Cross-Compile QT5 for the Raspberry PI

A little Guide to Cross-Compile QT for the PI to prevent the madness of “Try and Error”

A fresh QT is always cute

Preface

If we want to develop for the PI, why shall we do all following and time consuming things? We could just get a few packages for the PI and we are good to go, or aren't we ? We just want to build a nice Application with or without an UI.

Well this might be a way, but at the time of writing the Rasberian ships the QT in Version 5.7.1, which was developed in 2016. Also the QT Creator shipped with the Pi is from that age. Two years in the IT is like ages in the real world. You will miss all the improvements and most important all the bug fixes and security updates the QT got over the years. But if you ask why we do cross-compilation and not a build on the pi itself, well, I like my QT to be compiled in less than three days.

Why no one is providing freshly build packages will be a bit of a miracle by its own. As long as there non recent ones around we have to do it by our own. And on this way there a many dungeons to be walked and dragons to be killed. Therefore this guide shall be an assistant to ease your way through the adventure of getting an bleeding edge QT version on your PI and also have an development environment on your PC which will allow cross-compilation of your code, to speed things up, and remote debugging. Pick your gear may the journey being and the sources be with you….

Contents

Required Gear

Before we begin we need a few items to start. For the QT on PI we need:

- Rasperry Pi (preferred Model 2 or newer)
- SD-Card Reader
- SD-Card (8GB at least)
- A virtual machine hypervisor (we use Virtual Box here)
- A PC with 60Gb space left for the virtual machine
- Internet connection
- Monitor for the Pi (useful to see if it is working)
- A USB-Keyboard
- Ubuntu 18.04 ISO
- Time (if everything went well we will be ready in than less a day)

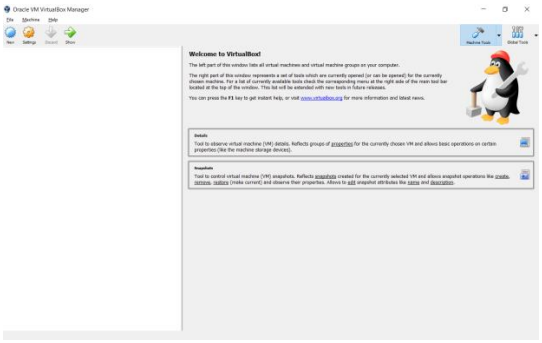
If we have all items prepared we can move on to get to our first stop. Install Virtual Box and create a Ubuntu virtual machine on it.

Install Ubuntu on a Virtual Box

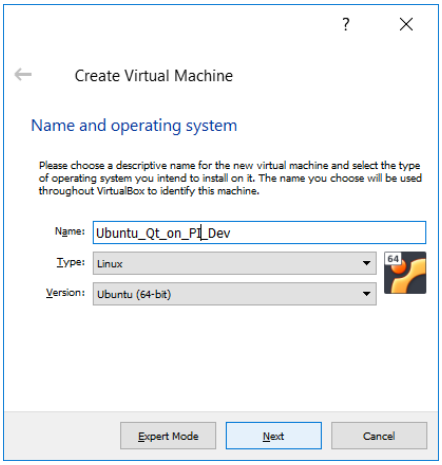
Why are we using a virtual machine, if we could install all tools to a physical system? In short terms, the physical system has no undo-button at all, with a virtual machine we can create snapshots and return to them any time we want if anything went wrong. Also you can move the virtual machine anywhere the virtual box runs or even clone the system if you want to setup a classroom. And if compilation will take longer than you expect, you just click on pause and resume another day. Also if your physical hardware gets wonky you can move to the next without many problems.

As we are using Windows as a host (or may we have to use it for different reasons) we get the current version of Virtual Box from www.virtualbox.org. At the time of writing this is version 5.2.12, and if you are reading this we may have a newer version around. Grab from the Download section the Version that matches your architecture, in my case the Windows version. If you are doing this for private use you may also download the Virtual Box extension pack. Be aware that for commercial use this requires a license. Also we shall have the current Ubuntu 18.04 ISO in place, which can be found at www.ubunut .org.

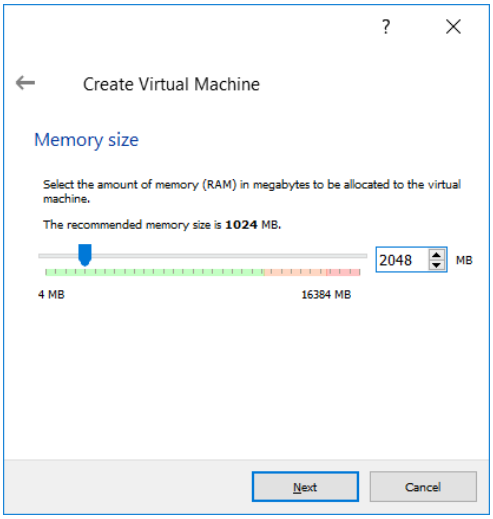
From here we install the Virtual Box and open it afterwards. The UI of the current Version looks like this.



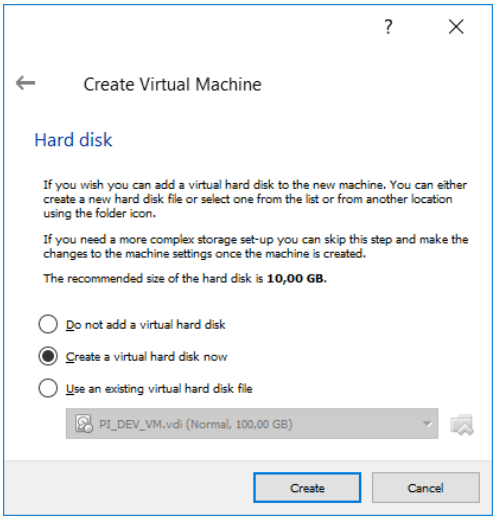
From here we start creating a new virtual machine. We use the “New”-button to start the wizard for the machine creation. As name for the virtual machine you can use anything you like. For the Type we choose “Linux” and for the Version “Ubunut (64-bit)”.

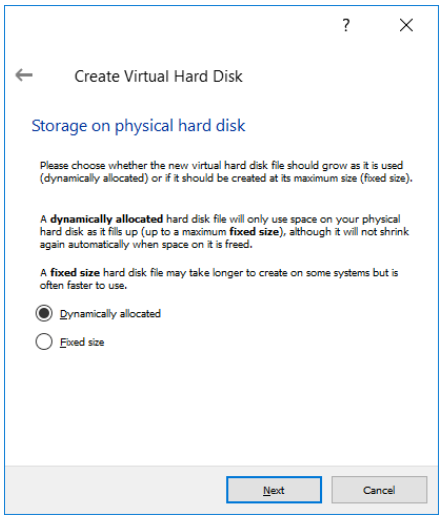


After we hit “Next” we need to set the amount of RAM the machine gets. If possible we use 2 GB, also 1 GB will be fine, everything less than this may cause slowdowns.

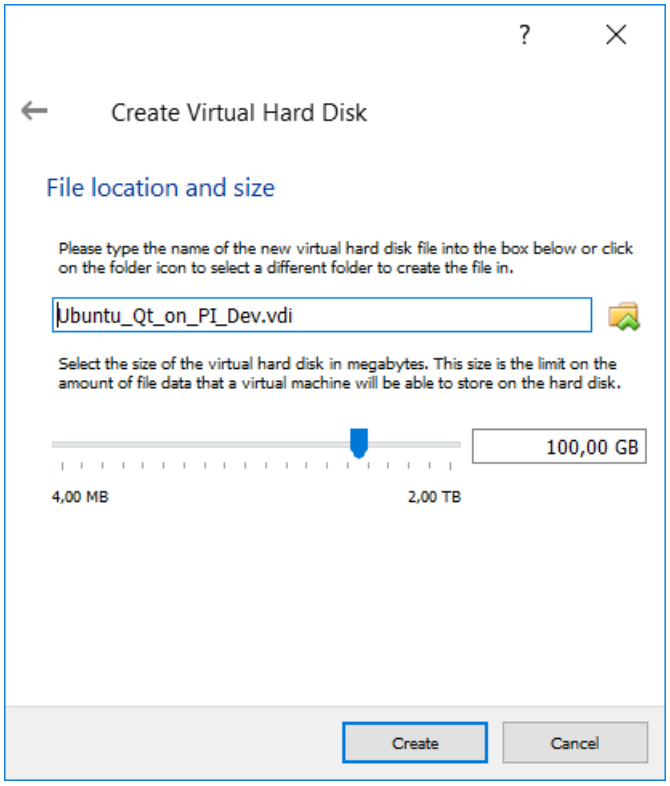


We confirm your selection with “Next” and create in the last step a virtual hard drive image. This image is a file on your PC which contains all the data written inside the virtual machine. This is like the image you put on the SD-card for the pi, besides the fact, that the pi won’t boot with this file. We now click on “Create” and a wizard for the virtual disc creation will pop up. We choose as VDI as file type and set the storage to be dynamically allocated.

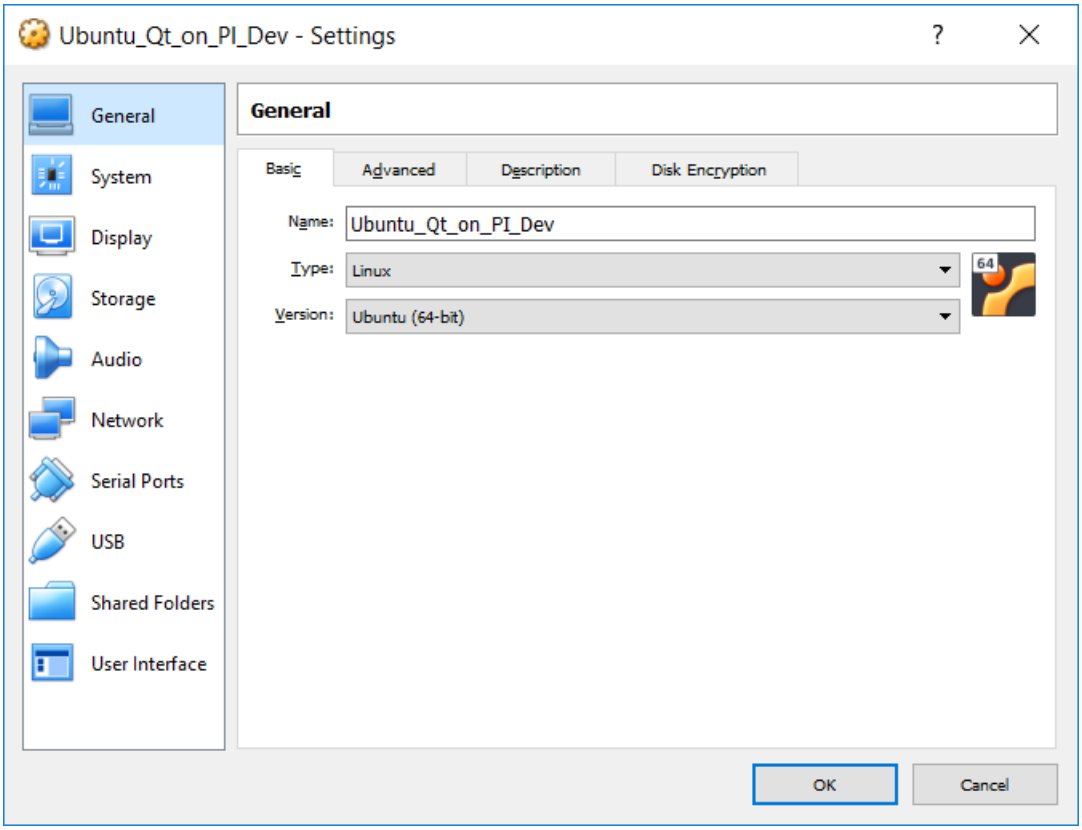




In the last step we choose the file location and set the size to be 100GB. After we hit “Create” the wizards close and we will be shown a new virtual machine.

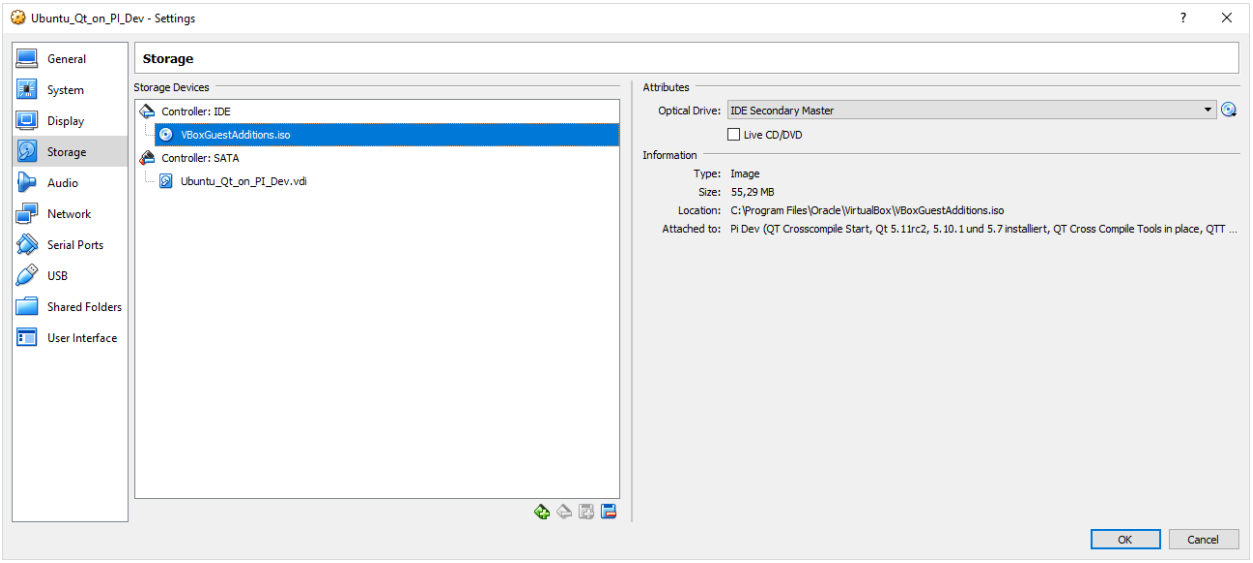


Now we need to adjust the virtual machine to use the Ubuntu ISO to boot. We select the created machine with the mouse and click on “Settings”. In the now opened Dialog we switch to “Storage” and select the CD-image, which shall be labeled “Empty”.

