Cross-Compile Qt 6 for Raspberry Pi

Introduction

Contents

This is a step-by-step guide to cross-compile Qt 6 for **Raspberry Pi OS.** The end result of this guide allows you to compile Qt 6 applications on a host machine and deploy it on the Raspberry Pi running a Raspberry Pi OS image. The instructions in this guide are targeted for beginners, but it should be easy to follow for everyone.

The steps on this guide was tested on **Ubuntu 20.04** targeting **Raspberry Pi 4 Model B 2GB (64-bit)** running Raspberry Pi OS, but it should work with any Debian-based OS.

For an older guide, check this Qt Wiki guide (https://wiki.qt.io/Raspberry_Pi_Beginners_Guide) or this one (https://wiki.qt.io/RaspberryPi2EGLFS).

Getting Started: Planning Phase

Before we get our hands dirty and mess around with the configurations, let's familiarise ourselves with essential components needed to cross-compile Qt and decide which directory names we want to use to store these components. This way, we can easily track the location of these important components when we are typing commands in the terminal.

At its most basic setup, we need **at least the following components** in a host machine to cross-compile Qt:

- **Sysroot,** which basically is the scaled down version of our target's filesystem inside our host machine. For this example, we will name this

  ```
  ~/rpi-sysroot
  ```

- **A cross-compiler on host machine.** To make it as straightforward as possible, for this guide we will use the cross-compiler for Aarch64 **from the official Ubuntu package repository**.
- **Qt source code with its submodules, at least the qtbase submodule.** For this guide, we will download Qt source code from Qt official GitHub repository. By default, it will be stored in

  ```
  ~/qt5
  ```

  .
- **CMake toolchain.** CMake is the recommended build generator for Qt 6. Inside this toolchain file, we will specify in detail about the exact specifications of our cross-compiled Qt build. We will store this

  ```
  toolchain.cmake
  ```

  file on our

  ```
  $HOME
  ```

  directory.

We also have to decide **the locations of our Qt builds**, one for host machine and one for our Raspberry Pi

in our PC before we transfer it to the Raspberry Pi.

- We will store **a build of Qt 6 for host machine** on

  ```
  ~/qt-host
  ```

  . This is necessary because when building a Qt for our Raspberry Pi target, we will refer to this build of Qt.

- Finally, our **Qt 6 build for Raspberry Pi** will be stored on

  ```
  ~/qt-raspi
  ```

  in the host machine temporarily before we move it to a location in our Raspberry Pi (for example,

  ```
  /usr/local/qt6
  ```

).

Following the best practices of building C/C++ applications, we also want to make sure that our final build directories are clean from all build configuration files that CMake will create during the configuration phase. For that, we will store **build configuration files** on

```
~/qt-hostbuild
```

for configuration files of Qt for our host machine and

```
~/qtpi-build
```

for configuration files of Qt for Raspberry Pi.

After we decided these locations, let's create them all!

On host machine,

```
$ cd ~
$ mkdir rpi-sysroot rpi-sysroot/usr rpi-sysroot/opt
$ mkdir qt-host qt-raspi qt-hostbuild qtpi-build
```

The second line will create the sysroot directory with its necessary subfolder, while the third line will create the directory for the rest of locations defined previously.

Installing Raspberry Pi OS

Let's now download and install/flash an image of Raspberry Pi OS to a microSD card. The easiest way to do this is to use the official Raspberry Pi Imager. You can download the latest Imager from https://www.raspberrypi.com/software/, and then flash an image of Raspberry Pi OS to your microSD card with

only a few clicks on the Imager's GUI.

Setting up Raspberry Pi

After successfully flashing an image of Raspberry Pi OS to a microSD card, insert it into the microSD card reader of our Raspberry Pi, connect a keyboard and monitor to the device and boot it up. The system will ask for a basic first-time setup. Please follow the setup according to the instruction.

After successfully setting up the device, the system will reboot to the home desktop. From here, open Terminal. We want to set our Raspberry Pi to be able to run Qt apps.

