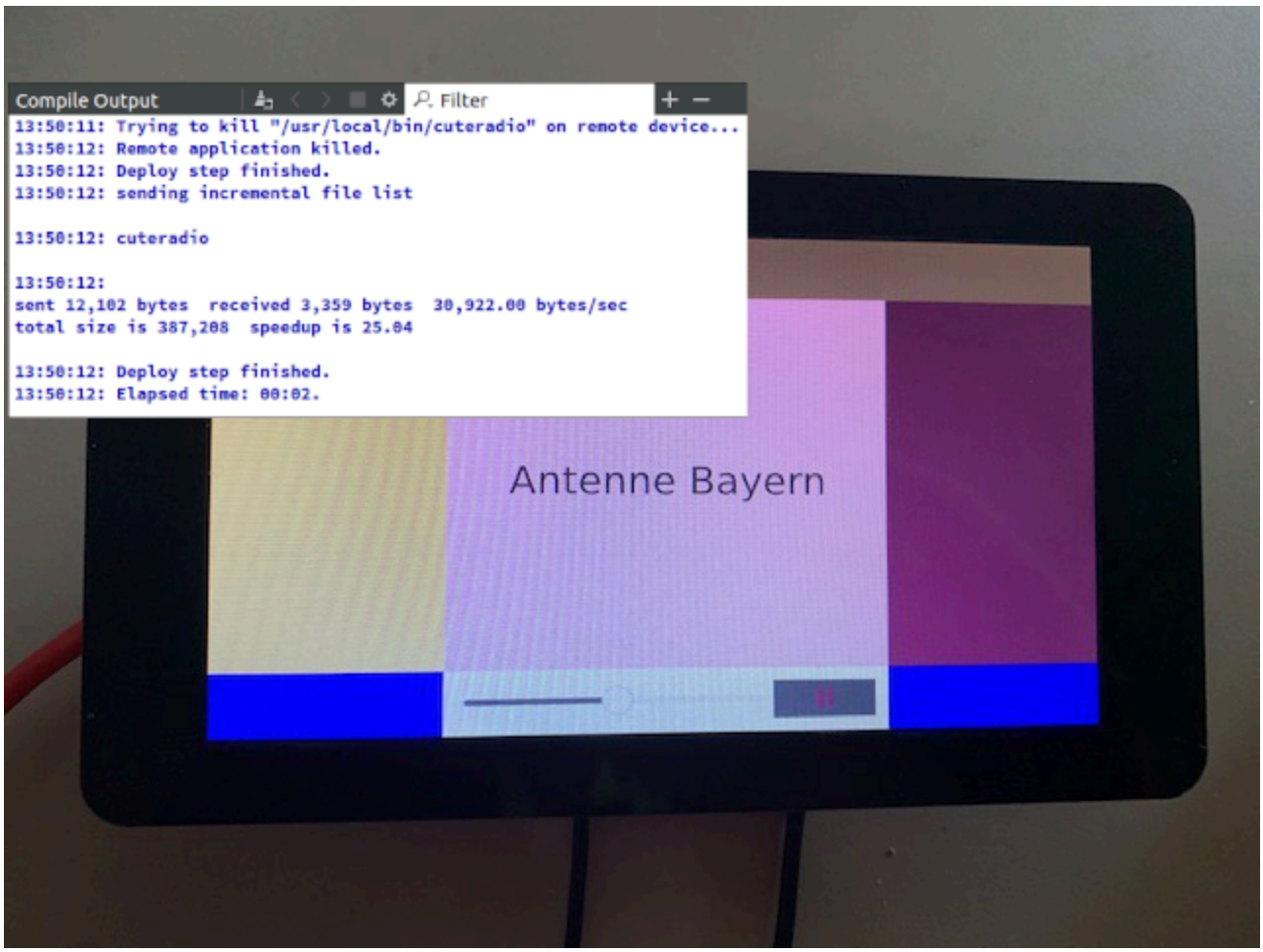
Qt Embedded Systems – Part 2: Building a Qt SDK with Yocto

by Burkhard Stubert / 2020/06/19 / 7 Comments



We want to develop the Internet radio application for the Raspberry Pi in the same way as for a PC. We change the source code in QtCreator and run the application. QtCreator cross-builds the application on the PC for the Raspberry Pi, deploys it with SSH to the Pi and runs it on the Pi. We

need a Qt SDK for this to work. In addition to the target libraries from the Linux image, the Qt SDK contains the library headers, a cross-compiler, a cross-linker, a cross-debugger and more.

Prerequisites

Linux Image

If you want to follow along with building the Qt SDK, you must build a Linux image as described in Part 1 of this series.

If you followed along Part 1 and built an image before 14 June 2020, you will have to update the layer repositories with `repo sync` (see here) and the build of the Linux image with `bitbake cuteradio-image` (see here). Two packages were missing from this Linux image: *rsync* and *sftp-server*. QtCreator needs these packages to deploy the radio application from the development PC to the Raspberry Pi. You must burn the updated Linux image on an SD card and plug the SD card into the Raspberry Pi – as described here.

Setup for Application Development

As the Docker container for building the Linux image and the SDK run on Ubuntu 18.04, our development PC must run on Ubuntu 18.04 or newer. For example, I run Ubuntu 18.04 in a virtual machine on a Macbook Pro.

We install QtCreator 4.11 or newer and CMake 3.16 or newer on the development PC. Using these versions makes QtCreator deploy all the files specified by CMake's `install` functions. If we use either an older QtCreator version or an older CMake version, we must use the workaround with *QtCreatorDeployment.txt* described here.

We download the Qt online installer to install a Desktop Qt version including QtCreator (e.g., Qt 5.14 with QtCreator 4.11). We download the installer for CMake 3.17 from here and run the self-extracting tarball.

Application Sources

We install the sources of the Internet radio application into the working directory.

```
$ cd /public/Work
$ git clone https://github.com/bstubert/cuteradio-apps.git
Cloning into 'cuteradio-apps'
...
```

Building the Qt SDK

We first enter the Docker container and then the Yocto build environment that we set up in Part 1.

```
$ cd /public/Work
$ ./dr-yocto/run-shell.sh 18.04

# cd ./cuteradio-thud
# source ./sources/poky/oe-init-build-env build-rpi3
# pwd
/public/Work/cuteradio-thud/build-rpi3
```

In a non-Qt world, we would call

```
// Do NOT call this!
# bitbake -c populate_sdk cuteradio-image
```

to build an SDK. The task `populate_sdk` builds the cross-compilation toolchain and packages the toolchain, most of the root file system, the header files and some additional files into the SDK, a self-extracting tarball. This task also cross-builds Qt tools like `qmake`, `moc` and `rcc`.

As we want to build applications on the host PC and not on the target device, we must make Yocto build these tools for the host PC. The recipe *meta-toolchain-qt5.bb* does exactly this in addition to executing the task `populate_sdk` internally.