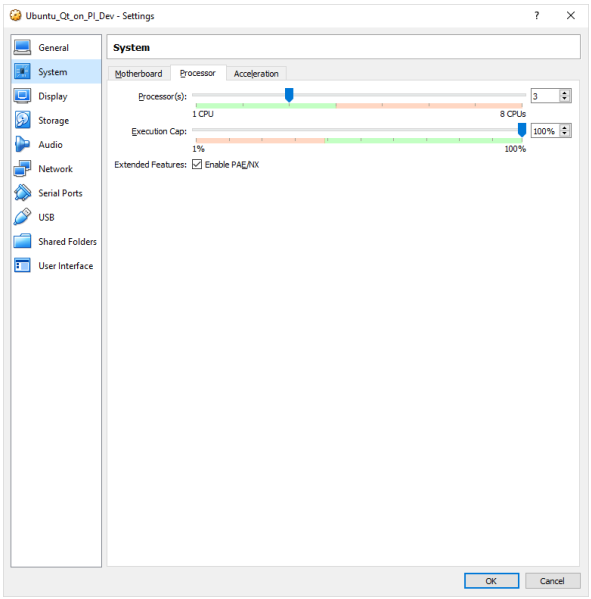


[Bild]

In "Storage" we use the disc image and select the Ubuntu ISO, by using "Choose Virtual Optical Disc File…". Now browse to the location of your Ubunut 18.04 iso.



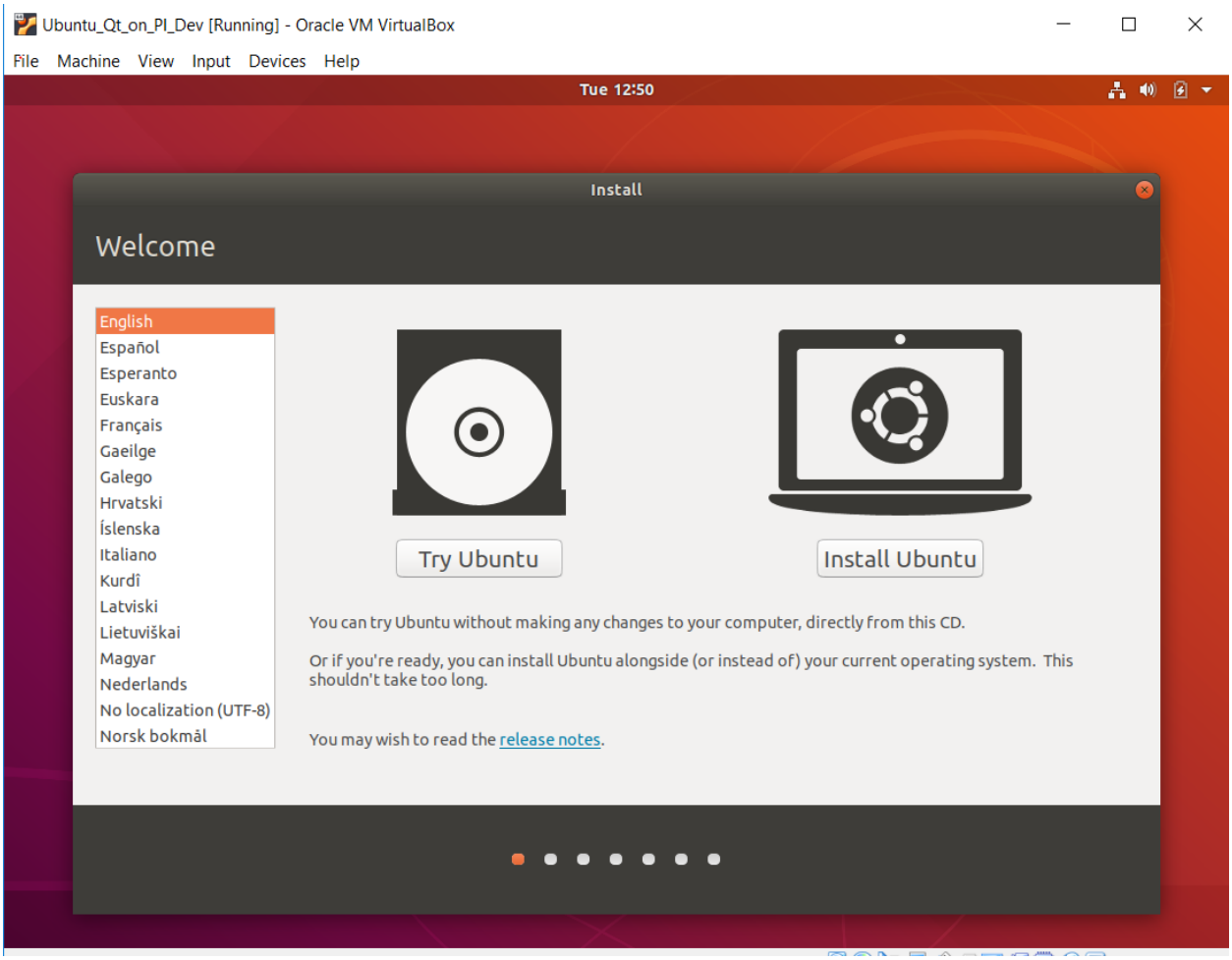
Also we want more than one CPU core to be used, as this will speed up things a lot. Go to System and in the Processor tab. At this point we use as many cores as the physical system has.



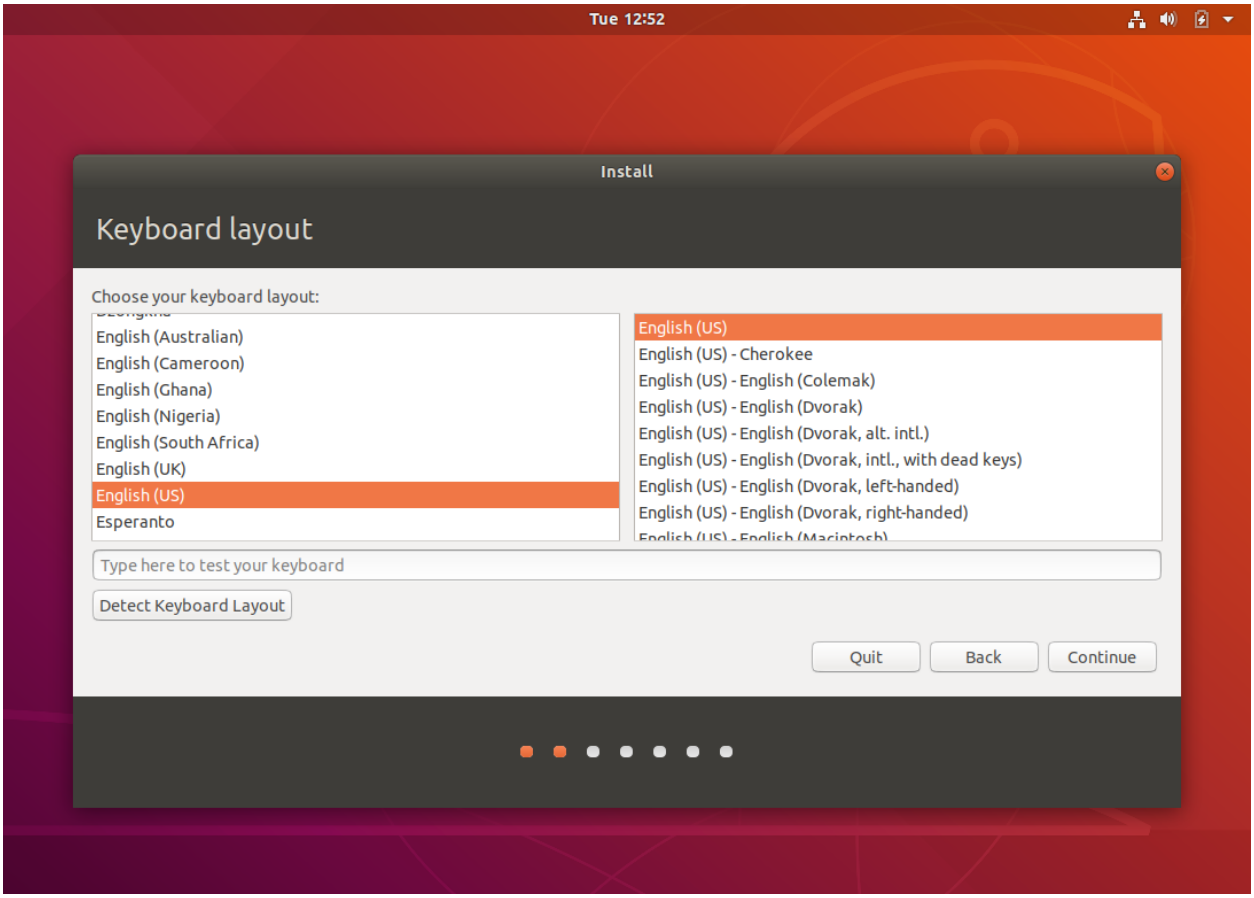
After this the virtual machine is ready to boot. After a bit of waiting the Setup shall welcome us for a new fresh install.

Install Ubuntu 18.04 LTS to the VM

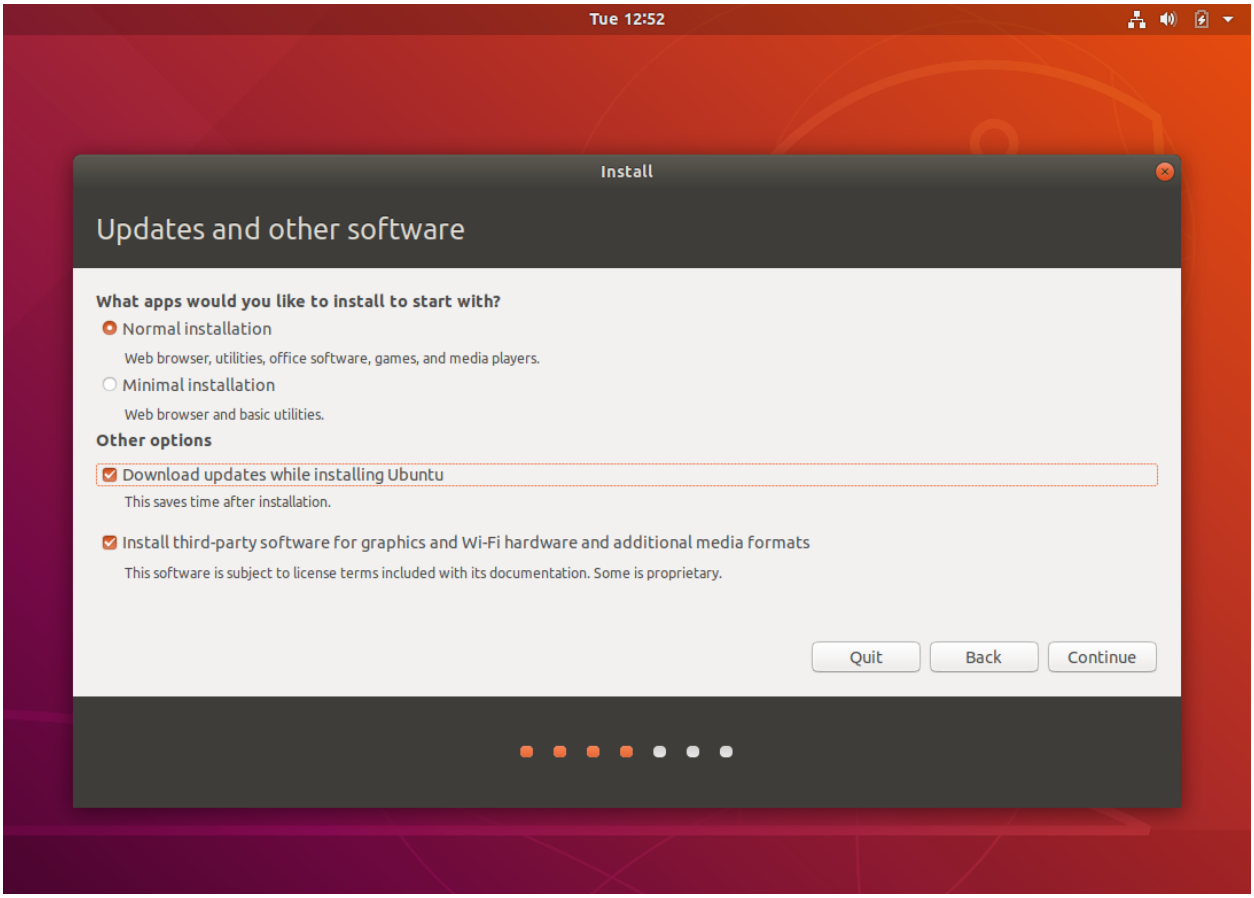
For convenience we cover the Ubuntu installation in short.



