` architecture. The following three commands do the trick.

```
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
$ sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

Now, cross-compiling our hello-world app works fine.

Configuring and Building Qt 5.5 from Sources

Getting the Qt Sources from Git

Getting the Qt source code from Git is described in the Qt documentation at [Building Qt 5 from Git – Getting the source code](#).

We clone the top-level Qt5 git repository in a directory of our choice, say `~/Qt`.

```
$ mkdir ~/Qt
$ cd ~/Qt
$ git clone https://code.qt.io/qt/qt5.git
```

We clone all the Qt submodules like `qtbases`, `qt3d` and `qtwebengine` by running the `init-repository` script. As we do not need `qtwebkit`, we skip cloning it by the option `--no-webkit`.

```
$ cd ~/Qt/qt5
$ perl init-repository --no-webkit
```

Now, the complete Qt5 repository is locally available in our Ubuntu machine. Qt 5.5.1 was the latest released version of Qt at the time of writing. So, we check out the tag “v5.5.1” into the new branch `qt-5.5.1`.

```
$ cd ~/Qt/qt5
$ git checkout -b qt-5.5.1 v5.5.1
```

In `~/Qt/qt5`, we have the versions of the Qt sources that went into the Qt release v5.5.1.

Instead of cloning the Qt sources from the git repository, we could download the source tarball for Qt 5.5.1 [from the Download page](#). However, I have found out over the years that nearly every project needs some changes, extensions or bug fixes to Qt. I prefer to have these modifications under version control, which is easy with the git repository.

Defining the Make Spec and Configure Command Iteratively

The keyword in the section title is “iteratively”. I have never been able to come up with a make spec and configure command first time right – despite having plenty of exercise over the years. Creating a make spec and figuring out the write options for the configure command are an iterative process.

First Iteration

We find the make specs for several common SoCs in the directory `~/Qt/qt5/qtbase/mkspecs/devices`. There is no make spec for the i.MX35 though. The closest SoC is the i.MX53, the big brother of the i.MX35. Its make spec `~/Qt/qt5/qtbase/mkspecs/devices/linux-imx53qsb-g++/qmake.conf` looks as follows.

```
# qmake.conf for i.MX53
include(../common/linux_device_pre.conf)

QMAKE_LIBS_EGL           += -lEGL
QMAKE_LIBS_OPENGL_ES2    += -lGLESv2 -lEGL
QMAKE_LIBS_OPENVG        += -lOpenVG -lEGL

IMX5_CFLAGS               = -march=armv7-a -mfpu=neon -DLINUX=1 -
DEGL_API_FB=1 -Wno-psabi
QMAKE_CFLAGS              += $IMX5_CFLAGS
QMAKE_CXXFLAGS             += $IMX5_CFLAGS

include(../common/linux_arm_device_post.conf)

load(qt_config)
```

As the i.MX35 does not support OpenGL, we can remove the three lines about `QMAKE_LIBS_*`.

The i.MX35 is part of the ARM11 family. A look at the [list of ARM microarchitectures](#) reveals that the ARM11 family uses the ARMv6 architecture. In contrast to the i.MX35, the i.MX53 with a Cortex-A8 core uses the ARMv7-A architecture. Hence, the `CFLAGS` for the i.MX35 differ from those of the i.MX53.

At this point, we are looking for a sample make spec based on the ARMv6 architecture. We either

At this point, we are looking for a sample make spec based on the ARMv6 architecture. We either know that the Raspberry Pi is part of the ARM11 family as well or we just search all make specs for “armv6”.