We build the Qt SDK with the following command.

```
# bitbake meta-toolchain-qt5

WARNING: Layer cuteradio should set LAYERSERIES_COMPAT_cuteradio in its
conf/layer.conf file to list the core layer names it is compatible with.
Loading cache: 100% |#####| Time: 0:00:00
Loaded 2936 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "1.40.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "raspberrypi3"
DISTRO               = "poky"
DISTRO_VERSION       = "2.6.4"
TUNE_FEATURES        = "arm armv7ve vfp thumb neon vfpv4 callconvention-
hard cortexa7"
TARGET_FPU           = "hard"
meta
meta-poky            = "HEAD:958427e9d2ee7276887f2b02ba85cf0996dea553"
meta-oe
meta-python          = "HEAD:446bd615fd7cb9bc7a159fe5c2019ed08d1a7a93"
meta-raspberrypi     = "HEAD:4e5be97d75668804694412f9b86e9291edb38b9d"
meta-qt5             = "HEAD:e6e464c9ed9266ce46452f953c1bdcb0e7b2d95f"
meta-cuteradio       = "thud:3376391188a4de4e7c29a8299e905b0e5b15960a"

Initialising tasks: 100% |#####| Time: 0:00:03
Sstate summary: Wanted 881 Found 153 Missed 728 Current 923 (17% match,
59% complete)

NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 4888 tasks of which 3773 didn't need to be
rerun and all succeeded.
```

We find the installer of the Qt SDK in the directory *tmp/deploy/sdk*.

```
# cd tmp/deploy/sdk
# ls
poky-glibc-x86_64-meta-toolchain-qt5-cortexa7t2hf-neon-vfpv4-toolchain-
2.6.4.sh
...
```

Installing and Using the Qt SDK

As the people responsible for building the image and the SDK, we give the installer to the application developers. As application developers, we install the SDK on a Linux computer with Ubuntu 18.04 or newer by executing its installer.

```
$ /path/to/poky-glibc-x86_64-meta-toolchain-qt5-cortexa7t2hf-neon-vfpv4-
toolchain-2.6.4.sh

Poky (Yocto Project Reference Distro) SDK installer version 2.6.4
==========================================================
Enter target directory for SDK (default: /opt/poky/2.6.4):
/public/Work/qt-sdk-thud
You are about to install the SDK to "/public/Work/qt-sdk-thud".
Proceed[Y/n]? y
Extracting SDK.....................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to
source the environment setup script e.g.
 $ . /public/Work/qt-sdk-thud/environment-setup-cortexa7t2hf-neon-vfpv4-
poky-linux-gnueabi
```

The installer asks in which directory to install the SDK. We must ensure that we have write permission to the installation directory. If the default directory */opt/poky/2.6.4* is OK (Thud is Yocto 2.6), we hit *Return* twice and are done. Otherwise, we enter a directory of our choice (e.g., */public/Work/qt-sdk-thud*) and hit *Return* twice.

Before we can perform any builds with the Qt SDK in our current shell session, we set up the build environment for the application. The last message of the SDK installation script tells us what to do.

```
$ . /public/Work/qt-sdk-thud/environment-setup-cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi
```

Don't forget the dot at the beginning of the first command, which sources the script. We'll work in this *SDK shell* for the rest of this post. The environment setup script defines a couple of environment variables. Here are the important ones.

```
OECORE_TARGET_SYSROOT=/public/Work/qt-sdk-thud/sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi

OECORE_NATIVE_SYSROOT=/public/Work/qt-sdk-thud/sysroots/x86_64-pokysdk-linux
PATH includes
    $OECORE_NATIVE_SYSROOT/usr/bin
    $OECORE_NATIVE_SYSROOT/usr/sbin
    $OECORE_NATIVE_SYSROOT/bin
    $OECORE_NATIVE_SYSROOT/sbin
    $OECORE_NATIVE_SYSROOT/usr/bin/arm-poky-linux-gnueabi
    ...
OE_QMAKE_PATH_HOST_BINS=$OECORE_NATIVE_SYSROOT/usr/bin

OE_CMAKE_TOOLCHAIN_FILE=$OECORE_NATIVE_SYSROOT/usr/share/cmake/OEToolchainConfig.cmake

CXX=arm-poky-linux-gnueabi-g++  -march=armv7ve -mthumb -mfpu=neon-vfpv4 -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=$OECORE_TARGET_SYSROOT
CXXFLAGS= -O2 -pipe -g -feliminate-unused-debug-types

LD=arm-poky-linux-gnueabi-ld  --sysroot=$OECORE_TARGET_SYSROOT
LDFLAGS=-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed
```

The directory *$OECORE_TARGET_SYSROOT* contains all the files from the Linux image (the root filesystem) plus header files. All binary files like executables and libraries are in ARM format for use

on the Raspberry Pi.

The directory `$OECORE_NATIVE_SYSROOT` contains the toolchain (compilers, linker, etc.) and other files needed for cross-building. All binary files are in Intel format, because they will be used on the Intel-based development PC. Hence, the usual directories for binary files are added to `PATH`.

The subdirectory *usr/bin* holds executables like *qmake*, *lupdate*, *cmake* and *make*. This subdirectory is referenced so often in Yocto recipes that it gets its own environment variable `OE_QMAKE_PATH_HOST_BINS`. The subdirectory *usr/bin/arm-poky-linux-gnueabi* holds the toolchain including the binaries for *g++*, *gcc*, *ldd*, *ar*, *gdb*, *objdump* and *strip*. All these binaries are prefixed with

```
CROSS_COMPILE=arm-poky-linux-gnueabi-
```

The toolchain file `$OE_CMAKE_TOOLCHAIN_FILE` tells CMake which compiler and linker flags to use, where to find CMake modules and the target root filesystem, and which processor is used. It translates the SDK environment variable into CMake variables. In an ideal world, the following CMake call would generate a Makefile for the radio application.

```
$ mkdir /public/Work/build
$ cd /public/World/build
$ cmake -DCMAKE_TOOLCHAIN_FILE=$OE_CMAKE_TOOLCHAIN_FILE ../cuteradio-apps
```

Well, in reality, it doesn't work out of the box. We'll work out later how to fix it.

The environment setup script also sets standard variables like `CXX`, `CXXFLAGS`, `LD`, `LDFLAGS` and some more.