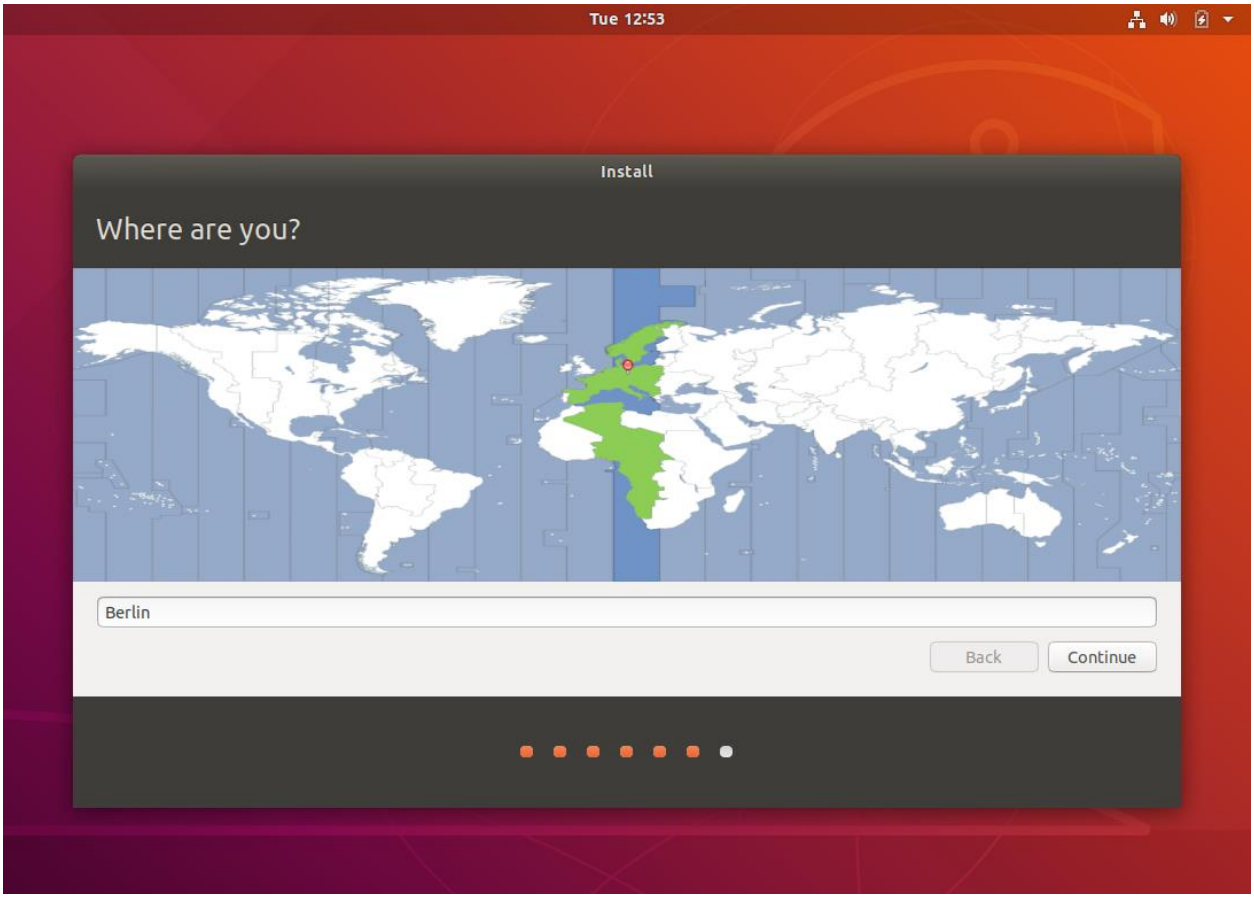
After you have powered on the virtual machine the boot will begin and after a while you will be greeted with the installer. Choose the language u like to have for the OS and continue with an click on “Install Ubuntu” or whatever this will be in your chosen language.



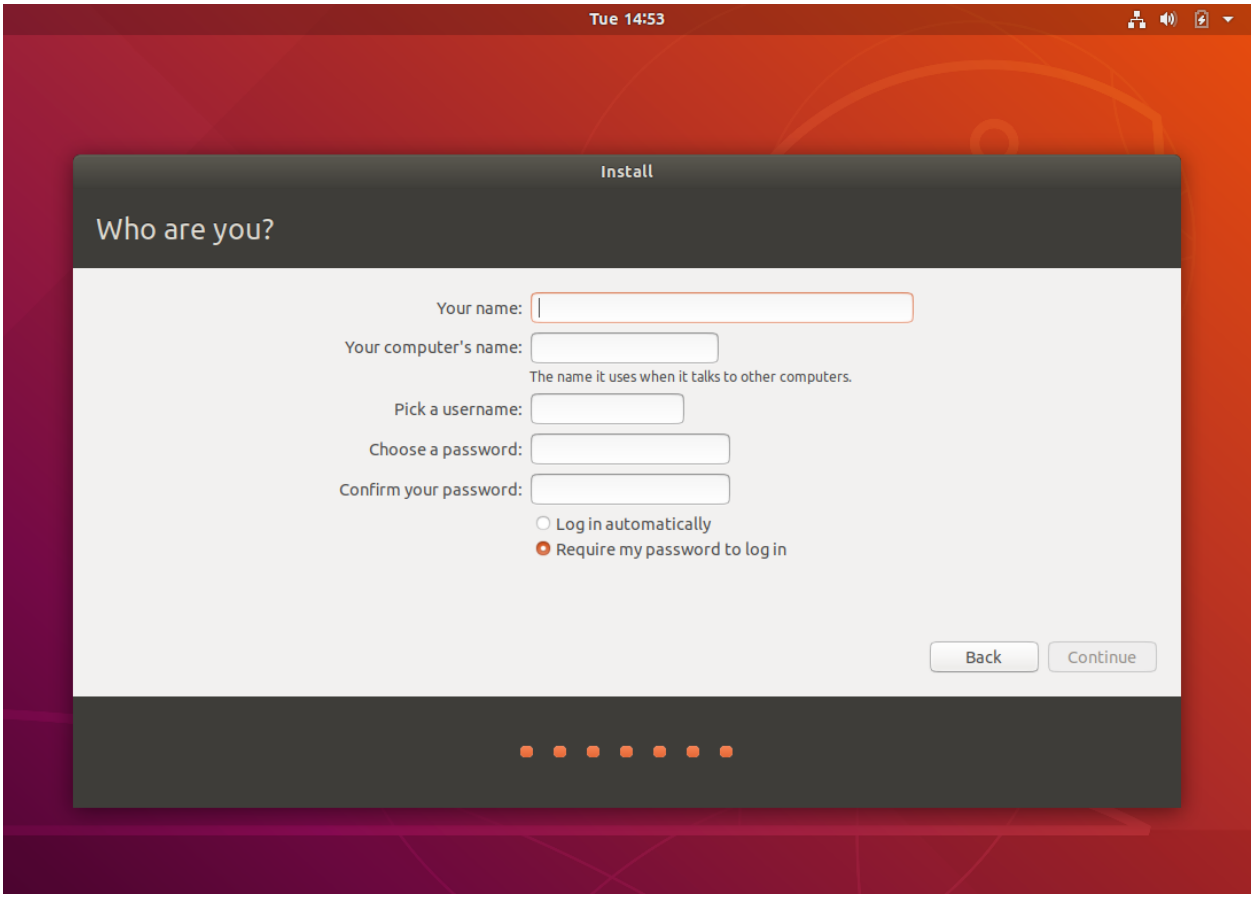
Pick the keyboard layout for you system afterwards, here we will use the default US setting. On your system set what is appropriate for your keyboard.



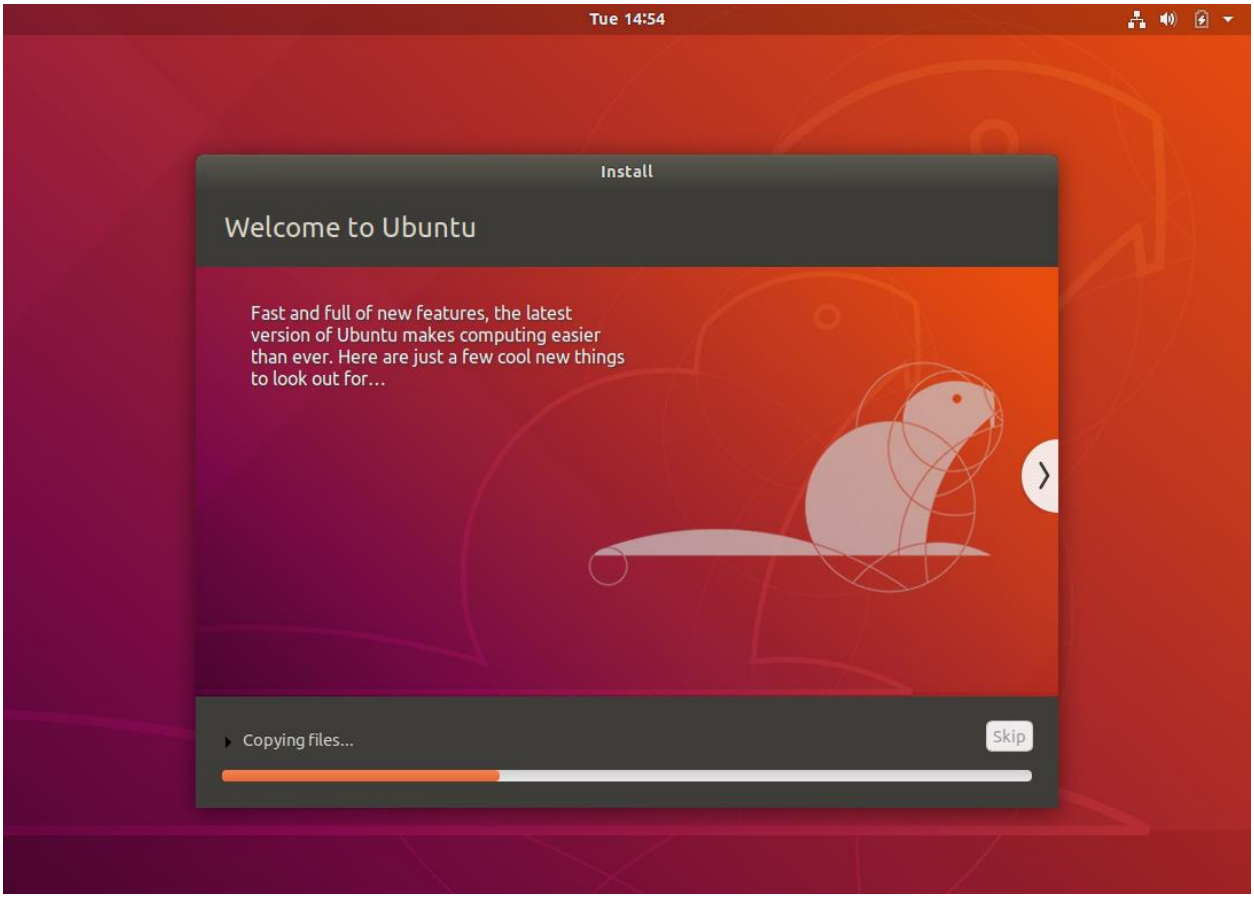
Check the “Normal installation”. We also want to download updates while installing Ubuntu and we want the third-party software to be installed.



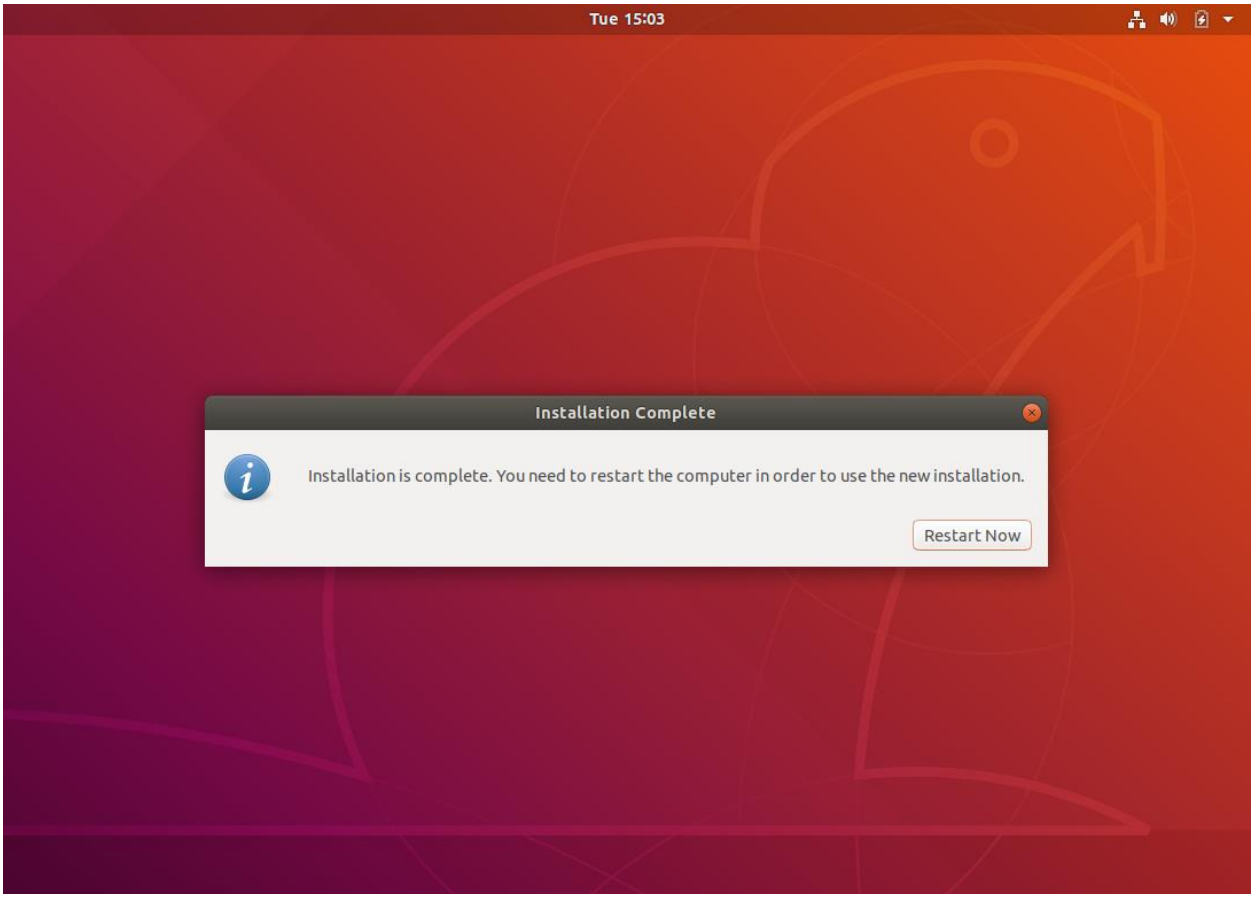
Depending on your region stet the Timezone.



Enter your user credentials. Be aware you need to remember the password for the machine. Otherwise a Login won't be possible.



And now it takes some time to complete the installation.



If no error occure you can press the restart button.

At this point, if you see the desktop after the installation, you have done the first step towards cross-compiling QT for the Pi, but there are more to follow.