Install Dependencies

Update and upgrade to the newest version.

```
$ sudo apt update
$ sudo apt full-upgrade
$ sudo reboot
```

Next, install the package dependencies.

```
$ sudo apt-get install libboost-all-dev libudev-dev libinput-dev libts-dev libmtdev-dev libjpeg-dev libfontconfig1-dev libssl-dev libdbus-1-dev libglib2.0-dev libxkbcommon-dev libegl1-mesa-dev libgbm-dev libgles2-mesa-dev mesa-common-dev libasound2-dev libpulse-dev gstreamer1.0-omx libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev gstreamer1.0-alsa libvpx-dev libsrtp2-dev libsnappy-dev libnss3-dev "^libxcb.*" flex bison libxslt-dev ruby gperf libbz2-dev libcups2-dev libatkmm-1.6-dev libxi6 libxcomposite1 libfreetype6-dev libicu-dev libsqlite3-dev libxslt1-dev

$ sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libx11-dev freetds-dev libsqlite3-dev libpq-dev libiodbc2-dev firebird-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 libxcb-xinerama0 libxcb-xinerama0-dev libatspi2.0-dev libxcursor-dev libxcomposite-dev libxdamage-dev libxss-dev libxtst-dev libpci-dev libcap-dev libxrandr-dev libdirectfb-dev libaudio-dev libxkbcommon-x11-dev
```

This will install a bunch of libraries from the Debian repository that Qt needed to deploy various kinds of applications on Raspberry Pi.

We also will make a directory that will hold the Qt installation targeting Raspberry Pi, as specified in the previous section. Actually, this can be done at every time before sending Qt 6 build from host machine to the Raspberry Pi, but in this guide we will create it now:

```
$ sudo mkdir /usr/local/qt6
```

Setting up Host Machine

Let's leave the Raspberry Pi for a moment and move to our host machine. As mentioned in the Introduction section, the steps below was tested for **Ubuntu 20.04** running on a host machine with **x86_64** architecture, but it should works with any Debian-based distributions running on x86_64

architecture.

Update System

Let's configure our Ubuntu/Debian host machine. First, let's update our packages to the latest version.

```
$ sudo apt update
$ sudo apt upgrade
```

Install Qt Dependencies

Next, we will install some more package dependencies. Some of these packages are build tools needed to compile Qt 6 that you might or might not have.

```
$ sudo apt-get install make cmake build-essential libclang-dev clang ninja-build gcc git bison python3 gperf pkg-
config libfontconfig1-dev libfreetype6-dev libx11-dev libx11-xcb-dev libxext-dev libxfixes-dev libxi-dev
libxrender-dev libxcb1-dev libxcb-glx0-dev libxcb-keysyms1-dev libxcb-image0-dev libxcb-shm0-dev libxcb-icccm4-dev
libxcb-sync-dev libxcb-xfixes0-dev libxcb-shape0-dev libxcb-randr0-dev libxcb-render-util0-dev libxcb-util-dev
libxcb-xinerama0-dev libxcb-xkb-dev libxkbcommon-dev libxkbcommon-x11-dev libatspi2.0-dev libgl1-mesa-dev libglu1-
mesa-dev freeglut3-dev
```

Install Cross Compiler

Next, let's get our cross-compiler. For this guide, we will install it from the Ubuntu/Debian package repository as this is the easiest way to get an ARM64 cross-compiler.

```
$ sudo apt install gcc-aarch64-linux-gnu g++-aarch64-linux-gnu
```

You can also download the cross-compiler from somewhere on the internet. If you want to do so, my recommendation is to check the ARM official download page (https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/downloads) and choose which one fits your purpose.

Setting up SSH Connection

We also want to setup an SSH connection between Raspberry Pi and a host machine. This is needed to transfer data between both devices. The official Raspberry Pi documentation page (https://www.raspberrypi.com/documentation/computers/remote-access.html#ssh) provided a very detailed and thorough instruction on how to setup SSH between Raspberry Pi and other device, so you can refer this step into that page.

Building Sysroot from Device

We need to build a Sysroot from our own Raspberry Pi device in the host machine. To do this, we will copy and paste a few directories from Raspberry Pi using rsync through SSH. If rsync is not installed yet in your host machine, simply install it by executing

```
sudo apt install rsync
```

. After the host machine and Raspberry Pi is connected,

```
$ cd $HOME
$ rsync -avzS --rsync-path="rsync" --delete <pi_username>@<pi_ip_address>:/lib/* rpi-sysroot/lib
$ mkdir $HOME/rpi-sysroot/usr
$ rsync -avzS --rsync-path="rsync" --delete <pi_username>@<pi_ip_address>:/usr/include/* rpi-sysroot/usr/include
$ rsync -avzS --rsync-path="rsync" --delete <pi_username>@<pi_ip_address>:/usr/lib/* rpi-sysroot/usr/lib
$ mkdir $HOME/rpi-sysroot/opt
$ rsync -avzS --rsync-path="rsync" --delete <pi_username>@<pi_ip_address>:/opt/vc rpi-sysroot/opt/vc
```

Change

```
<pi_username>
```

and

```
<pi_ip_address>
```

with your Raspberry Pi username and IP Address, respectively.