Setting Up QtCreator

Starting QtCreator

We start QtCreator from the SDK shell. For example, I have Qt 5.14 on my PC for application development and start the included QtCreator.

```
$ ~/Qt/Qt5.14.0/Tools/QtCreator/bin/qtcreator &
```

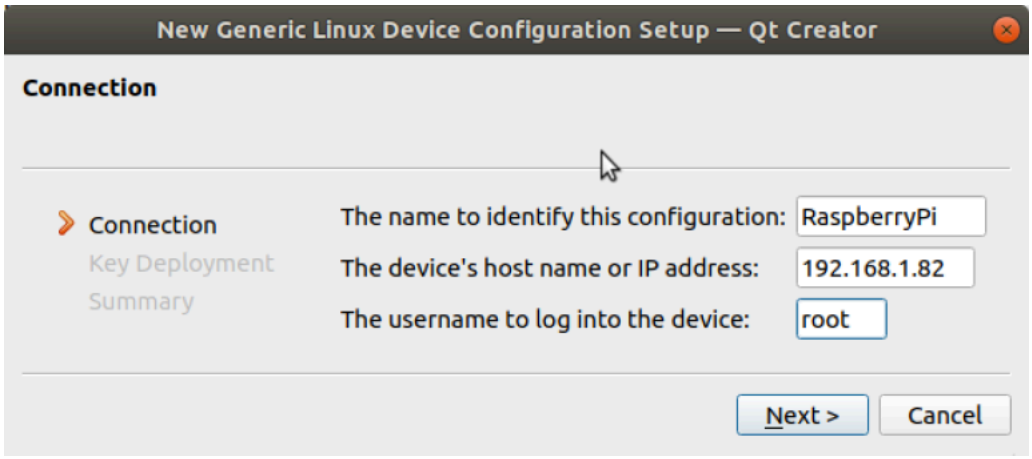
Accessing the Target Device via SSH

QtCreator uses `rsync` or `sftp` over SSH to copy files from the development PC to the target system and `ssh` to execute or terminate the application on the Raspberry Pi. The Raspberry Pi runs a DropBear SSH server, which is more lightweight than the OpenSSH server.

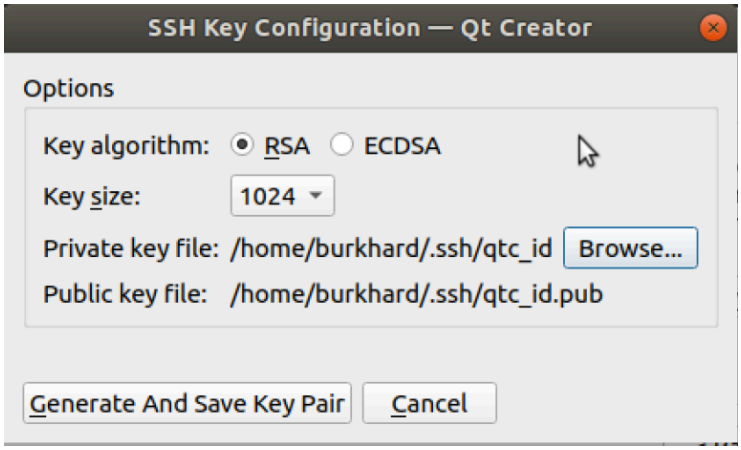
The easiest way (a.k.a. the only way I know) to connect the development PC and the Raspberry Pi over SSH is to put them on the same subnetwork. If we do application development in a Linux VM, we use bridge networking between the host PC and the VM. The VM gets its an IP address on the same subnetwork as the host PC.

If not done yet, we power on the Raspberry Pi. The Raspberry Pi starts the radio application and starts playing the preset radio station. The Raspberry Pi is in the same subnetwork as the development PC.

In QtCreator, we open the dialog *Tools* | *Devices* | *Devices* and press the *Add* button. We select the option *Generic Linux Device* from the dialog *Device Configuration Selection* and press the *Start Wizard* button. In the first wizard step *Connection*, we enter *RaspberryPi* as the configuration name, 192.168.1.82 as the IP address of the target device and *root* as the user name for logging into the device.

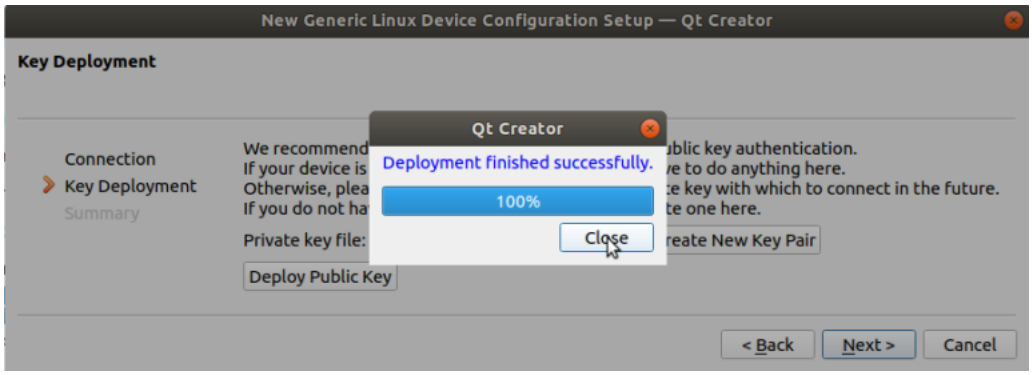


We press the *Next* button to go to the second wizard step *Key Deployment*. We press the button *Create New Key Pair*, which brings up the *SSH Key Configuration* dialog.



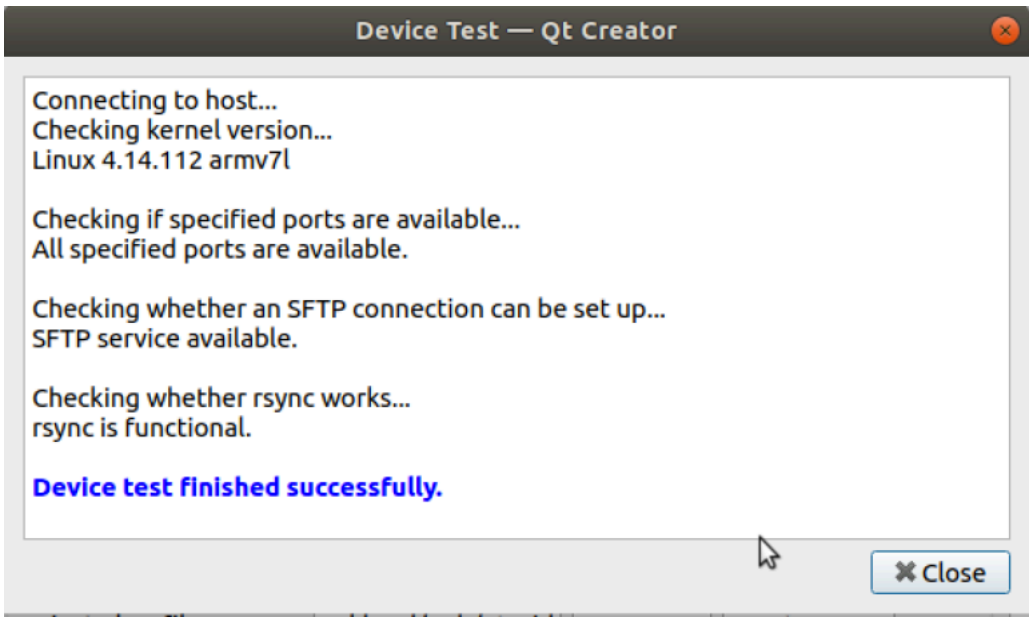
We press the button *Generate And Save Key Pair*. QtCreator shows the next dialog in response.

We press the button Deploy Public Key. QtCreator copies the public key to the Raspberry Pi, adds it to the file */home/root/.ssh/authorized_keys* and confirms it with a dialog. Computers, on which the corresponding private key is stored, can access the Raspberry Pi over SSH.