Get the Pi Image ready

If you look for guides on cross-compiling QT for the Pi, they will tell you just to download a Rasbian image and use it, some are up to date and provide a really good starting point, e.g. https://github.com/AlbrechtL/welle.io/blob/master/RASPBERRY-PI.md . With the current release (**June 2018**) there are issues and if we want to build the QT. We need it to be transferred to the Pi and boot the system. As you are reading this because you are interested on QT on Pi, I assume you already know how to get the image on the SD-Card an read it back to the PC. After the Pi has booted initiate the image and go to a console, we don't need a running X at this point.

The first step is to have a working internet connection from the pi. You now to a rpi-update, to get the latest firmware for the pi. This also fixes some missing library files inside the system. Afterwards we need to install a list of packages. We will prepare the PI for the QT on X11 and also the EGLFS option. The later can only be used for application in full screen, but doesn't require an X11 server to be present.

We will add some libs and header to the pi. This will allow later to have all feature we require to build for the pi. We need:

- pulseaudio
- libglib2.0-dev
- libdbus-1-dev
- libfontconfig1-dev
- libx11-dev
- libglib2.0
- libx11-xcb-dev
- libxrender-dev
- libxi-dev
- libdrm-dev
- libinput10
- libinput-dev
- libxkbcommon0
- libxkbcommon-dev
- libudev0
- libudev1
- libpulse-dev
- libgstreamer1.0-dev
- libgstreamer-plugins-base1.0-dev
- libasound2-dev
- lirc-x

You can do it with an “sudo apt install pulseaudio libglib2.0-dev libdbus-1-dev libfontconfig1-dev libx11-dev libglib2.0 libx11-xcb-dev libxrender-dev libxi-dev libdrm-dev libinput10 libinput-dev libxkbcommon0 libxkbcommon-dev libudev0 libudev1 libpulse-dev libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev libasound2-dev lirc-x”

The last step here is to intstall a GDB server, to be able to do later some debugging. If you not intend to do so skip this one, else in a terminal type:

```
sudo apt-get install gdbserver
```

After pressing enter the system shall install the gdb server.

If we are done, we need to make an image from the SD-Card. This will be needed later for the cross-compilation of QT. It is suggested to have some spare SD-Cards by hand if something may went wrong. Also remember to keep the system up to date by doing an rpi-update and an apt-get update followed by an apt-get upgrade to bring the system to the latest available patch level.

To get an diskimage we can use dd on the linux vm. As Command we use:

```
sudo dd if=/dev/mmcblk0 of=~/raspi_image_clean.img bs=4M
```

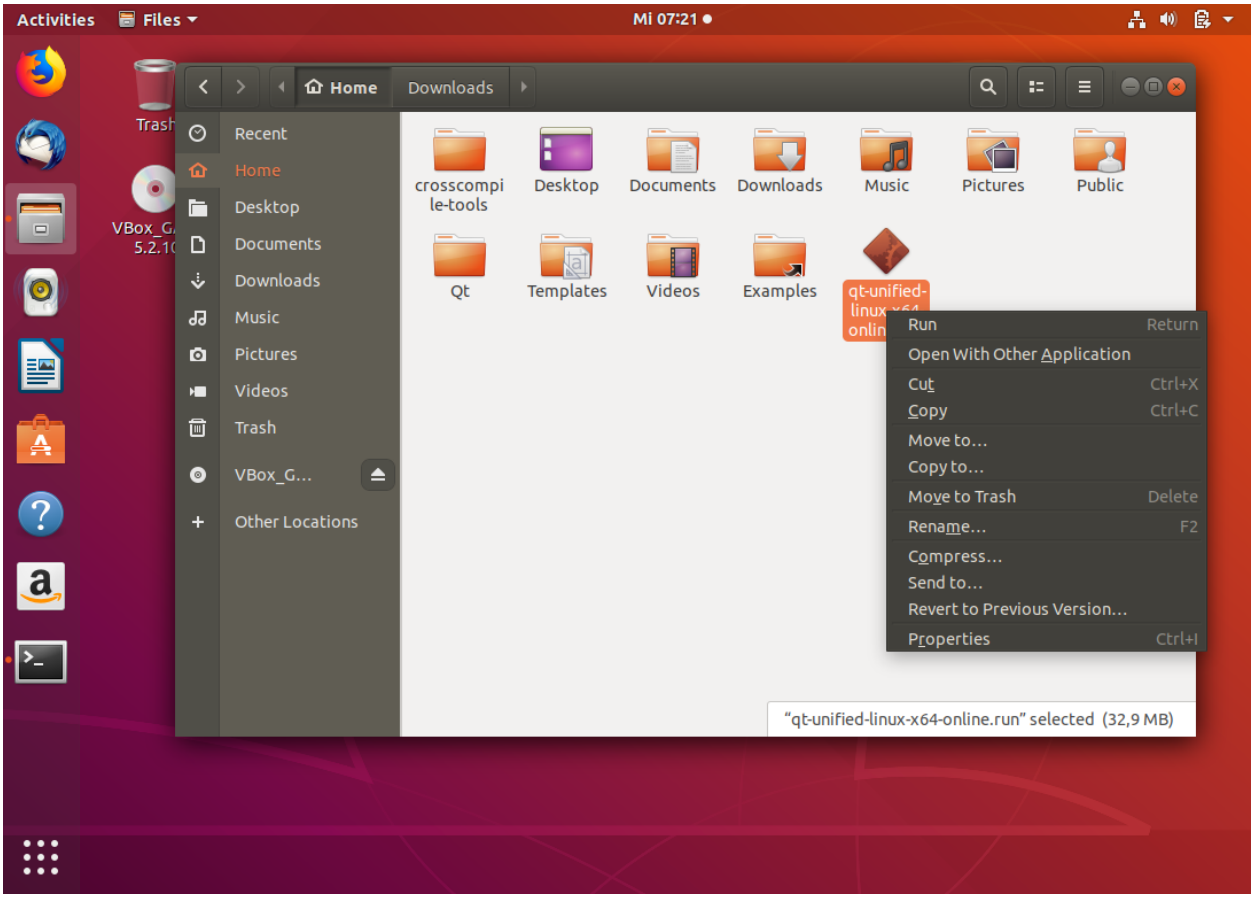
This will copy the raw data from the sdcard to your PC.

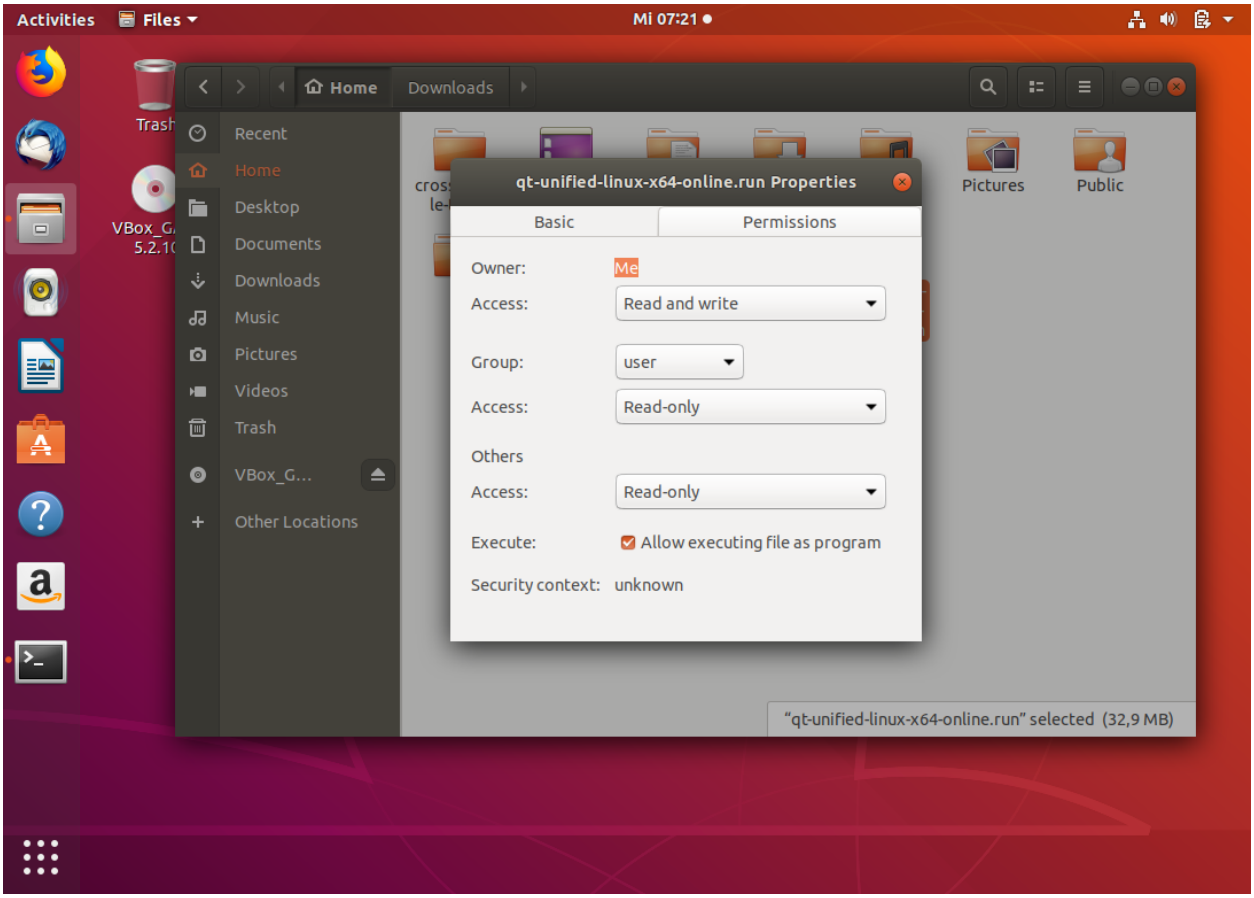
At this point the image will be sufficient to build the QT with it, any modification we need to do later can be done with

Getting QT in place

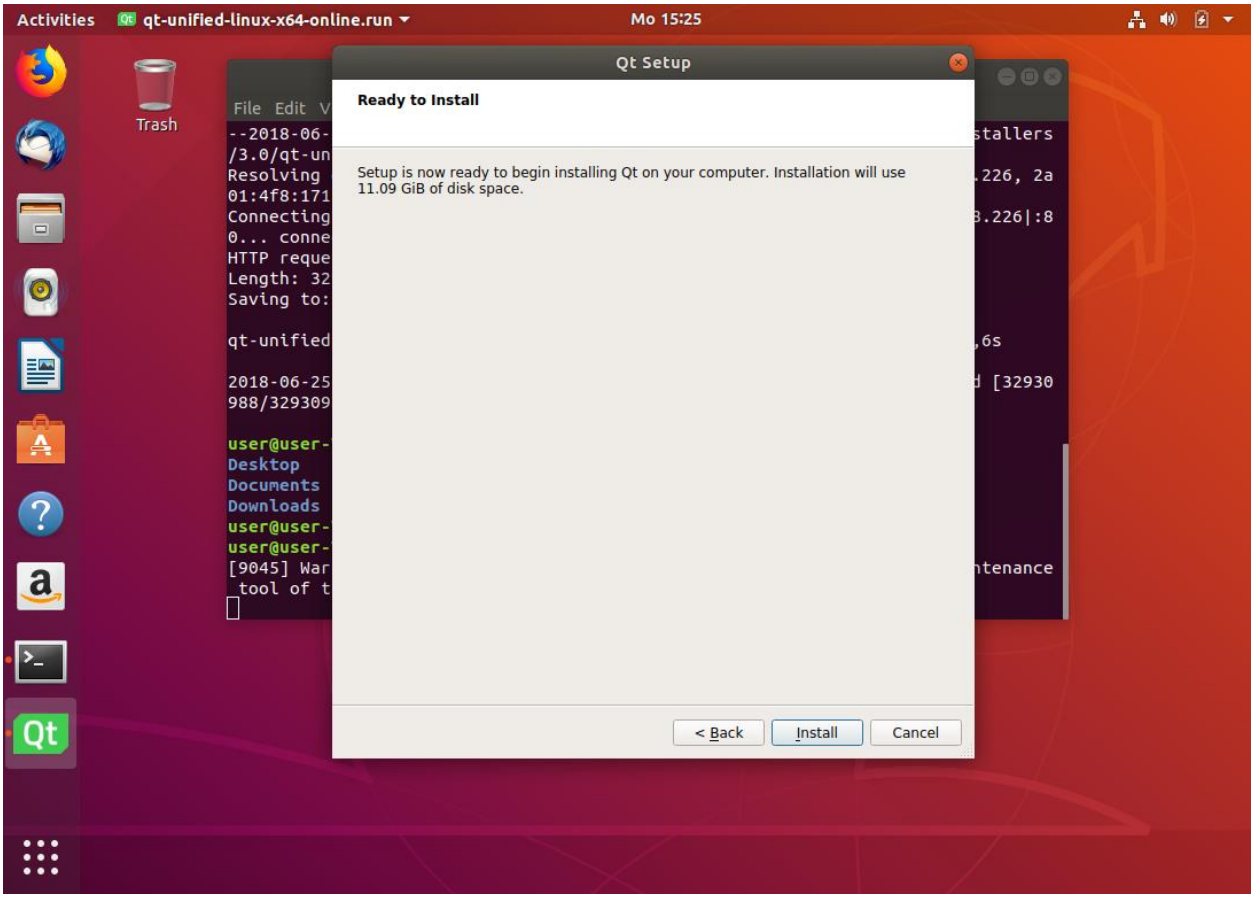
After the preparations are done we set up the Ubuntu with the QT Version we want to build. Currently this was at time of writing 5.10.1. Version 5.11 was released during the writing of the document. We will end up with an 5.11.1 build at the end of the document, but we need to change some files in the default compiler.

We need to get the Qt installer from the Qt website. To download it you can take the short path with an browser and enter http://download.qt.io/official_releases/online_installers/qt-unified-linux-x64-online.run which will give you the latest installer. After the download, assumed in the download folder of your user folder, execute it. To do so you need to set the downloaded file to executable. Navigate to the folder you downloaded the file and do a right click on it to open the file properties.