```
$ cd ~/Qt/qt5/qtbase/mkspecs/devices
$ find . -name "qmake.conf" | xargs grep -i armv6
./linux-rasp-pi-g++/qmake.conf:                                -march=armv6zk
\
```

The relevant lines of the Raspberry Pi’s make spec read as follow:

```
QMAKE_CFLAGS += \
    -marm -march=armv6zk -mtune=arm1176jzf-s \
    -mfpv=vfp -mabi=aapcs-linux
```

We figure out the correct values for the above machine options by checking out [the datasheet of the i.MX35](#). Section “2.4 ARM11 Microprocessor Core” gives us the needed information. The i.MX35 uses an ARM1136JF-S core (hence, `-mtune=arm1136jf-s`), which has a vector floating point co-processor (hence, `-mfpv=vfp`). Looking up the ARM1136JF-S core in the [list of ARM microarchitectures](#) tells us that its architecture is neither ARMv6Z (not TrustZone support) nor ARMv6K (no multi-core) but simply ARMv6 (hence, `-march=armv6`). Using the “ARM Architecture Procedure Call Standard” (AAPCS) for Linux as the ABI sounds reasonable (hence, `-mabi=aapcs-linux`).

We can check the admissible values of the machine options on the page [“ARM Options”](#) of the GCC documentation. All is fine!

We have just finished the first version of *qmake.conf* of our make spec for the i.MX35.

```
# qmake.conf for i.MX35
include(../common/linux_device_pre.conf)

IMX35_CFLAGS += \
```

```

IMX35_CFLAGS
    -marm \
    -mfpv=vp \
    -mtune=arm1136jf-s \
    -march=armv6 \
    -mabi=aapcs-linux
QMAKE_CFLAGS += $IMX35_CFLAGS
QMAKE_CXXFLAGS += $IMX35_CFLAGS

include(../common/linux_arm_device_post.conf)
load(qt_config)

```

Besides the *qmake.conf* file, the make spec contains another file, *qplatformdefs.h*. This one is easy as it is the same for all Linux devices. So, we can simply copy it, say, from the make spec of the i.MX53. It contains one line:

```
#include "../linux-g++/qplatformdefs.h"
```

All we need now is a configure command. I'll show you my first version and show you how I came up with all the options.

```

../qt5/configure -v -opensource -confirm-license \
    -release -prefix /opt/qt-5.5.1-imx35 \
    -device linux-imx35-g++ \
    -sysroot /home/burkhard/Wachendorff/OpusA3/rootfs \
    -device-option CROSS_COMPILE=/opt/imx35/usr/local/gcc-4.6.2-
glibc-2.13-linaro-multilib-2011.12/fsl-linaro-toolchain/bin/arm-fsl-
linux-gnueabi- \
    -linuxfb -qpa linuxfb -no-eglfs -no-directfb -no-kms -no-opengl \
    -no-dbus -no-largefile -no-qml-debug -nomake tests -nomake examples
-no-gstreamer \
    -skip qtenginio -skip qtlocation -skip qtmultimedia -skip qtpim -
skip qtwayland \
    -skip qtwebchannel -skip qtwebengine -skip qtwebkit -skip
qtwebsockets

```

I always run configure in verbose mode (`-v`) to get more information why configure failed. I use Qt under LGPLv3 (`-opensource`). I don't want configure to stop and ask me to configure the license (`-confirm-license`).

I typically start with a release build (`-release`), because it builds much faster than a debug build and it shows how fast our HMI runs on the target device. Building Qt as fast as possible is paramount, because the Qt build is likely to fail. If the build fails, we want it to fail fast. The option `-prefix /opt/qt-5.5.1-imx35` specifies that Qt will be installed in the directory `/opt/qt-5.5.1-imx35` on the target device.

The option `-device linux-imx35-g++` tells the configure command to use the make spec we just created.

The options `-sysroot` and `-device-option` tell the configure command and later qmake commands where to find the root file system and the toolchain, respectively. The value of `-sysroot` is the directory, `~/Wachendorff/OpusA3/rootfs`, where we unpacked the tarball with the root file system. The option `-device-option` defines the environment variable `CROSS_COMPILE` as the prefix of the path to tools like gcc, g++, objcopy and strip in the toolchain. These tools are located in the directory

```
/opt/imx35/usr/local/gcc-4.6.2-glibc-2.13-linaro-multilib-2011.12/fsl-
linaro-toolchain/bin/
```

and start with the prefix `arm-fsl-linux-gnueabi-`. For example,

```
/opt/imx35/usr/local/gcc-4.6.2-glibc-2.13-linaro-multilib-2011.12/fsl-
linaro-toolchain/bin/arm-fsl-linux-gnueabi-g++
```

is the C++ compiler for cross-compiling Qt. The environment variable `CROSS_COMPILE` is used in the configuration file `linux_device_pre.conf` included by our make spec file. It used to define qmake's tool variables:

```
QMAKE_CC          = ${CROSS_COMPILE}gcc
QMAKE_CXX         = ${CROSS_COMPILE}g++
# More ...
```

We should never hard-code the root-fs path or the cross-compile path prefix into any configuration file or any Qt project file. As true believers of the DRY principle (“don’t repeat yourself”), we should define these paths only once – in the options of the configure command. If we need to define more environment variables for the configuration of Qt, we can add another `-device-option` option with the definition of another environment variable.

The options

```
-linuxfb -qpa linuxfb -no-eglfs -no-directfb -no-kms -no-opengl
```

specify that we want to use the Linux framebuffer (`-linuxfb`) as the window-system backend (a.k.a. QPA plugin) and that the Linux framebuffer is the default QPA plugin (`-qpa linuxfb`). Defining the default QPA plugin saves from passing the command-line option `-qpa linuxfb` when we start our application. We do not build the QPA plugins `eglfs`, `directfb` and `kms` and turn off OpenGL support, as none of these is supported by our target device.

The final three lines of options

```
-no-dbus -no-largefile -no-qml-debug -nomake tests -nomake examples
-no-gstreamer \
    -skip qtenginio -skip qtlocation -skip qtmultimedia -skip qtpim -
skip qtwayland \
    -skip qtwebchannel -skip qtwebengine -skip qtwebkit -skip
qtwebsockets
```

excludes every feature and module that we don’t need for the initial run of our application. Remember that we want our build to fail fast – and fail it will. Once we have a working build and a

running sample app, we can add features and modules as we need them.

Now we are finally ready to run the configure command. We will perform a shadow build outside the Qt sources, because it is pretty likely that we will build multiple versions of Qt (for example, a 64-bit version for our Ubuntu machine or a debug version for the i.MX35).

```
// Create directory for shadow build
$ cd ~/Qt
$ mkdir build-qt-5.5.1-imx35
$ cd build-qt-5.5.1-imx35
$ ../qt5/configure -v -opensource -confirm-license \
    -release -prefix /opt/qt-5.5.1-imx35 \
    -device linux-imx35-g++ \
    -sysroot /home/burkhard/Wachendorff/OpusA3/rootfs \
    -device-option CROSS_COMPILE=/opt/imx35/usr/local/gcc-4.6.2-
glibc-2.13-linaro-multilib-2011.12/fsl-linaro-toolchain/bin/arm-fsl-
linux-gnueabi- \
    -linuxfb -qpa linuxfb -no-eglfs -no-directfb -no-kms -no-opengl \
    -no-dbus -no-largefile -no-qml-debug -nomake tests -nomake examples
-no-gstreamer \
    -skip qtenginio -skip qtlocation -skip qtmultimedia -skip qtpim -
skip qtwayland \
    -skip qtwebchannel -skip qtwebengine -skip qtwebkit -skip
qtwebsockets
```

Bad news! As expected the configuration command fails. All feature tests fail because the C++ compiler cannot find system headers. It cannot even find a header like *stdio.h*. Something is fundamentally wrong. Let us have a closer look at the first feature that fails.

```
Determining architecture... ()
/opt/imx35/.../bin/arm-fsl-linux-gnueabi-g++ \
    -c -pipe -marm -mfpv=vfp -mtune=arm1136jf-s -march=armv6 -
mabi=aapcs-linux \
    -mfloat-abi=softfp --sysroot=/home/burkhard/Wachendorff/OpusA3/
rootfs \
    -g -Wall -W -fPIC -I../../../../qt5/qtbase/config.tests/arch -I. \
    I../../../../qt5/qtbase/mkspecs/devices/linux imx35 g++ o arch o \
```