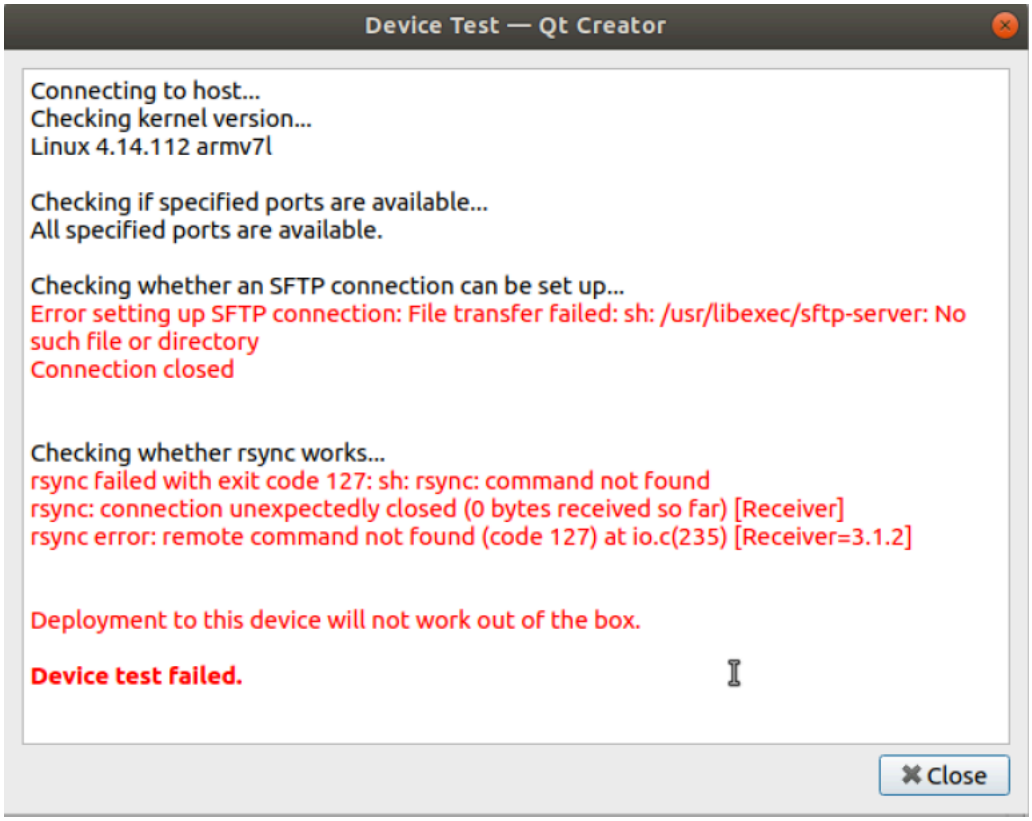


We close the pop-up dialog and press the *Next* button on the dialog underneath. QtCreator shows the final wizard step.

We finish the SSH setup by pressing the *Finish* button. If we set up SSH correctly, we'll the following dialog.



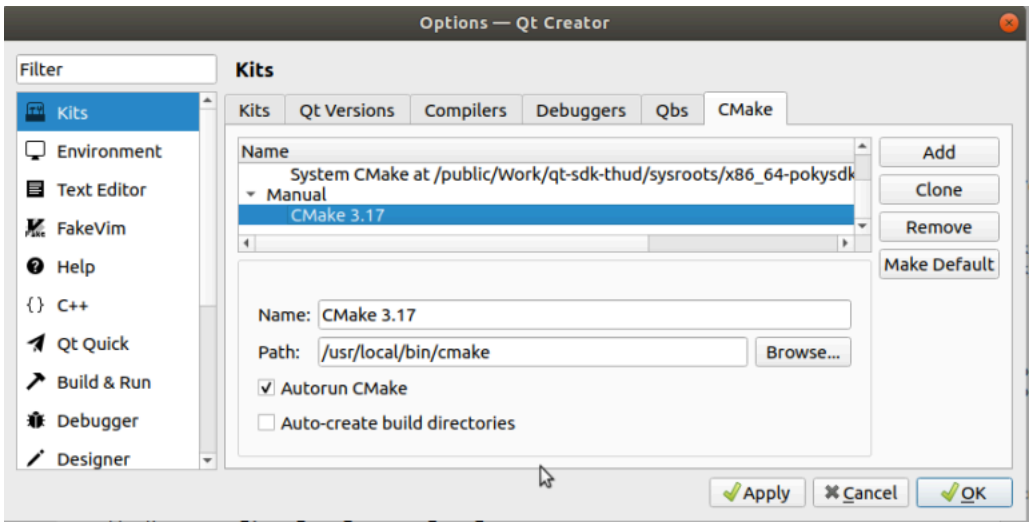
If something went wrong during the setup, we'll be greeted by an error dialog.



These error messages made me add the packages *rsync* and *sftp-server* to the Linux image.