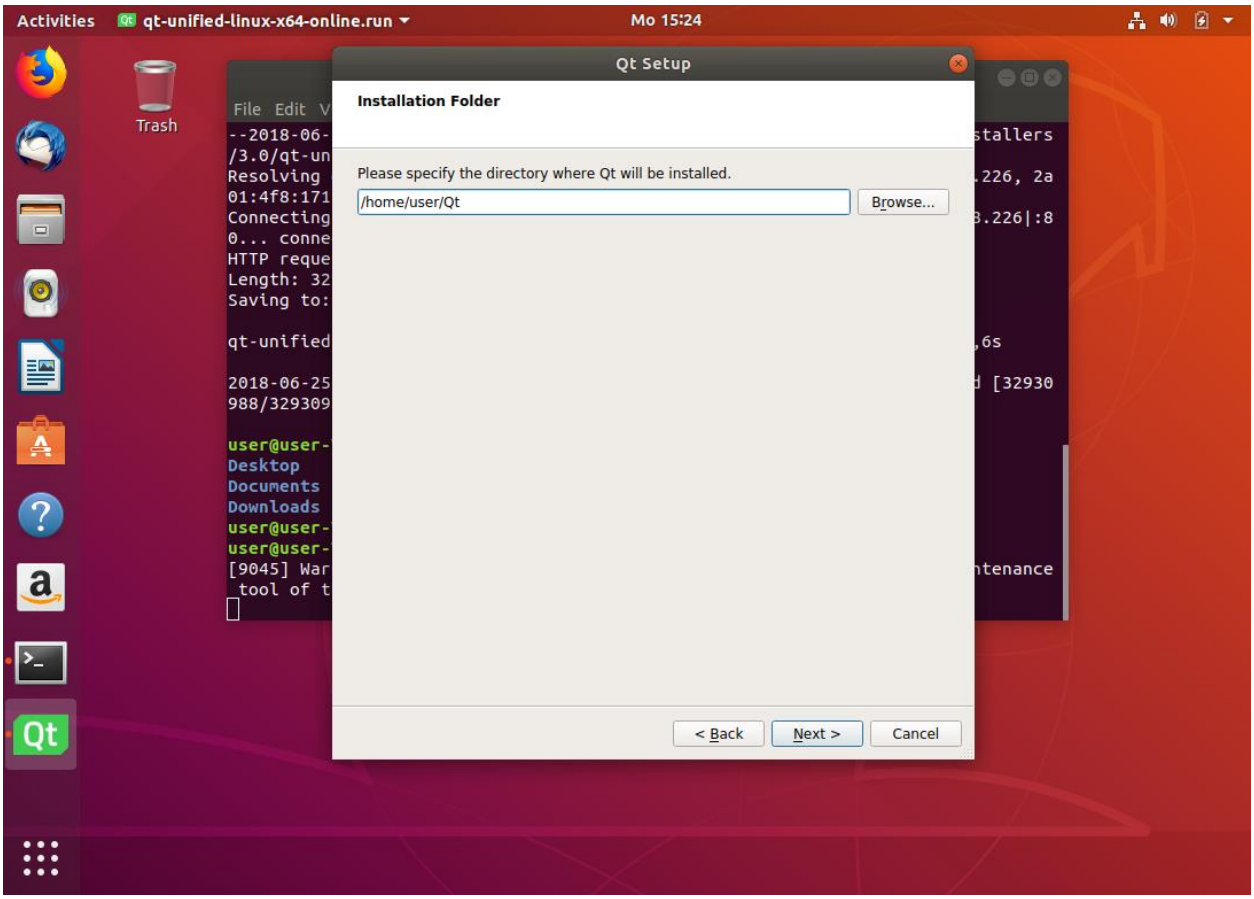




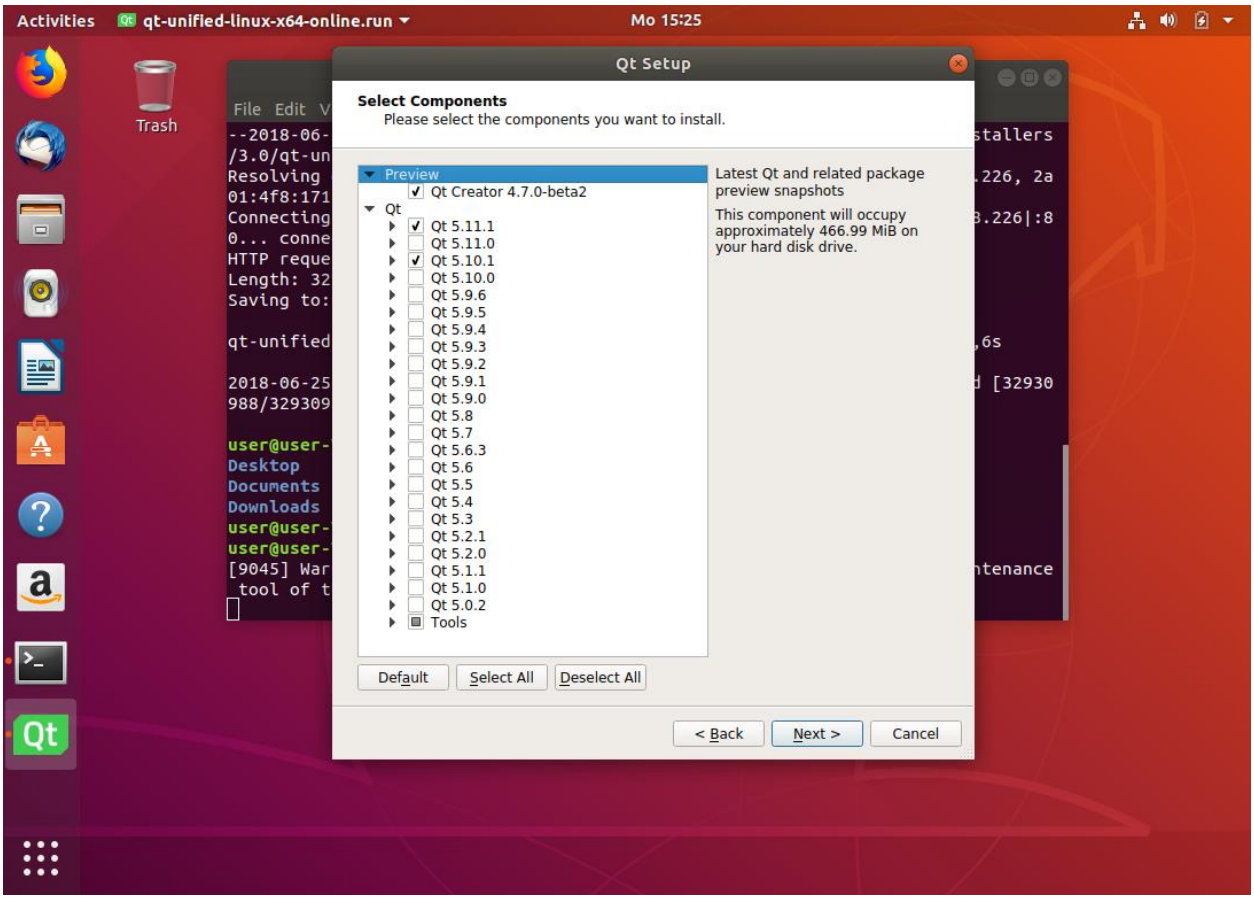
On the permission page check the “Execute” field and close. Now you can run the installer with a double click on it and get the initial window for the installation process.

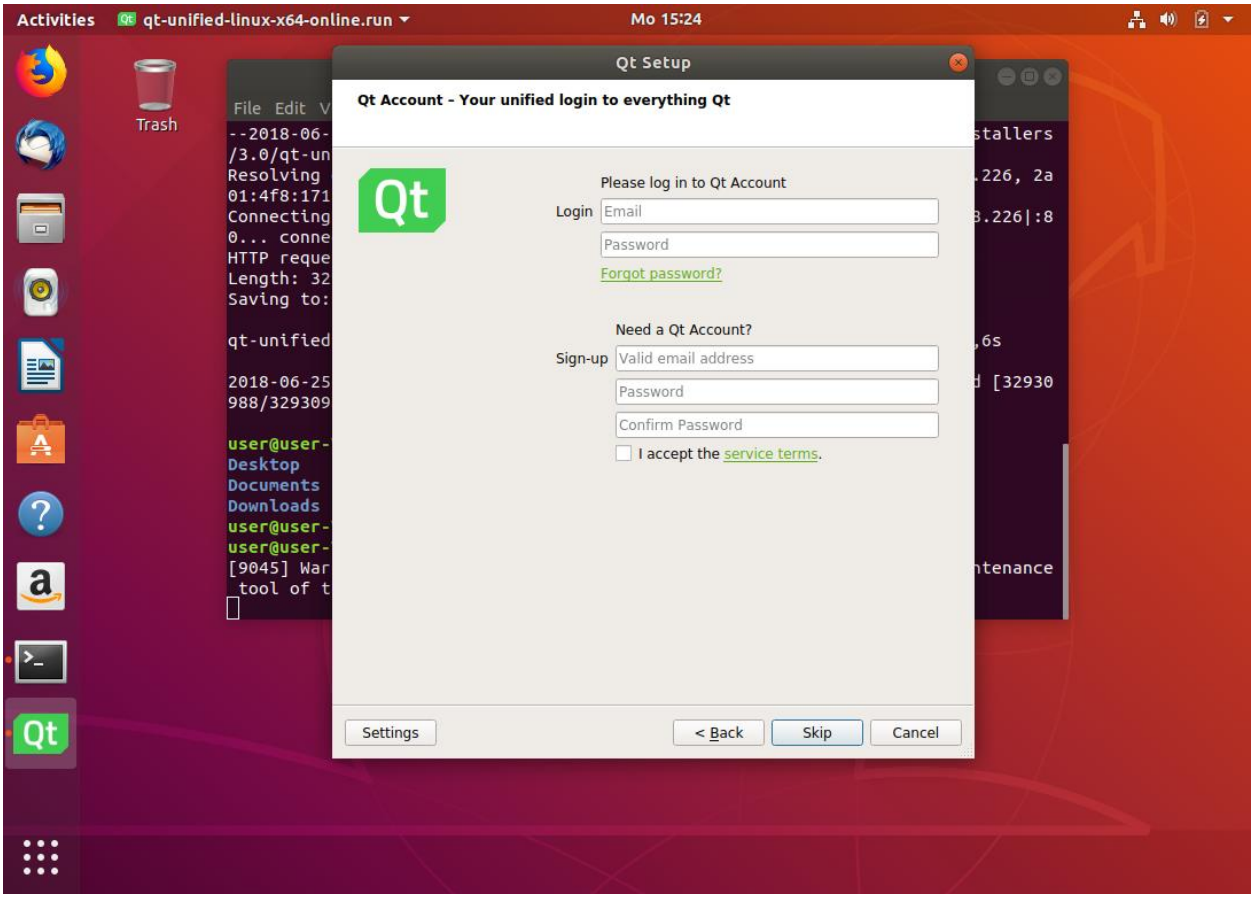

Activities
qt-unified-linux-x64-online.run
Mo 15:25
Trash
Qt Setup
Ready to Install
Setup is now ready to begin installing Qt on your computer. Installation will use 11.09 GiB of disk space.
< Back
Install
Cancel

For the installation we set the path to “/home/user/QT” as we need to go there in the later process.



The next page presens a selection on what to install, including the Qt Versions we want. We take the 5.10.1 and also the 5.11.1.





If you are asked to enter credentials, just skip it and let the installation begin.

Install QT and Cross-Compiler on Ubuntu

To install the tools in Ubuntu, we start with opening a terminal. All following steps will mostly be done on the terminal. If you haven't use the terminal that much, now it's time to do so.

First we need to get a few folders and some data in place. We start with creating a "crosscompile-tools" folder inside the user's home. Type in the terminal "mkdir ~/crosscompile-tools" and press enter. This will be the place where we store the tools we need for the cross compilation. Change to the folder using "cd ~/crosscompile-tools".

Before we begin do in the terminal a "sudo update" and a "sudo upgrade" to keep everything as fesh. After this has been done we start with installing packages required to build the Qt. First we will add this bunch of packages to get most of the Qt features into our build.

We need to install the following:

```
sudo apt-get install libgl1-mesa-dev

sudo apt-get install libxcb-xinerama0-dev

sudo apt-get install build-essential perl python git

sudo apt-get install "^libxcb.*" libx11-xcb-dev libglu1-mesa-dev libxrender-dev libxi-dev

sudo apt-get install flex bison gperf libicu-dev libxslt-dev ruby

sudo apt-get install libssl-dev libxcursor-dev libxcomposite-dev libxdamage-dev libxrandr-dev
libfontconfig1-dev libcap-dev libxtst-dev libpulse-dev libudev-dev libpci-dev libnss3-dev libasound2-dev
libxss-dev libegl1-mesa-dev gperf bison

sudo apt-get install libbz2-dev libgcrypt11-dev libdrm-dev libcups2-dev libatkmm-1.6-dev

sudo apt-get install libasound2-dev libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev

sudo apt-get install libxi6 libxi-dev libxcomposite-dev libxcomposite1

sudo apt-get install libfontconfig1-dev libdbus-1-dev libfreetype6-dev libudev-dev libicu-dev libsqlite3-
dev libxslt1-dev libssl-dev libasound2-dev libavcodec-dev libavformat-dev libswscale-dev
libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev gstreamer-tools gstreamer0.10-plugins-good
libraspberrypi-dev libpulse-dev libx11-dev libglib2.0-dev freetds-dev libsqlite0-dev libpq-dev libiodbc2-
dev firebird-dev libjpeg9-dev libgst-dev libxext-dev libxcb1 libxcb1-dev libx11-xcb1 libx11-xcb-dev libxcb-
keysyms1 libxcb-keysyms1-dev libxcb-image0 libxcb-image0-dev libxcb-shm0 libxcb-shm0-dev libxcb-
icccm4 libxcb-icccm4-dev libxcb-sync1 libxcb-sync-dev libxcb-render-util0 libxcb-render-util0-dev libxcb-
xfixes0-dev libxrender-dev libxcb-shape0-dev libxcb-randr0-dev libxcb-glx0-dev libxi-dev libdrm-dev
libssl-dev libxcb-xinerama0 libxcb-xinerama0-dev
```

sudo apt-get install libatspi-dev

sudo apt-get install libssl-dev libxcursor-dev libxcomposite-dev libxdamage-dev libxrandr-dev libfontconfig1-dev libxss-dev libxtst-dev libpci-dev libcap-dev libsrtp0-dev

sudo apt-get install libxrandr-dev

sudo apt-get install libxtst-dev

sudo apt-get install libcursor-dev

sudo apt-get install libdrm-dev

sudo apt-get install libnss3-dev

sudo apt-get install libx11-xcb-dev libxrender-dev libxext-dev libxcb-render-util0-dev libdbus-1-dev libdirectfb-dev libpulse-dev libaudio-dev libts-dev

Next step is to get the crosscompiler. In yout terminal now type:

git clone https://github.com/raspberrypi/tools.git

We also need something to fix later some symlinks in the image:

wget https://raw.githubusercontent.com/riscv/riscv-poky/master/scripts/sysroot-relativelinks.py

and make it executable

chmod +x sysroot-relativelinks.py./sysroot-relativelinks.py

Here we just run with an outdated compiler. To get a recent one we use the Leonardo 5.5.0, as this is or was the current one during writing this. We will be able to compile Qt 5.11.1 with this.