```

-I../../../../qt5/qtbase/mkspecs/devices/linux-imx35-g++ -o arch.o \
../../../../qt5/qtbase/config.tests/arch/arch.cpp
../../../../qt5/qtbase/config.tests/arch/arch.cpp:37:19: fatal error:
stdio.h: No such file or directory
compilation terminated.
Makefile:207: recipe for target 'arch.o' failed
make: *** [arch.o] Error 1
Unable to determine architecture!

```

Searching for the file *stdio.h* in the root file system yields nothing. Searching for it in the toolchain yields three hits. The relevant hit is

```

/opt/imx35/.../arm-fsl-linux-gnueabi/multi-libs/default/usr/include/
stdio.h

```

Running the command

```

/opt/imx35/.../bin/arm-fsl-linux-gnueabi-g++ -print-sysroot

```

yields

```

/opt/imx35/.../arm-fsl-linux-gnueabi/multi-libs/default/

```

as the standard sysroot of the compiler. By passing a different sysroot to the compiler, we override the standard sysroot. The compiler looks for system headers in the new sysroot. And we know that it cannot find these headers there. Let us test our theory and run just the failing test for determining the architecture again – but this time without the sysroot option.

```

$ cd ~/Qt/build-qt-5.5.1-imx35/qtbase/config.tests/arch
$ /opt/imx35/.../bin/arm-fsl-linux-gnueabi-g++ \

```

```

-c -pipe -marm -mfpv=vfp -mtune=arm1136jfs -march=armv6 -
mabi=aapcs-linux \
-mfloat-abi=softfp \
-g -Wall -W -fPIC -I../../../../qt5/qtbase/config.tests/arch -I. \
-I../../../../qt5/qtbase/mkspecs/devices/linux-imx35-g++ -o arch.o \
../../../../qt5/qtbase/config.tests/arch/arch.cpp

```

Now the compile command works. Tweaking the command line of a feature test in isolation is a common trick to solve the configuration problems. So, add it to your bag of tricks.

We must tell configure somehow that it should not use our sysroot but the compiler's sysroot. We apply wishful thinking and hope that configure provides some option for that already.

```

$ cd ~/Qt/build-qt-5.5.1-imx35
$ ../qt5/configure -help | grep sysroot
-sysroot <dir> ..... Sets <dir> as the target compiler's and
qmake's sysroot and also sets pkg-config paths.
-no-gcc-sysroot ..... When using -sysroot, it disables the passing
of --sysroot to the compiler

```

And bingo! It worked. The second option `-no-gcc-sysroot` is exactly what we wished for.

Second Iteration

We add the option `-no-gcc-sysroot` to the configure command from the first iteration and try our luck again.

```

$ cd ~/Qt/build-qt-5.5.1-imx35
$ ../qt5/configure -v -opensource -confirm-license \
-release -prefix /opt/qt-5.5.1-imx35 \
-device linux-imx35-g++ \
-no-gcc-sysroot -sysroot /home/burkhard/Wachendorff/OpusA3/rootfs \
-device-option CROSS_COMPILE=/opt/imx35/usr/local/gcc-4.6.2-
glibc-2.13-linaro-multilib-2011.12/fsl-linaro-toolchain/bin/arm-fsl-

```

```
linux-gnueabi- \
  -linuxfb -qpa linuxfb -no-eglfs -no-directfb -no-kms -no-opengl \
  -no-dbus -no-largefile -no-qml-debug -nomake tests -nomake examples
-no-gstreamer \
  -skip qtenginio -skip qtlocation -skip qtmultimedia -skip qtpim -
skip qtwayland \
  -skip qtwebchannel -skip qtwebengine -skip qtwebkit -skip
qtwebsockets
```

The configure command runs to the end successfully. On first glance, the summary looks pretty reasonable. LinuxFB is listed as the only QPA backend. OpenGL is disabled. Evdev, which handles touch, keyboard and rotary-encoder output, is enabled. The other features are disabled or enabled as specified in the configure command.

On second glance, there are few minor things that need at least an explanation and possibly a fix. Here are the suspicious lines.