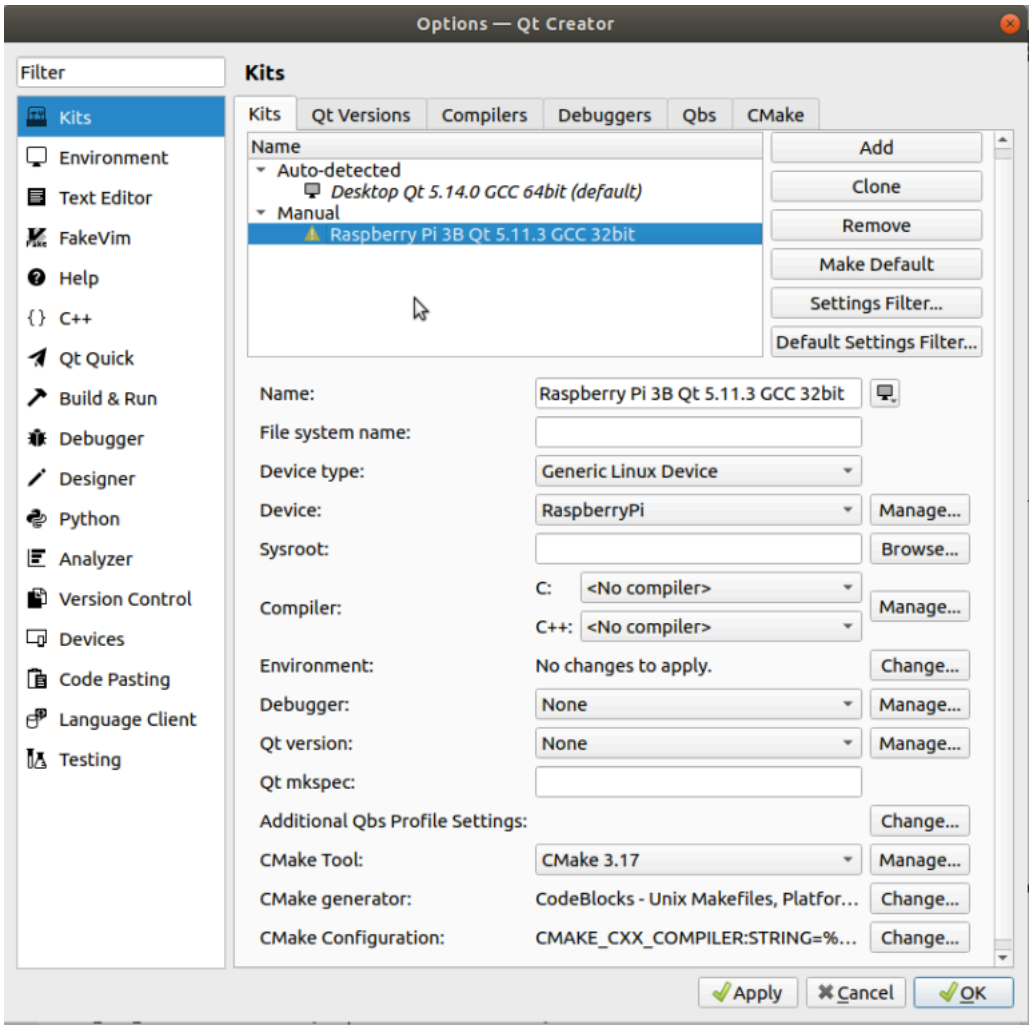
Selecting CMake from Host PC

QtCreator finds CMake 3.12.2 from the Qt SDK automatically, as a look at the tab page *Tools* | *Options* | *Kits* | *CMake* shows. However, we want to use the CMake version that we installed in Prerequisites. We press the *Add* button on the *CMake* tab page and browse to the CMake excutable (e.g., */usr/local/bin/cmake*). The result looks something like this.



Assembling a Kit

We change to the tab page *Tools* | *Options* | *Kits* | *Kits* to define a kit from the Qt Version, the Compilers and CMake. The filled-out form looks as follows.



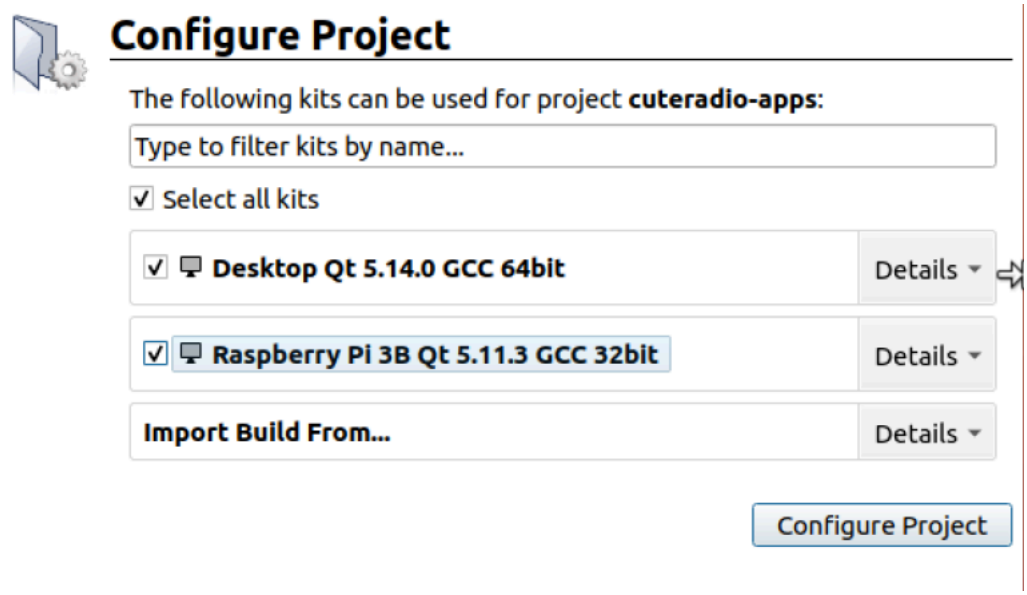
We enter a telling *Name* for the kit: *Raspbery Pi 3B Qt 5.11.3 GCC 32bit*. We choose *Generic Linux Device* as the *Device type*, *RaspberryPi* as the *Device* and *CMake 3.17* as the *CMake Tool*. We don't use a *Debugger* for now.

Important! We do **not** set *Sysroot*, *Compiler*, *Qt version* and *Qt mkspec*, as they are handled by the CMake toolchain file. If we set any of these variables, we'll spend a lot of time trying to figure out why running CMake fails. We'll work out the CMake Configuration in Running CMake below.

Building and Running the Application

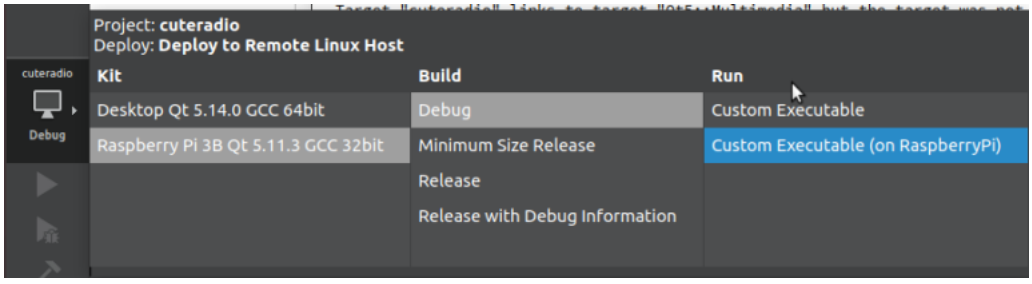
Running CMake

In QtCreator, we execute the action *File* | *Open File or Project…* or press *Ctrl+O* and open the file */public/Work/cuteradio-apps/CMakeLists.txt*. In the next dialog, we select the *Desktop* and the *Raspberry Pi* kit as shown and press the *Configure Project* button.



QtCreator takes the first kit in the list, the *Desktop* kit, as the current kit. If we hit *Ctrl+R*, QtCreator will build and run the radio application for the *Desktop*. The radio application starts play the station *Antenne Bayern*.

We switch to the *Raspberry Pi* kit by opening the configuration switcher towards the bottom of the left toolbar and by selecting *Raspberry Pi 3B Qt 5.11.3 GCC 32bit* as the kit, *Debug* as the build configuration and *Custom Executable (on RaspberryPi)* as the run configuration.



QtCreator runs CMake as a response to our selection and prints a couple of warning and error messages in the output pane *General Messages*. The first line of the output shows the CMake command line. We are interested in the settings of the cached CMake variables passed with option `-D`.

```
-DCMAKE_BUILD_TYPE:STRING=Debug
-DCMAKE_CXX_COMPILER:STRING=
-DCMAKE_C_COMPILER:STRING=
-DCMAKE_PREFIX_PATH:STRING=
-DQT_QMAKE_EXECUTABLE:STRING=
```

These settings look alright, as we do a Debug build and we didn't set the C compiler, the C++ compiler and the QMake executable. The unset `CMAKE_PREFIX_PATH` explains why CMake cannot find any configurations for Qt modules (e.g., *Qt5CoreConfig.cmake*). We also see that QtCreator doesn't pass a toolchain file to CMake. Let us remedy this.