Quick explanation:

This line uses

```
rsync
```

to copy all the files and subdirectories in the

```
/lib
```

directory on the Raspberry Pi to a local directory called

```
rpi-sysroot/lib
```

. The

```
-a
```

option preserves file permissions, ownership, and timestamps,

```
-v
```

enables verbose output,

```
-z
```

compresses the data during transfer,

```
-S
```

optimizes data transfer for sparse files, and

```
--delete
```

removes files from the destination that are no longer present on the source. The

```
--rsync-path
```

option is used to specify the path to the

```
rsync
```

executable on the Raspberry Pi. **Note:** Your Raspberry Pi might not have a directory named

```
/opt/vc
```

, and it is fine. Usually this directory contains proprietary Broadcom libraries, but during the testing the author did not find any issue with the lack of this directory. After building our own sysroot, we need to fix absolute symbolic links that were broken during the syncing process by converting them to relative symlinks. The easiest, fastest way to do this is to use a command-line tool called

```
symlinks
```

, which is available from the Ubuntu official repository.

```
$ sudo apt install symlinks
$ cd ~
$ symlinks -rc rpi-sysroot
```

Check the symlinks documentation (https://linux.die.net/man/8/symlinks) to learn more.

Note: Make sure that there are no errors in the terminal. You may need to use **sudo** here.

Building Qt 6

After all the setup process, finally we can start building Qt 6! This will be divided into two parts: Building Qt 6 for host machine, and building Qt 6 for our target device (i.e. the Raspberry Pi).

Building Qt 6 for Host Machine

```
cd $HOME
git clone "https://codereview.qt-project.org/qt/qt5"
cd qt5/
git checkout 6.4.0
perl init-repository -f
cd ..
mkdir $HOME/qt-hostbuild
cd $HOME/qt-hostbuild/
cmake ../qt5/ -GNinja -DCMAKE_BUILD_TYPE=Release -DQT_BUILD_EXAMPLES=OFF -DQT_BUILD_TESTS=OFF -
DCMAKE_INSTALL_PREFIX=$HOME/qt-host
cmake --build . --parallel 8
cmake --install .
```

One thing to remember is that, in this guide, we will install Qt on folder

```
~/qt-host
```

and we will store all build configuration files on

```
~/qt-hostbuild
```

. Notice that you can also choose to install only Qt submodules that you need. This is good if you have a limited space in your host machine. At minimum, you only need the

```
qtbase
```

submodule to be able to develop and run a simple Qt Widget application. But to access other Qt features you need to install other or all Qt submodules.

Building Qt 6 for Target Device

This is the interesting part. We will compile Qt 6 on a host machine and install it to the Raspberry Pi. The instruction for configuring and building are similar to building Qt 6 for host machine, but there are a few things we need to do to get the intended results.

Create a Toolchain File

First, we want to create our own CMake toolchain file. This file is important to link the right compilers and libraries so that CMake can produces the suitable Qt build for Raspberry Pi. For this guide, we will use nano as the editor for the following CMake code and save it as toolchain make in home directory

```
cd ~
nano toolchain.cmake
```