```
Image formats:
  GIF ..... yes (plugin, using bundled copy)
  JPEG ..... yes (plugin, using bundled copy)
  PNG ..... yes (in QtGui, using bundled copy)
  tslib ..... no
  zlib ..... yes (bundled copy)
```

Since the arrival of the Linux v3.x kernels, tslib is not needed any more to handle touch input. Touch input is now handled by evdev. Hence, tslib need not be built and the “no” is OK.

It is a bit strange that the Qt build wants to use the copies of the jpeg, png and zlib libraries bundled with Qt. It should use the system libraries, which are available on every proper Linux system. A quick search through our root file system shows that the Linux system provided by Wachendorff is proper. The headers and libraries exist.

With the `-no-gcc-sysroot` option, we told g++ to ignore our sysroot, which points to our root file system. Obviously, the jpeg, png and zlib headers and libraries are not contained in the toolchain. As we do not want these libraries twice on the target system – once provided by the

system and once by Qt, we make the configure command pick up the system versions. We do this by adding include and library search directories to our make spec. The modified *qmake.conf* looks as follows.

```
include(../common/linux_device_pre.conf)

IMX35_CFLAGS += \
    -marm \
    -mfpv=vfp \
    -mtune=arm1136jf-s \
    -march=armv6 \
    -mabi=aapcs-linux

QMAKE_INCDIR += ${QT_SYSDIR}/usr/include
QMAKE_LIBDIR += ${QT_SYSDIR}/usr/lib
QMAKE_CFLAGS += $IMX35_CFLAGS
QMAKE_CXXFLAGS += $IMX35_CFLAGS

include(../common/linux_arm_device_post.conf)
load(qt_config)
```

``${QT_SYSDIR}`` refers to the value of the `-sysroot` option defined in the configure command. It is time for the third iteration.

Third Iteration

We run the same configure command as for the second iteration. The command will use our modified *qmake.conf* file.

```
$ cd ~/Qt/build-qt-5.5.1-imx35
$ ../qt5/configure -v -opensource -confirm-license \
    -release -prefix /opt/qt-5.5.1-imx35 \
    -device linux-imx35-g++ \
    -no-gcc-sysroot -sysroot /home/burkhard/Wachendorff/OpusA3/rootfs \
    -device-option CROSS_COMPILE=/opt/imx35/usr/local/gcc-4.6.2-
glibc-2.13-linaro-multilib-2011.12/fsl-linaro-toolchain/bin/arm-fsl-
```

```
glibc 2.13 linaro multilib 2011.12/fsl linaro toolchain/bin/arm fsl
linux-gnueabi- \
    -linuxfb -qpa linuxfb -no-eglfs -no-directfb -no-kms -no-opengl \
    -no-dbus -no-largefile -no-qml-debug -nomake tests -nomake examples
-no-gstreamer \
    -skip qtenginio -skip qtlocation -skip qtmultimedia -skip qtpim -
skip qtwayland \
    -skip qtwebchannel -skip qtwebengine -skip qtwebkit -skip
qtwebsockets
```

The summary looks better now, but not yet perfect. The critical lines from the second iteration have changed as follows.

```
Image formats:
GIF ..... yes (plugin, using bundled copy)
JPEG ..... yes (plugin, using system library)
PNG ..... yes (in QtGui, using bundled copy)
tslib ..... yes
zlib ..... yes (system library)
```

The enabling of tslib is collateral damage from adding the include directory in our root file system. If we want to disable tslib and gif, we can add the options `-no-tslib -no-gif` to the configure command.

The png library remains stubborn. A quick look into the verbose output of configure shows that the linker complains:

```
$ /opt/.../bin/arm-fsl-linux-gnueabi-g++ -mfloat-abi=softfp -Wl,-O1 -o
libpng libpng.o -L/home/burkhard/Wachendorff/OpusA3/rootfs/usr/lib -lpng
/opt/.../arm-fsl-linux-gnueabi/bin/ld: warning: libz.so.1, needed by /
home/.../OpusA3/rootfs/usr/lib/libpng.so, not found (try using -rpath or
-rpath-link)
```

The executable `libpng` links directly against `libpng.so`, which links directly against `libz 1`. Hence, `libpng` links indirectly or implicitly against `libz 1`. We could fix this

`libz.so.1`. Hence, `libpng` links indirectly or implicitly against `libz.so.1`. We could fix this problem quick and dirty by adding `-lz` at the end of the linker command (after `-lpng`). This would only solve the linking problem for `libz`, but not for any other library linked against implicitly.

Fortunately, the error message gives us a hint how to solve this problem generally: “try using `-rpath` or `-rpath-link`”. Whereas the option `-rpath` is for finding libraries at runtime, the option `-rpath-link` is for finding libraries at linktime. We insert the line