We clear the output pane General Message so that we can recognise new messages from the next CMake run more easily. We clear CMake's variable cache CMakeCache.txt by executing the action *Build* | *Clear CMake Configuration*. This prevents us from debugging problems with stale CMake variables. We select the Raspberry Pi kit on the tab page *Tools* | *Options* | *Kits* | *Kits* and press the button *Change…* in the bottom line *CMake Configuration*.

We change the value of `CMAKE_PREFIX_PATH` and add a line for `CMAKE_TOOLCHAIN_FILE`. The resulting CMake configuration looks as follows.

```
CMAKE_CXX_COMPILER:STRING=%{Compiler:Executable:Cxx}
CMAKE_C_COMPILER:STRING=%{Compiler:Executable:C}
QT_QMAKE_EXECUTABLE:STRING=%{Qt:qmakeExecutable}
CMAKE_PREFIX_PATH:STRING=%{Env:OECORE_TARGET_SYSROOT}/usr
CMAKE_TOOLCHAIN_FILE:STRING=%{Env:OE_CMAKE_TOOLCHAIN_FILE}
```

We close the two open dialogs by pressing their *OK* buttons. QtCreator runs CMake. CMake fails with some warnings and errors. The first warning reads:

```
CMake Warning at
$OECORE_TARGET_SYSROOT/usr/lib/cmake/Qt5Core/Qt5CoreConfig.cmake:7
(message):
  SkippingbecauseOE_QMAKE_PATH_EXTERNAL_HOST_BINSisnotdefined
```

```
Call Stack (most recent call first):
  CMakeLists.txt:17 (find_package)
```

A look into *Qt5CoreConfig.cmake* reveals the problem:

```
if(NOT DEFINED OE_QMAKE_PATH_EXTERNAL_HOST_BINS)
    message(WARNING Skipping because OE_QMAKE_PATH_EXTERNAL_HOST_BINS is
not defined)
    return()
endif()
```

The rest of this CMake file is not executed, because the CMake variable `OE_QMAKE_PATH_EXTERNAL_HOST_BINS` is not defined. Although it looks like an environment variable from the SDK shell, it is a CMake variable. This variable is used, for example, in *Qt5CoreConfigExtras.cmake* to specify the absolute path to *qmake*, *moc* and *rcc*. The SDK shell provides an environment variable called `OE_QMAKE_PATH_HOST_BINS` for this purpose.

We set the CMake variable `OE_QMAKE_PATH_EXTERNAL_HOST_BINS` to the value of the environment variable `OE_QMAKE_PATH_HOST_BINS` in QtCreator's CMake Configuration. We first clear the messages in the General Messages pane and clear the CMake Configuration. We then open the dialog to change the *CMake Configuration* from *Tools* | *Options* | *Kits* | *Kits* | *Raspberry Pi 3B*.

We add the following line to the configuration:

```
OE_QMAKE_PATH_EXTERNAL_HOST_BINS:STRING=%{Env:OE_QMAKE_PATH_HOST_BINS}
```

Here is the complete CMake configuration for reference.

```
CMAKE_CXX_COMPILER:STRING=%{Compiler:Executable:Cxx}
CMAKE_C_COMPILER:STRING=%{Compiler:Executable:C}
CMAKE_PREFIX_PATH:STRING=%{Env:OECORE_TARGET_SYSROOT}/usr
CMAKE_TOOLCHAIN_FILE:STRING=%{Env:OE_CMAKE_TOOLCHAIN_FILE}
```

```
QT_QMAKE_EXECUTABLE:STRING=%{Qt:qmakeExecutable}
OE_QMAKE_PATH_EXTERNAL_HOST_BINS:STRING=%{Env:OE_QMAKE_PATH_HOST_BINS}
```

We close the two dialogs with *OK*. QtCreator runs CMake without any warnings or errors.

Cross-Building the Application

We execute the menu action *Build* | *Build Project "cuteradio"* or hit *Ctrl+B* to cross-build the radio application. The *Compile Output* pane shows the build messages.

```
11:52:21: Running steps for project cuteradio...
11:52:21: Persisting CMake state...
11:52:21: Starting: "/usr/local/bin/cmake" --build . --target all
Scanning dependencies of target cuteradio_autogen
[ 16%] Automatic MOC for target cuteradio
...
Scanning dependencies of target cuteradio
[ 50%] Building CXX object
...
[100%] Linking CXX executable cuteradio
[100%] Built target cuteradio
11:52:23: The process "/usr/local/bin/cmake" exited normally.
11:52:23: Elapsed time: 00:02.
```

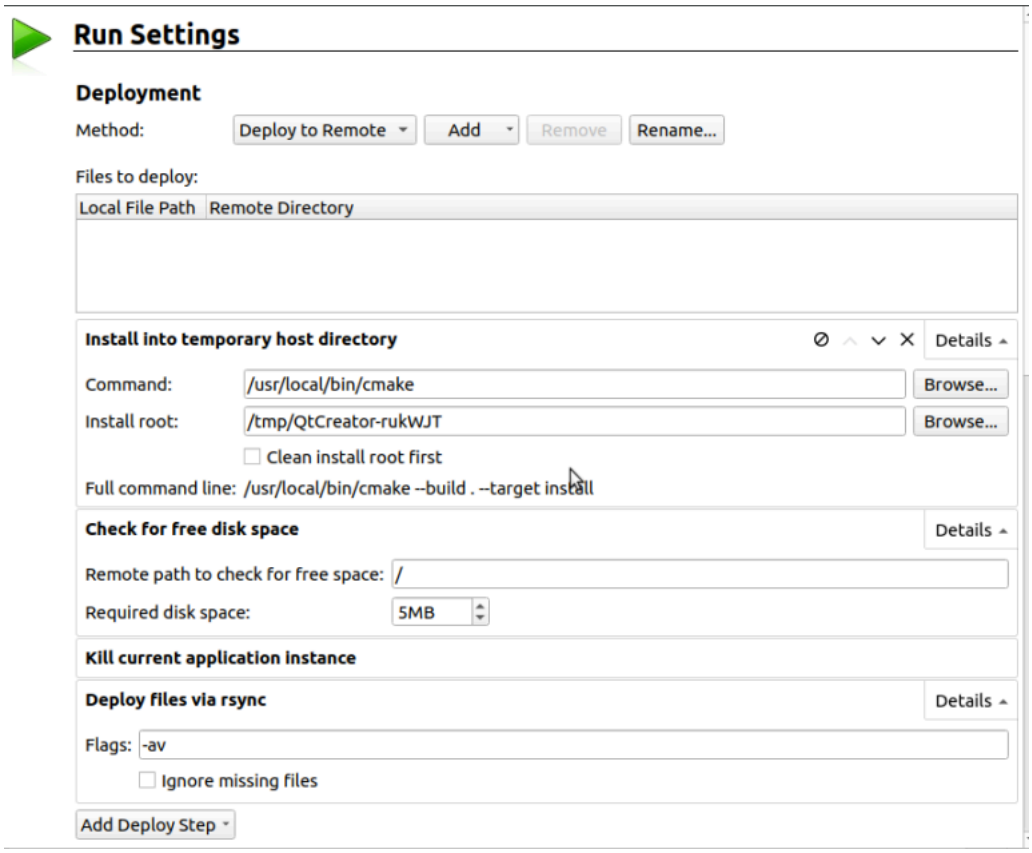
Checking that the executable was built for the target's ARM architecture and not for the host's Intel architecture is a good idea. It's too easy to mess up things when cross-compiling. The `file` command shows that all is fine. The `ll` command reveals that the executable has a size of 379 KB.

```
$ cd /public/Work/build-*Raspberry*-Debug/
$ file cuteradio
cuteradio: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-armhf.so.3,
for GNU/Linux 3.2.0, BuildID[sha1]=33f1788, with debug_info,
not stripped
```

```
$ ll -h cuteradio
-rwxr-xr-x 1 burkhard burkhard 379K Jun 18 11:52 cuteradio
```