To get the version just download : https://releases.linaro.org/components/toolchain/binaries/latest-5/arm-linux-gnueabihf/gcc-linaro-5.5.0-2017.10-x86_64_arm-linux-gnueabihf.tar.xz

And extract the content to ~/crosscompile-tools/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/

With the new compiler in place even Qt 5.11.1 is not a problem anymore.

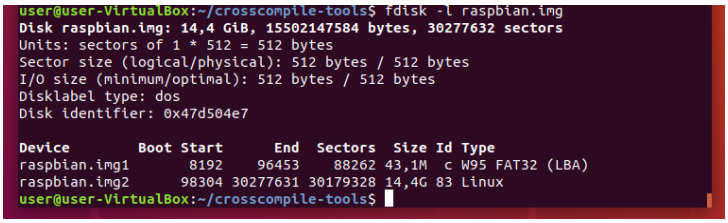
If we are on a 64 Bit OS we also need:

sudo apt-get install lib32z1

This shall be the first part. The next one will be preparing the pi image

Do a “sudo mkdir /mnt/rasp-pi-rootfs”

Next we need some info about the pi image we created earlier. We use fdisk with "fdisk -l raspbian.img". The output shall be like this:



Now we can mount the image to our filesystem. We use "sudo mount raspbian.img -o loop,offset=$((512 * 98304)) /mnt/rasp-pi-rootfs/"

Now we can fix all symbolic links inside the image. Currently they are broken as they are not relative but absolute ones. The phyton script we downloaded will be used to get it back in place.

sudo ./sysroot-relativelinks.py /mnt/rasp-pi-rootfs

If this has done we have to do a bit more. The GLES libarys have ne names in Raspbian stretch. We need to do a "cd /mnt/rasp-pi-rootfs/opt/vc/lib/" and add there a frew symbolic links.
To do this run the following

```
sudo ln -s libbrcmEGL.so libEGL.so
sudo ln -s libbrcmGLESv2.so libGLESv2.so
sudo ln -s libbrcmOpenVG.so libOpenVG.so
sudo ln -s libbrcmWFC.so libWFC.so
```

Now we need to do some exports which will make your life a bit easier:

export RPI_SYSROOT=/mnt/rasp-pi-rootfs

export RPI_TOOLCHAIN=~/crosscompile-tools/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-**x64**/bin/arm-linux-gnueabihf-

If this is done we can go to "~/Qt/5.11.1/Src" to build the current version. Note we build for generic use, meaning the Qt will run from a pi zero to a pi3, but has no optimizations for the newer CPU features found in the pi2 and onwards. If you intend to run on PI2 and PI3 only you need to change the "-device linux-rasp-pi-g++" part from the configure command to "-device linux-rasp-pi2-g++". To optimize only for the PI3 use "-device linux-rasp-pi3-g++".

As we want some hardware acceleration when it comes to drawing we need EGLFS. This is the OpenGL subset for embedded devices like the pi. With the current driver for the vc4 on the pi, we can't get EGL o X up and running. So you are limited to X11 without any hardware acceleration, or you use EGLFS and work completely without the X11.

To tell that we want this in we run the following command inside the "~/Qt/5.11.1/Src" directory within a terminal:

```
./configure -opengl es2 -qt-xcb -xcb -device linux-rasp-pi-g++ -device-option
CROSS_COMPILE=$RPI_TOOLCHAIN -sysroot $RPI_SYSROOT -opensource -confirm-license -optimized-
qmake -reduce-exports -release -make libs -prefix /usr/local/qt5pi -hostprefix /usr/local/qt5pi –v
```

If we use the Leonardo 5.5.0 toolchain we need to add

-no-use-gold-linker. This results in :

```
./configure -opengl es2 -qt-xcb -xcb -device linux-rasp-pi-g++ -device-option
CROSS_COMPILE=$RPI_TOOLCHAIN -sysroot $RPI_SYSROOT -opensource -confirm-license -no-use-gold-
linker -optimized-qmake -reduce-exports -release -make libs -prefix /usr/local/qt5pi -hostprefix
/usr/local/qt5pi -v
```

to the commandlist. If we want to build **for X11** we need the following line:

```
./configure -release -no-eglfs -qt-xcb -xcb -nomake tests -nomake examples -skip qtscript -skip
qtwebengine -no-use-gold-linker -device linux-rasp-pi2-g++ -device-option
CROSS_COMPILE=$RPI_TOOLCHAIN -sysroot $RPI_SYSROOT -opensource -confirm-license -make libs -
prefix /usr/local/qt5pi -hostprefix /usr/local/qt5pi –v
```

This will configure the Qt for our target, but we will miss some features we currently don't require. The process will take a few minutes. Afterwards the Qt is ready to be compiled for our target.

Remark: As we have set up the system to use all cores we have, this still need some time.

In the terminal now type "make –jX" where X is the amount of cores we set to the VM. In the authors case this was "make –j3" to use the three cores the vm had.

If the process ends without errors you are ready to install the fresh compiled Qt to the SD-Card image. Do a "sudo make install" and things will start running. After a while if no errors occurred your image is read to be written to a sd-card. We now write the image to the SD-Card.

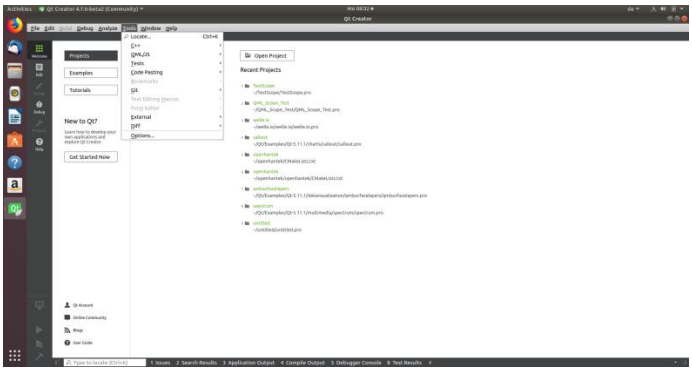
With this the first part Is done and you have now a working Qt on your Pi.

Setup for cross development

You may ask how to develop your own apps. We first need to add a package to our Ubuntu. As the Raspberry Pi is not an X86 as our virtual machine, we need to install the gdb-multiarch. Do this by opening a terminal and type “sudo apt-get install gdb-multiarch”. We will need this later to setup the debugging for the pi.

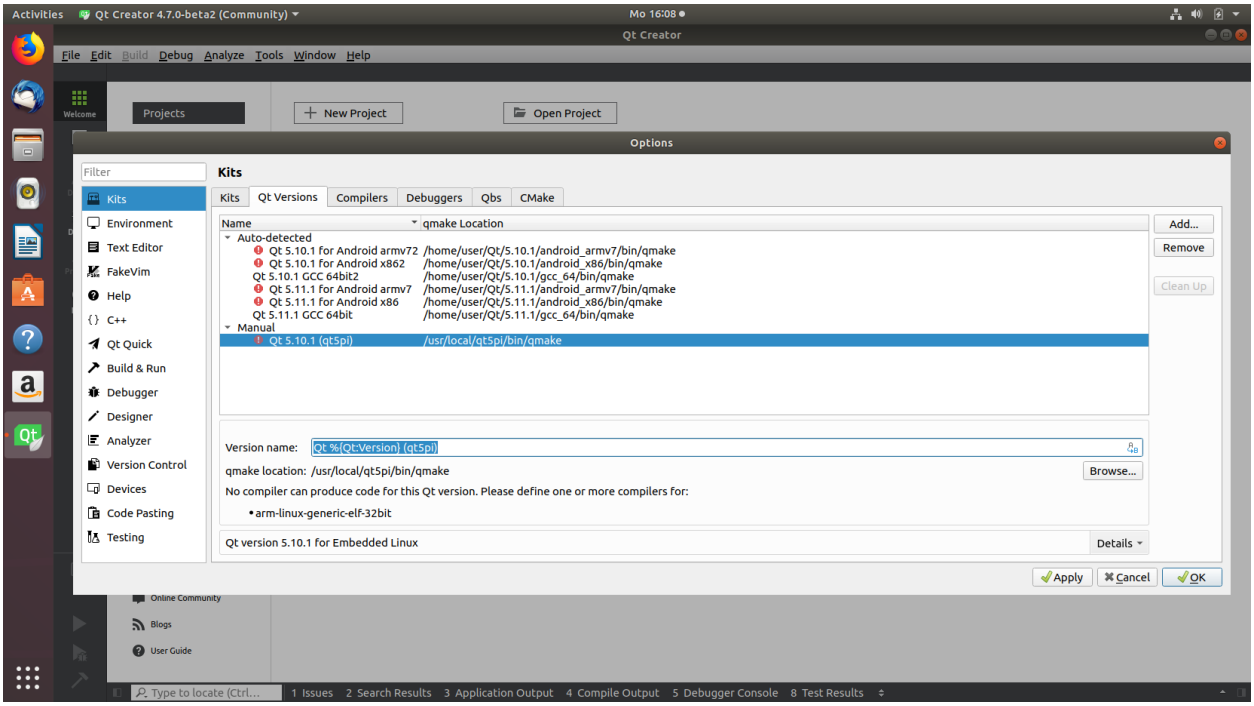
For app develoment, you need to setup the QT Creator for cross-development. To do so, start the Creator by a click on “Activities”. In the search field type “qtcre” and an green icon with “Qt Creator (Community) “ will appear.

Click on it to start the Qt Creator. After loading we need to setup the environment for the PI. To do this, we need to create a new Kit. This will be done here under “Tools -> Options”

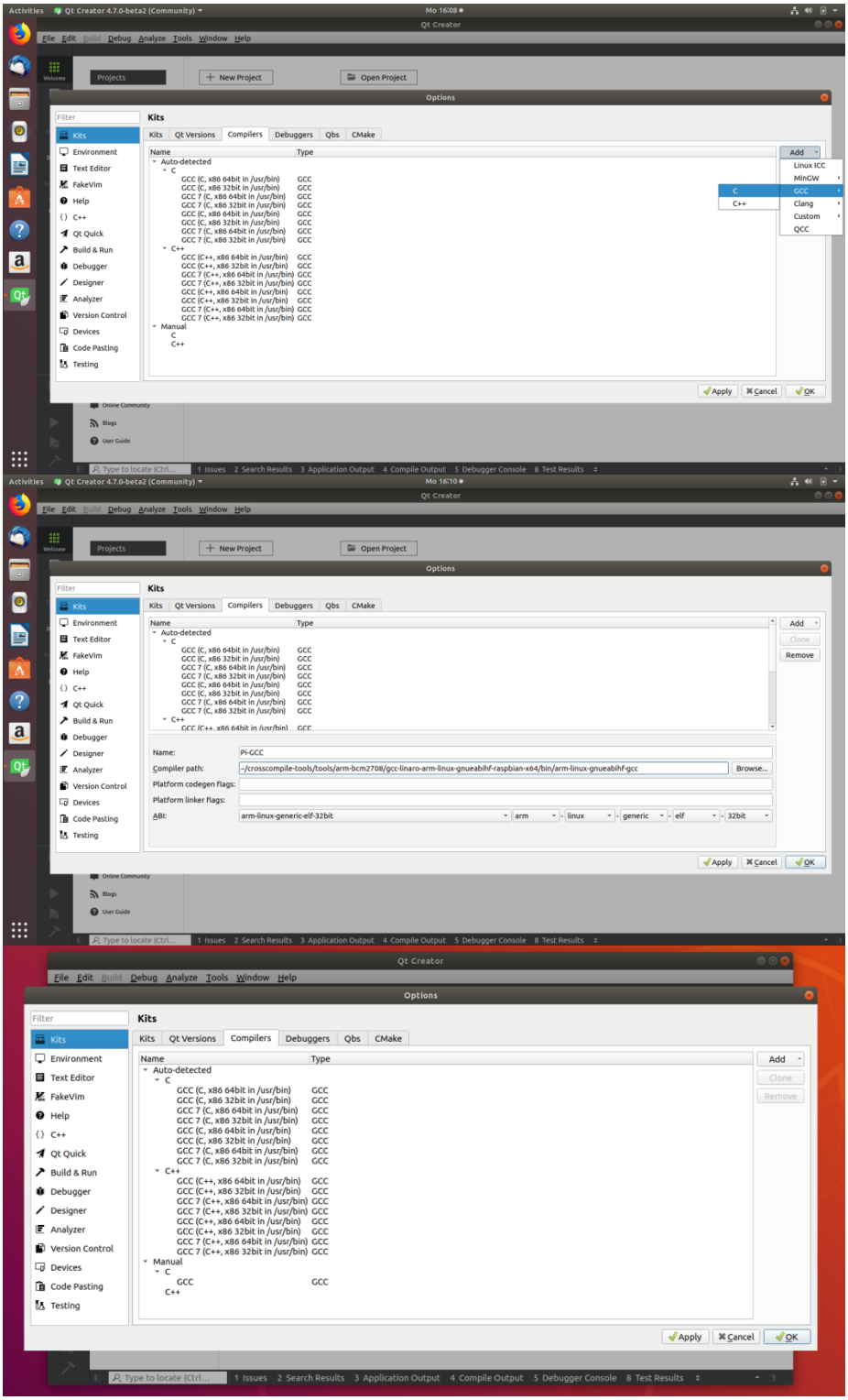


Now you can use the “Kits” to setup the Raspberry as a new Kit for use.