```
QMAKE_LFLAGS += -Wl,-rpath-link,$[QT_SYSROOT]/usr/lib
```

after the definition of `QMAKE_LIBDIR` into our `qmake.conf` file. It is time for the fourth iteration.

Fourth Iteration

We run configure with gif and tslib disabled on the modified make spec.

```
$ cd ~/Qt/build-qt-5.5.1-imx35
$ ../qt5/configure -v -opensource -confirm-license \
  -release -prefix /opt/qt-5.5.1-imx35 \
  -device linux-imx35-g++ \
  -no-gcc-sysroot -sysroot /home/burkhard/Wachendorff/OpusA3/rootfs \
  -device-option CROSS_COMPILE=/opt/imx35/usr/local/gcc-4.6.2-
glibc-2.13-linaro-multilib-2011.12/fsl-linaro-toolchain/bin/arm-fsl-
linux-gnueabi- \
  -linuxfb -qpa linuxfb -no-eglfs -no-directfb -no-kms -no-opengl \
  -no-gif -no-tslib -no-dbus -no-largefile -no-qml-debug -nomake tests
-nomake examples -no-gstreamer \
  -skip qtenginio -skip qtlocation -skip qtmultimedia -skip qtpim -
skip qtwayland \
  -skip qtwebchannel -skip qtwebengine -skip qtwebkit -skip
qtwebsockets
```

Finally, the configure summary is perfect. The features gif and tslib are disabled and png uses the

system library. It is high time to kick off the build.

```
$ cd ~/Qt/build-qt-5.5.1-imx35  
$ make -j4
```

Surprise! The build succeeded. It is not unusual for the build to fail, because some headers or libraries cannot be found or because we forgot to skip some modules or to disable some features. Then we must fix the problem either in the make spec or in the configure command and re-configure and re-build Qt again.

Now we are good to install Qt.

```
$ make install
```

Qt is installed into `<sysroot>/<prefix>`, that is, into `~/Wachendorff/OpusA3/rootfs/opt/qt-5.5.1-imx35`.

You can download the complete make spec from [here](#). Just unpack the ZIP archive in the directory `~/Qt/qt5/qtbase/mkspecs/devices`.

Downloads

- [linux-imx35-g++](#) – Make spec for the i.MX35. Unpack ZIP archive in directory `~/Qt/qt5/qtbase/mkspecs/devices`.
- [HelloWorld](#) – Project of HelloWorld app. Unpack ZIP archive in directory of your choice. Open project file `/path/to/HelloWorld/HelloWorld.pro` into QtCreator set up for i.MX35.

Tags: 32-BIT VS 64-BIT ARM11 CROSS-COMPILE I.MX35 OPUS A3 QML QT

WACHENDORFF

Leave a Reply

Your email address will not be published. Required fields are marked *

Name

Email

Website

Comment *

☐ By ticking this checkbox, you consent to this privacy policy. *

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.

Post Comment





 [Contact Me](#)

Better Built By Burkhard

My bi-weekly musings about building smart HMIs for embedded devices: system architecture, team topologies, continuous delivery, licensing, solo business.

By Burkhard Stubert

Subscribe

By subscribing you agree to [Substack's Terms of Use](#), [our Privacy Policy](#) and [our Information collection notice](#)

 substack

Recent Posts

EU CRA: Essential Requirements Related to Product Properties

Embedded Devices Covered by EU Cyber Resilience Act (CRA)

Embedded Devices Covered by EU Cyber Resilience Act (CRA)

Updating U-Boot with an A/B Strategy

A Yocto Recipe for Qt Applications Built with CMake

Ports-and-Adapters Architecture: The Pattern

Categories

Select Category

[Imprint](#)

[Privacy Policy](#)

[Terms of Use](#)

Neve | Powered by WordPress