Running the Application

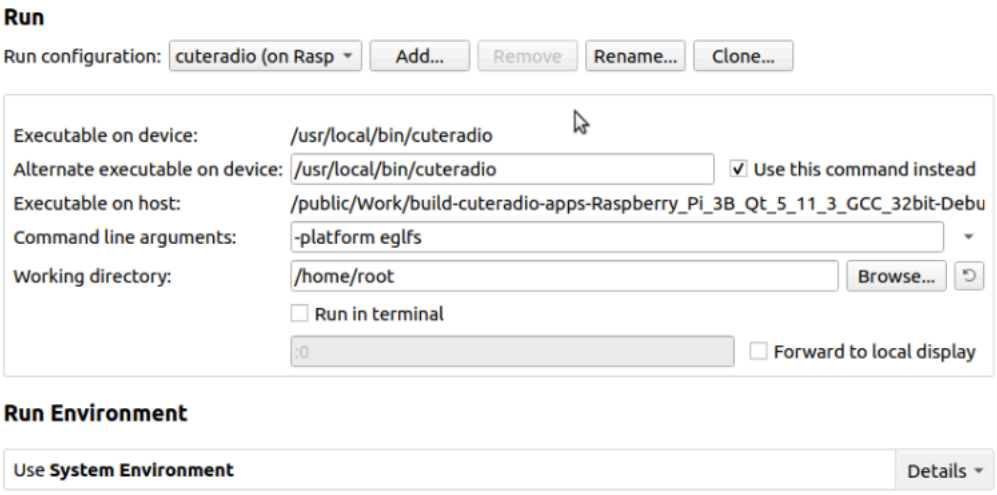
We open the run settings of the project by selecting *Projects* | *Raspberry Pi…* | *Run*. The *Deployment* settings should look like this. For a change, the default settings are OK.



QtCreator fills out the *Files to deploy*, once it runs the *install* target. The files to deploy are specified with the CMake `install` functions in the CMakeLists.txt files. We only install the application executable. QtCreator stages the files to deploy in the install root, before it transfers the files to the target device with `rsync` in the last deployment step.

QtCreator checks whether there is enough free disk space on the target device. The 5 MB given are more than enough for the 379-KB radio application. QtCreator kills the running application, before it copies the staged files to the device with `rsync`.

We scroll down on the run settings page to the section *Run* and fill out its fields as follows.



Our big moment has arrived. We execute the menu action *Build* | *Run* or simply hit *Ctrl+R*. The radio station stops playing. The screen of the Raspberry Pi goes black for a couple of seconds, before it shows the radio application again. The radio station starts playing again.

QtCreator logs the build and deployment steps in the *Compile Output* pane. We see the staging step, the check for free disk space, the killing of the application and the `rsync` step.

```
13:12:45: Running steps for project cuteradio...
13:12:45: Starting: "/usr/local/bin/cmake" --build . --target all
...
[100%] Built target cuteradio
Install the project...
-- Install configuration: "Debug"
-- Installing: /tmp/QtCreator-rukWJT/usr/local/bin/cuteradio
13:12:47: The process "/usr/local/bin/cmake" exited normally.
13:12:47: The remote file system has 461 megabytes of free space, going
ahead.
13:12:47: Deploy step finished.
13:12:47: Trying to kill "/usr/local/bin/cuteradio" on remote device...
13:12:48: Remote application killed.
13:12:48: Deploy step finished.
13:12:48: sending incremental file list

13:12:48: cuteradio

13:12:48:
sent 2,322 bytes  received 3,359 bytes  11,362.00 bytes/sec
```

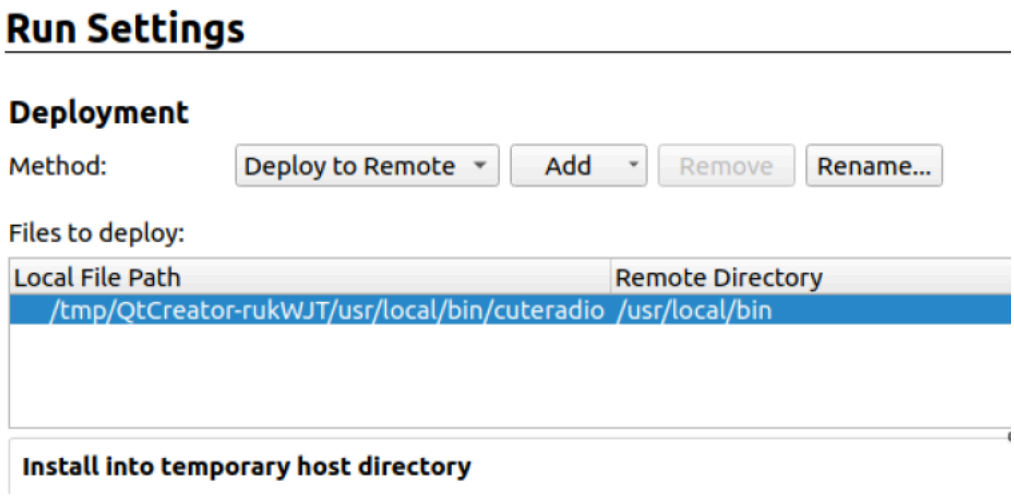
```
total size is 387,208  speedup is 68.16

13:12:48: Deploy step finished.
13:12:48: Elapsed time: 00:03.
```

The Application Output pane shows the messages from starting the application.

```
13:12:48: Starting /usr/local/bin/cuteradio -platform eglfs...
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
Unable to query physical screen size, defaulting to 100 dpi.
To override, set QT_QPA_EGLFS_PHYSICAL_WIDTH and
QT_QPA_EGLFS_PHYSICAL_HEIGHT (in millimeters).
@ Audio device =  "default"
@ Audio device =  "default:CARD=ALSA"
@ Audio device =  "sysdefault:CARD=ALSA"
@ Default audio device =  "default"
```

The page Run Settings | Deployment now lists the files to deploy, which is just the application executable in our case.