We now add the Qt-version we just build. Depending on yout project the may be 5.10.1 or 5.11.1 as those are currently the recent ones while writing this manual. Select “Kist” and click on “Add..” to add a new kit. You will be shown a filebrowser, you use to navigate to “/usr/local/qt5pi/bin/” and select the qmake file. Confirm your choise with “open”, and the version we just build, will be added. But we will see a red exclamationmark on this version. This is due to the fact that we have not set up a compiler in the Qt creator for the Raspberry Pi.

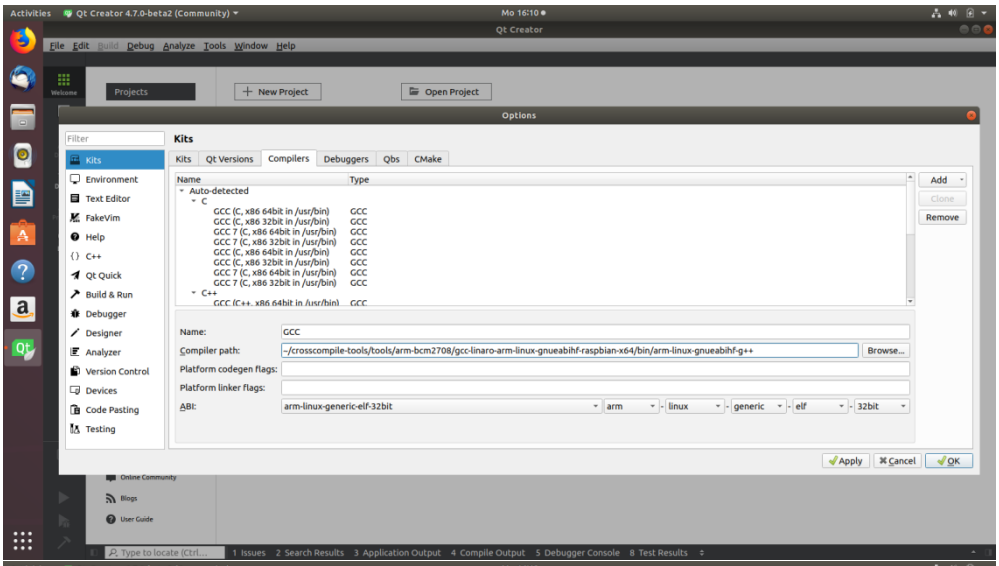


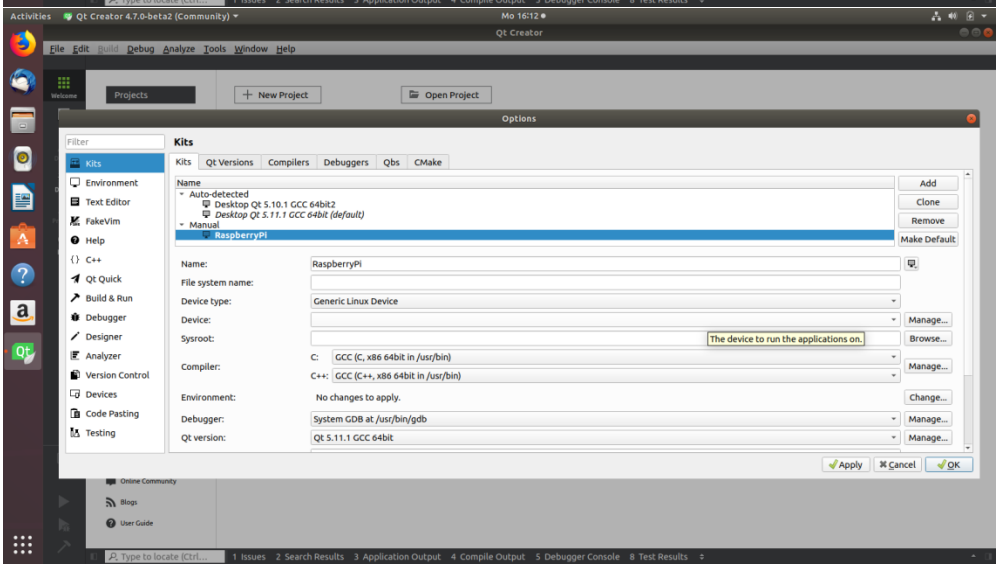
Next thing we do, is to setup the compiler. We need one for “C” and “C++”. As the Compiler won’t be autodetected, we add them under the “Manual” section. First we add the “C”-compiler. Click on “Add->GCC->C” and a new entry for the compiler will apprear.



You now need to specify a few things. First we change the name here from GCC to Pi-GCC and use "~/crosscompile-tools/tools/arm-bcm2708/gcc-lionardo-arm-linux-gnueabihf-raspbian-x64/bin/amr-

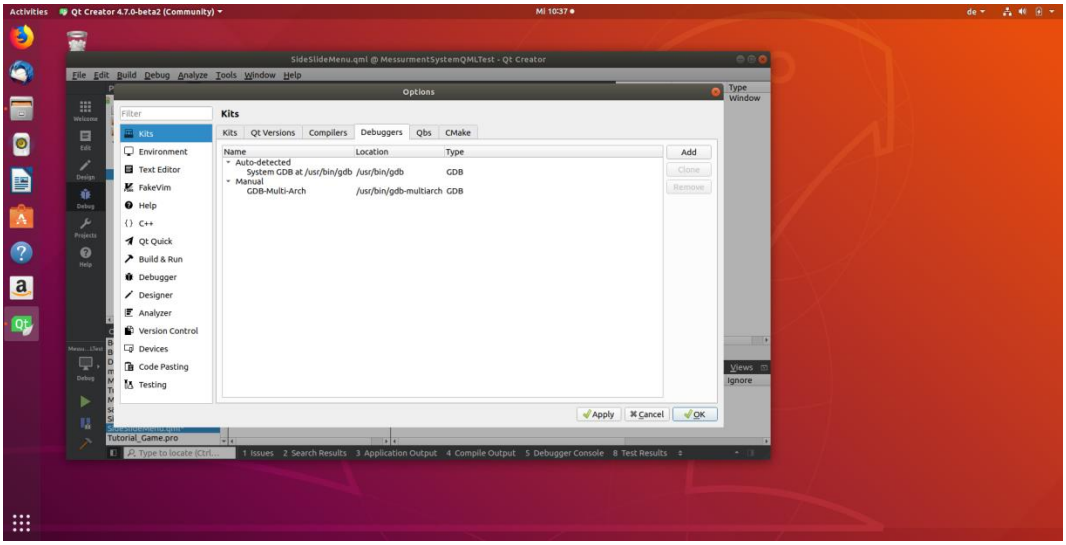
linux-gnueabihg-gcc” for the compiler. All other settings should be fine for now. Click on “Apply”. The next step is to add in the same manner the “C++”-compiler.





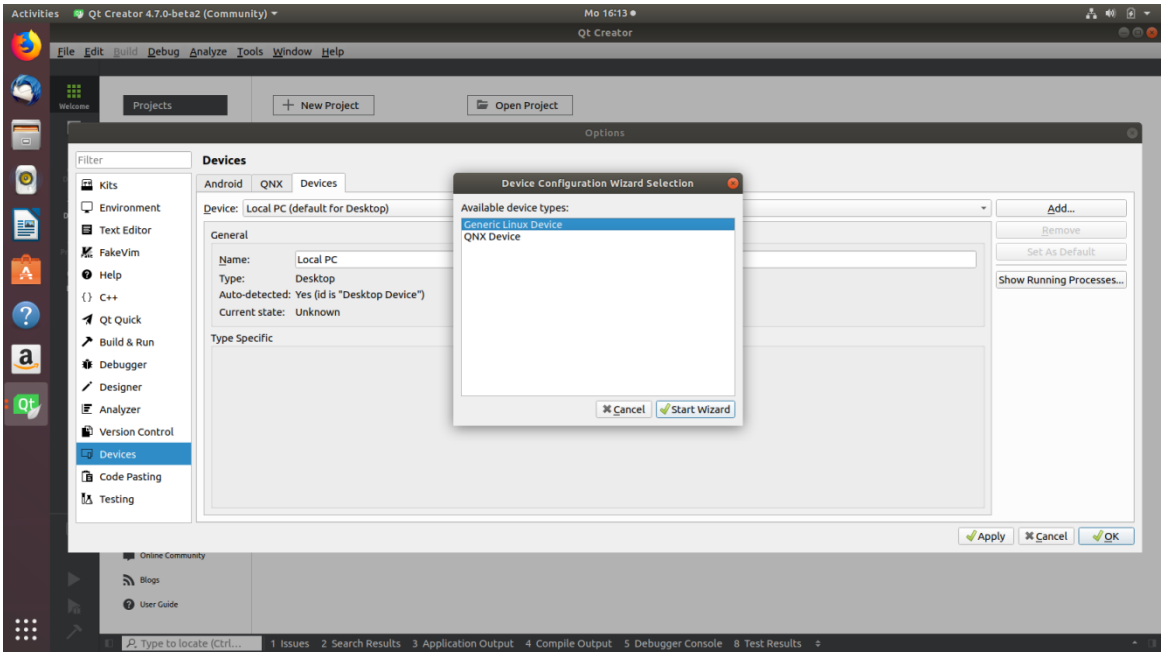
Use “Add->GCC->C++” to add the g++. Use a suitable name like “Pi-GPP”. As “Compiler path” use “~/crosscompile-tools/tools/arm-bcm2708/gcc-lionardo-arm-linux-gnueabihf-raspbian-x64/bin/amr-linux-gnueabihg-g++” and leave the rest as it is.

The next part is to setup the debugger. We use the “Debuggers” tab. We need to now to setup the gdb-multiarch. For this click on “Add”.

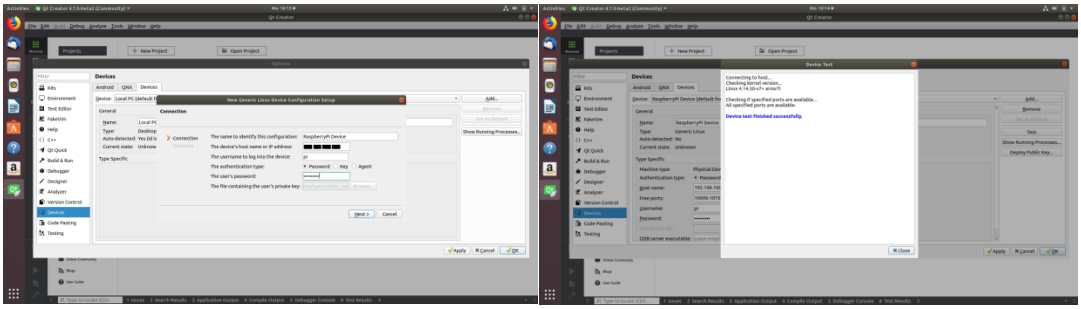


As name we use “GDB-Multi-Arch” and as path we need to set “/usr/bin/gdb-multiarch”. We confirm the settings with “Apply”

We now need to setup a new device, we want to use. In this case we click on “Devices”. You will be shown a few auto detected ones, most likely your local pc. As your Raspberry Pi runs a linux we need to add a new linux device.

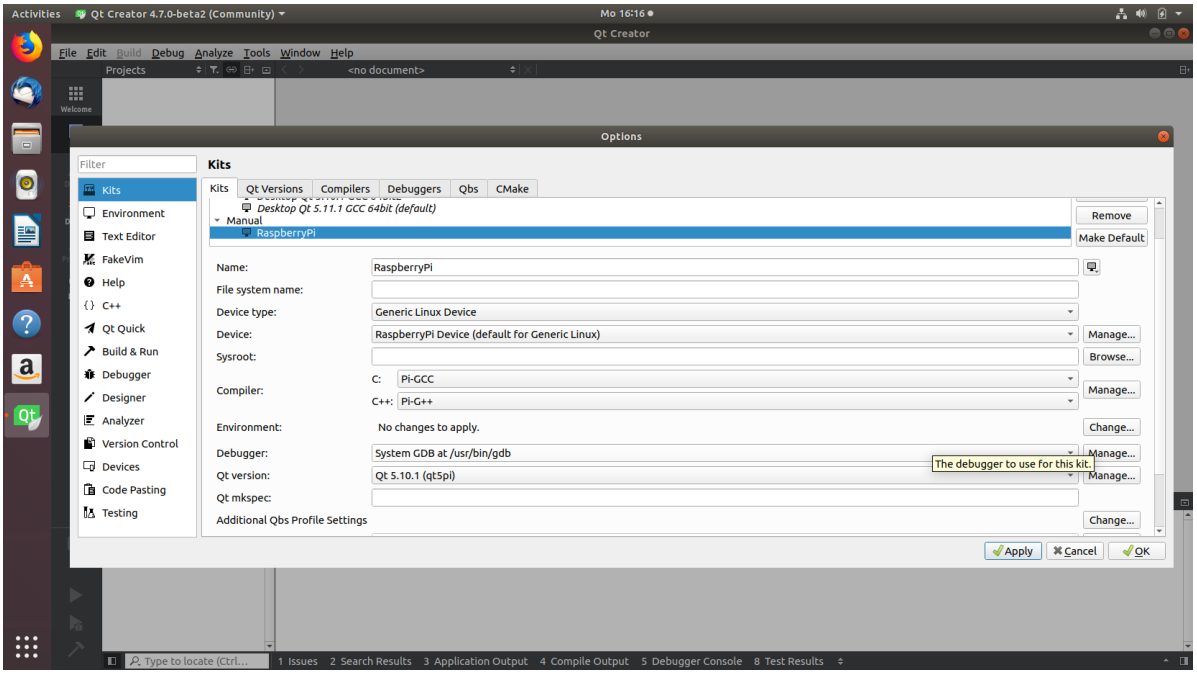


Click on the “Generic Linux Device” and continue using the “Start Wizzard”. You will be asked a few questions about the raspberry, you want to add.



Tye in a Devicename, here we choose “RaspberryPi Device” here to identify it later with easy. Also not you need to add the pi’s ip and give the passwort for the ssh login of our user pi. Click on “Next” to star the connectiontest.

We have now assembled the parts to define a KIT. To do so we need to go to KIT and click to “Add” on the right side. A new KIT will appear and you need to provide some information. First this KIT needs a name, here we use “RaspberryPi”.



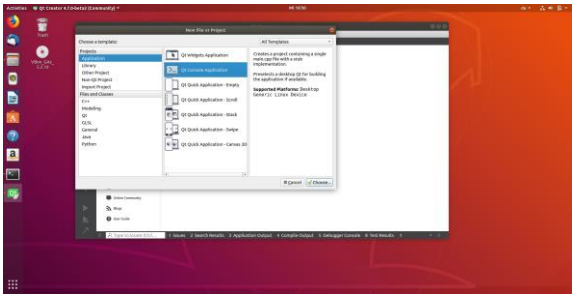
We now need to supply a few things we added before. First we choose as Device type “Generic Linux Device”. The “RaspberryPi Device” needs now to be chosen under device. Also we need to provide the “C” and “C++” compiler we added under the “Compiler” setting. For the debugger we need to choose the “GDB-Multi-Arch” we defined before.

Finally we choose the “Qt version” we build for our pi and added. We finaly click on Apply an can now consider the Qt Creator as ready to develop. Last but not least we will show you how to buid your first App on the System for the Raspberry PI.

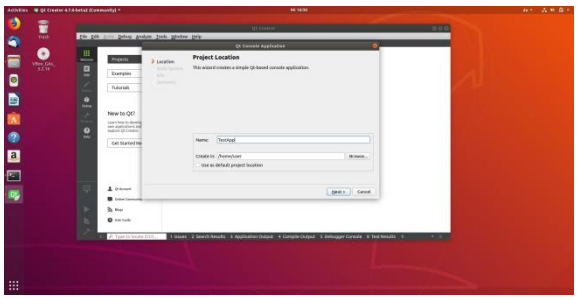
"Hello World"

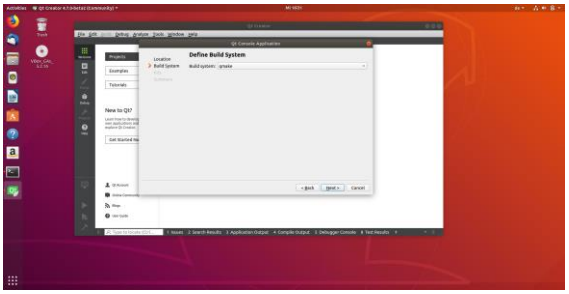
We now start our first project. We could do a simple console output, but with the QT framework at hand this might be a bit boring if you know C / C++, so we use a timer to print every second a part of the "99 Bottles of Beer". The code will not be perfect but give you an idea what you can do.

In the QT Creator select "File->New File or Project"

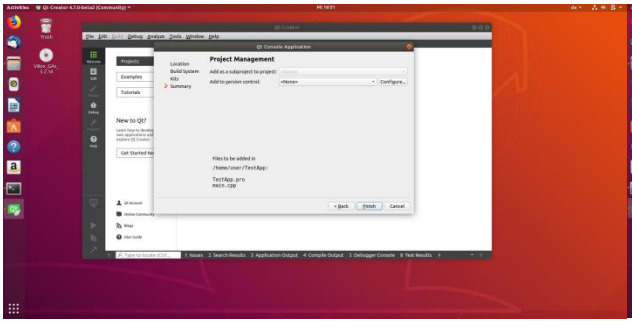


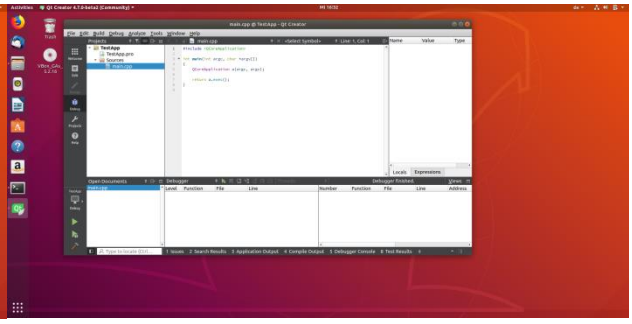
We now choose "Qt Console Application" and will then need to give the Project a name and a Path where we save it. We call it here "TestApp" and leave the path unchanged.





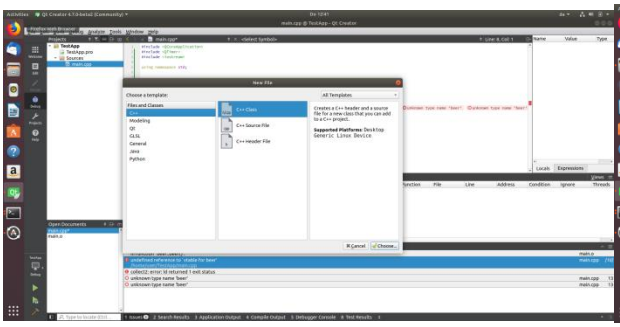
Also we leave the Build System on qmake. The next step is to choose which kits we want to use for the application. For our ease we use The Desktop QT 5.11.1 GCC 64bit and the Raspberry PI one. So we can build our App for the Pi and our desktop system.

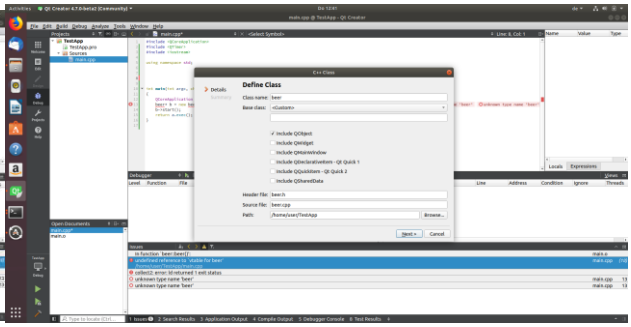




As last step we sys "Finish" and have our project setup. Afterwards we will be back in the editor and show the a basic QT application we now going to add some code to.

We build a new class for “beer” and to a left click on “TestApp”. In the context menu we choose “Ad New…” and a dialog will pop up.





We select “C++ Class” and the choose “include QObject” and base Class “<Custom>” and finish the selection. You will now have a new bare class “beer” in the QT environment. For the main.cpp we use the following code at the end:

```
#include <QCoreApplication>
#include "beer.h"

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    beer* b = new beer();
    b->start();
    return a.exec();
}
```

For the beer.h

```
#ifndef BEER_H
#define BEER_H

#include <QObject>

class beer: QObject
{
    Q_OBJECT
public:
    beer();
    void start( void );
public slots:
    void _99bottelsofBeer( void );
};


#endif // BEER_H
```

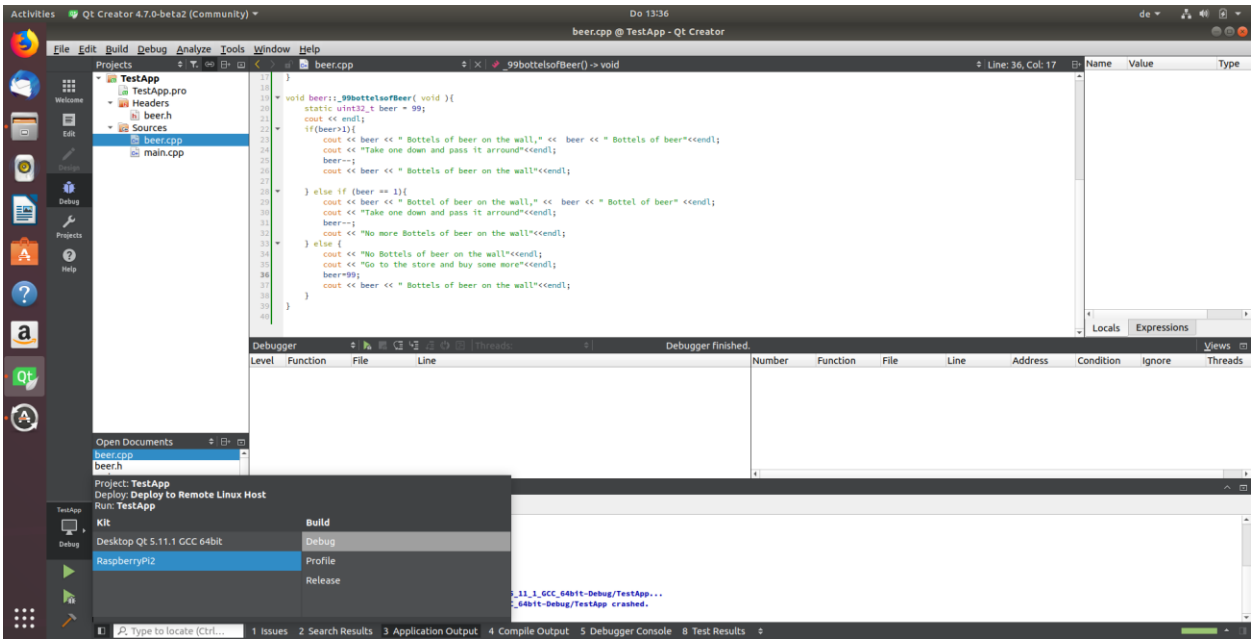
In the beer.cpp we use the following code:

```
#include "beer.h"
#include <QTimer>
#include <iostream>
```

```
using namespace std;
beer::beer()
{

}

void beer::start(){
    QTimer *timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(_99bottelsofBeer()));
    timer->setInterval(1000);
    timer->start();
}

void beer::_99bottelsofBeer( void ){
    static uint32_t beer = 99;
    cout << endl;
    if(beer>1){
        cout << beer << " Bottels of beer on the wall," <<  beer << " Bottels of beer"<<endl;
        cout << "Take one down and pass it arround"<<endl;
        beer--;
        cout << beer << " Bottels of beer on the wall"<<endl;

    } else if (beer == 1){
        cout << beer << " Bottel of beer on the wall," <<  beer << " Bottel of beer" <<endl;
        cout << "Take one down and pass it arround"<<endl;
        beer--;
        cout << "No more Bottels of beer on the wall"<<endl;
    } else {
        cout << "No Bottels of beer on the wall"<<endl;
        cout << "Go to the store and buy some more"<<endl;
        beer=99;
        cout << beer << " Bottels of beer on the wall"<<endl;
    }
}
```

We can now do a test on the pc by pressing the green run arrow. This will open a console and run our program.

To use it now on the pi, we need to change the kit.

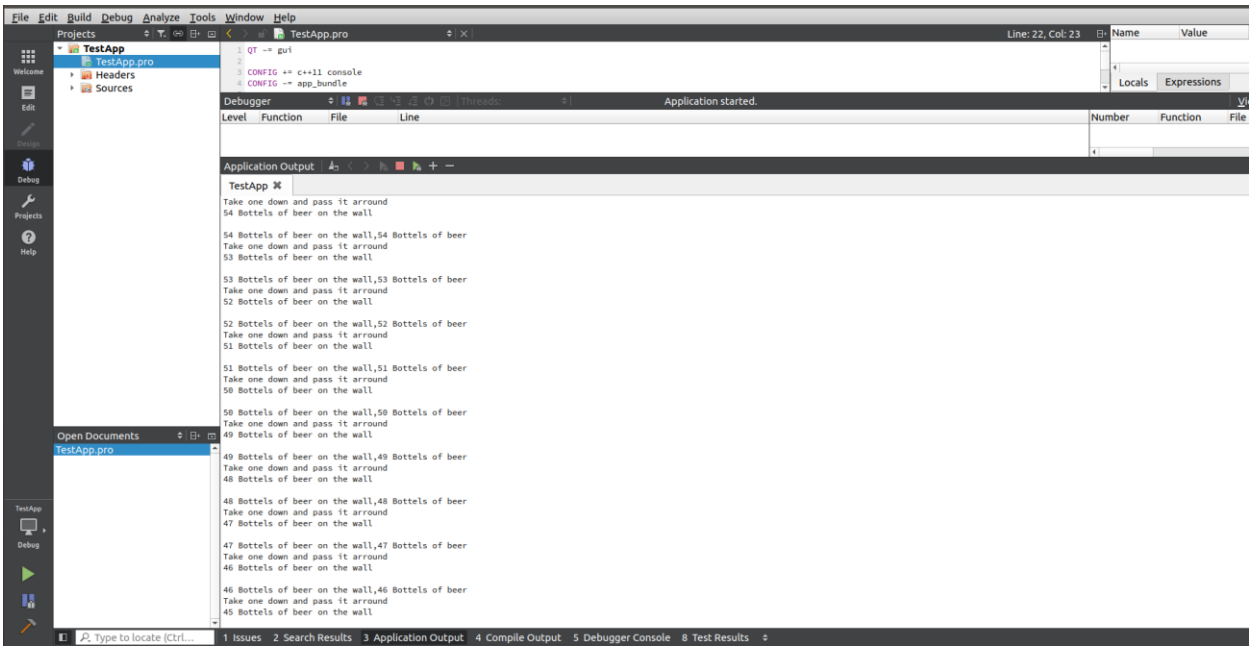


To get the Files on the PI at the right place we need some modification to the Test.pro file. This will change the target directory to the home of the user pi.

The .pro file looks now like this:

```
QT -= gui

CONFIG += c++11 console

CONFIG -= app_bundle

DEFINES += QT_DEPRECATED_WARNINGS

#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0

SOURCES += \

        main.cpp \

    beer.cpp

target.path = /home/pi

INSTALLS += target

HEADERS += \

    beer.h
```

If we now press the debug symbol (green arrow wit bug), the code gets executed o the pi. You won't see a output on the ui of the pi, but if you look at the debug window in the QT Creator you will see some output.



This means your application is running. From here on it's now on you to study some QT manuals or books.