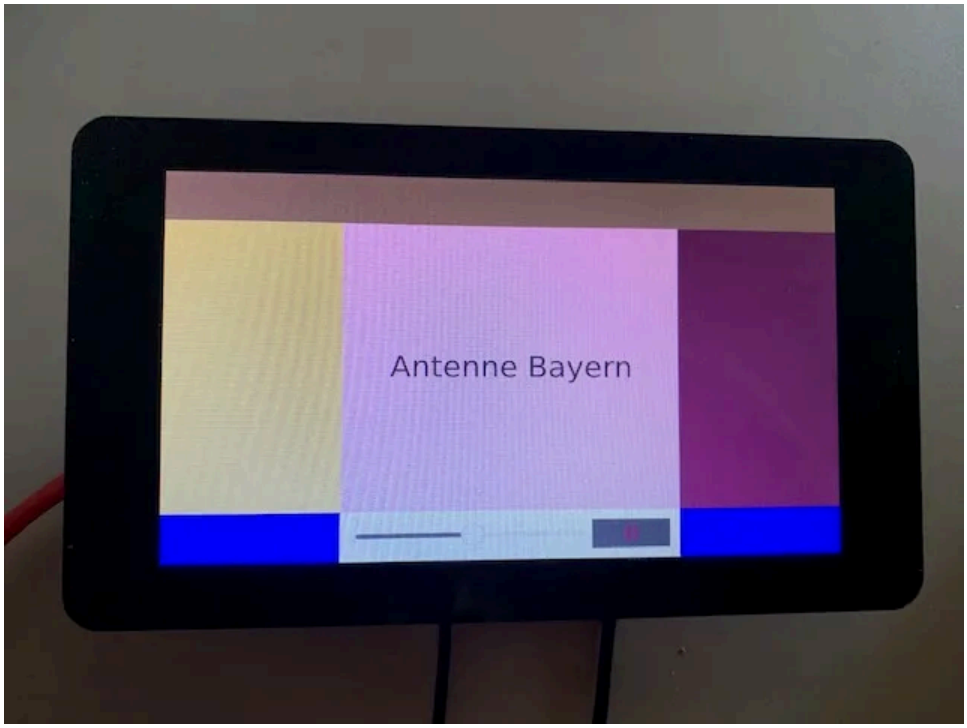
The Edit-and-Run Cycle in Action

We set out with the goal to make development on an embedded device as easy as on a desktop PC. We try out the edit-and-run cycle.

The central item displaying the currently playing radio has a yellow background. We change this colour to pink in *main.qml*.

```
Rectangle {
    id: stationInfo
    ...
    color: "pink" // Was "#FAFF70"
    Label { ... }
}
```

We press Ctrl+R to build, deploy and run the application. And, indeed, the central item has now a pink background.



About the Series

This post is part of a series on Qt Embedded Systems. I plan to write one post per month. The goal is to build a full-blown Internet radio running on a custom embedded Linux system powered by a Raspberry Pi. I'll walk you through all the steps needed to build a product. Topics will include QML, Qt, C++, Wayland, Wifi, Bluetooth, Yocto, fast start-up, OTA updates, etc.

So far in the Series:

- Part 1: Building a Linux Image with Yocto.
- Part 2: Building a Qt SDK with Yocto (this post).

I have plenty of ideas for the next posts:

- We learn how to write the recipes for the *meta-cuteradio* layer and clean up the existing recipes.
- We extract reusable parts of *meta-cuteradio* into a separate layer. The new layer provides several base Linux images, on which products like Cuteradio can build.
- We use the Linux package manager to update the packages on the target devices.
- We upgrade the Linux system from Yocto Thud to Yocto Danfell.
- The radio uses Wifi for the Internet connection and Bluetooth for the speaker.
- The radio supports multiple radio stations from different categories.
- We implement a Wayland-based window and application manager to accomodate multiple applications like alarm clock, image viewer and settings.
- And more…

Tags: EMBEDDED LINUX EMBEDDED SYSTEMS QT RASPBERRY PI SDK YOCTO

7 thoughts on “Qt Embedded Systems – Part 2: Building a Qt SDK with Yocto”

Matthias Klein

2020/06/30 at 07:30

REPLY

Do you know the yocto for the raspberry from jumpnowtek?:
https://jumpnowtek.com/rpi/Raspberry-Pi-Systems-with-Yocto.html

A dual-boat with U-Boot and RAUC I would find very interesting.

Burkhard Stubert

2020/06/30 at 17:51

REPLY

Jumpnowtek's git repository shows that their *meta-rpi* layer is available from *jethro* to *dunfell*. I plan to update my layer to *warrior* and *dunfell* this month.

OTA updates with RAUC are on my todo list. Thanks for you feedback.

What are the two operating systems for dual boot?

Matthias Klein

2020/06/30 at 18:00

REPLY

Twice the same Yocto, which always overwrites the inactive one when updating.

Alexander Mueller

2023/04/26 at 16:37

REPLY

Hey, your post, and also the in-depth explanation of the script to set up QT Creator helped me a ton already.

There is one detail I had to tweak, so QT was detected for me, but that may be my setup (btw. I am using Xubuntu 22.04 QT Creator 6.01):

${SDKTOOL} \
addQt \
–id "${BASEID}.qt" \

–name “${NAME}” \
–type “Qt4ProjectManager.QtVersion.Desktop” \
–qmake “$(type -p qmake)”

But I am still stuck halfway through:

When I build from the shell everything is fine. If I build a .pro qmake project using the new kit, everythings works as expected.

But when I try to compile the same code, but with a CMakeLists.txt / cmake project, I get the error:
fatal error: gnu/stubs-soft.h: No such file or directory
7 | # include
| ^~~~~~~~~~~~~~~~~

It seems as if some –sysroot/mkspecs setting get lost in QT Creator.

We are cross-compiling to Yocto (the code we built from the shell works on the target as expected).

Do you have any idea, where to look?

Thanks a lot for you post and thanks a lot in advance if you have a comment/suggestion for my problem!

Alex

Alexander Mueller

2023/04/26 at 17:06

REPLY

Follow-Up to my question: I found the solution.
CMAKE_C_FLAGS/CMAKE_CXX_FLAGS gets set to “” by the toolchain file instead of the settings for the target. I fixes this with an additional .cmake file in toe conf.d directory.

Burkhard Stubert

2023/05/06 at 09:50

REPLY

Glad that you found a fix! I think I need to update the post. I learned a lot since I wrote it.

MD

2023/08/10 at 10:23

REPLY

Hi Alexander,
can you explain what you did exactly to fix that issue? I have the same. I use the yocot toolchain file. In qt-creator 4.12 that worked. Now I use qtcreator 11.0 and I have the same issue that you have.

Thanks,
Michael

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

Search for...

Recent Posts

Extracting Microservices from a Modular Monolith

EU CRA: Essential Requirements Related to Product Properties

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

Categories

Select Category

Imprint

Privacy Policy

Terms of Use