.

```
cmake_minimum_required(VERSION 3.18)
```

```
include_guard(GLOBAL)

set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(TARGET_SYSROOT /path/to/your/sysroot)
set(CMAKE_SYSROOT ${TARGET_SYSROOT})

set(ENV{PKG_CONFIG_PATH} $PKG_CONFIG_PATH:/usr/lib/aarch64-linux-gnu/pkgconfig)
set(ENV{PKG_CONFIG_LIBDIR} /usr/lib/pkgconfig:/usr/share/pkgconfig/:${TARGET_SYSROOT}/usr/lib/aarch64-linux-gnu/
pkgconfig:${TARGET_SYSROOT}/usr/lib/pkgconfig)
set(ENV{PKG_CONFIG_SYSROOT_DIR} ${CMAKE_SYSROOT})

# if you use other version of gcc and g++ than gcc/g++ 9, you must change the following variables
set(CMAKE_C_COMPILER /usr/bin/aarch64-linux-gnu-gcc-9)
set(CMAKE_CXX_COMPILER /usr/bin/aarch64-linux-gnu-g++-9)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -I${TARGET_SYSROOT}/usr/include")
set(CMAKE_CXX_FLAGS "${CMAKE_C_FLAGS}")

set(QT_COMPILER_FLAGS "-march=armv8-a")
set(QT_COMPILER_FLAGS_RELEASE "-O2 -pipe")
set(QT_LINKER_FLAGS "-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
set(CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE)
set(CMAKE_BUILD_RPATH ${TARGET_SYSROOT})


include(CMakeInitializeConfigs)

function(cmake_initialize_per_config_variable _PREFIX _DOCSTRING)
  if (_PREFIX MATCHES "CMAKE_(C|CXX|ASM)_FLAGS")
    set(CMAKE_${CMAKE_MATCH_1}_FLAGS_INIT "${QT_COMPILER_FLAGS}")

    foreach (config DEBUG RELEASE MINSIZEREL RELWITHDEBINFO)
      if (DEFINED QT_COMPILER_FLAGS_${config})
        set(CMAKE_${CMAKE_MATCH_1}_FLAGS_${config}_INIT "${QT_COMPILER_FLAGS_${config}}")
      endif()
    endforeach()
  endif()


  if (_PREFIX MATCHES "CMAKE_(SHARED|MODULE|EXE)_LINKER_FLAGS")
    foreach (config SHARED MODULE EXE)
      set(CMAKE_${config}_LINKER_FLAGS_INIT "${QT_LINKER_FLAGS}")
    endforeach()
  endif()

  _cmake_initialize_per_config_variable(${ARGV})
endfunction()

set(XCB_PATH_VARIABLE ${TARGET_SYSROOT})

set(GL_INC_DIR ${TARGET_SYSROOT}/usr/include)
set(GL_LIB_DIR ${TARGET_SYSROOT}:${TARGET_SYSROOT}/usr/lib/aarch64-linux-gnu/:${TARGET_SYSROOT}/usr:
${TARGET_SYSROOT}/usr/lib)

set(EGL_INCLUDE_DIR ${GL_INC_DIR})
set(EGL_LIBRARY ${XCB_PATH_VARIABLE}/usr/lib/aarch64-linux-gnu/libEGL.so)

set(OPENGL_INCLUDE_DIR ${GL_INC_DIR})
set(OPENGL_opengl_LIBRARY ${XCB_PATH_VARIABLE}/usr/lib/aarch64-linux-gnu/libOpenGL.so)

set(GLESv2_INCLUDE_DIR ${GL_INC_DIR})
set(GLESv2_LIBRARY ${XCB_PATH_VARIABLE}/usr/lib/aarch64-linux-gnu/libGLESv2.so)

set(gbm_INCLUDE_DIR ${GL_INC_DIR})
set(gbm_LIBRARY ${XCB_PATH_VARIABLE}/usr/lib/aarch64-linux-gnu/libgbm.so)
```

```
set(Libdrm_INCLUDE_DIR ${GL_INC_DIR})
set(Libdrm_LIBRARY ${XCB_PATH_VARIABLE}/usr/lib/aarch64-linux-gnu/libdrm.so)

set(XCB_XCB_INCLUDE_DIR ${GL_INC_DIR})
set(XCB_XCB_LIBRARY ${XCB_PATH_VARIABLE}/usr/lib/aarch64-linux-gnu/libxcb.so)
```

Now that we have our toolchain, as well as the cross compiler and sysroot, we can continue with configuring and building Qt for our Raspberry Pi! As decided in the planning process, we will put all build configuration files on

```
qtpi-build
```

, so let's

```
cd
```

into that directory:

```
$ cd ~/qtpi-build
```

Now inside this directory, we can run this configuration command:

```
$ ../qt5/configure -release -opengl es2 -nomake examples -nomake tests -qt-host-path $HOME/qt-host -extprefix
$HOME/qt-raspi -prefix /usr/local/qt6 -device linux-rasp-pi4-aarch64 -device-option CROSS_COMPILE=aarch64-linux-
gnu- -- -DCMAKE_TOOLCHAIN_FILE=$HOME/toolchain.cmake -DQT_FEATURE_xcb=ON -DFEATURE_xcb_xlib=ON -DQT_FEATURE_xlib=ON
```

The command above will run a configuration script that came with the Qt source code and complementing it with the CMake toolchain that we just created. It is equivalent to this CMake command:

```
$ cmake ../qt5/ -GNinja -DCMAKE_BUILD_TYPE=Release -DINPUT_opengl=es2 -DQT_BUILD_EXAMPLES=OFF -DQT_BUILD_TESTS=OFF
-DQT_HOST_PATH=$HOME/qt-host -DCMAKE_STAGING_PREFIX=$HOME/qt-raspi -DCMAKE_INSTALL_PREFIX=/usr/local/qt6 -
DCMAKE_TOOLCHAIN_FILE=$HOME/toolchain.cmake -DQT_QMAKE_TARGET_MKSPEC=devices/linux-rasp-pi4-aarch64 -
DQT_FEATURE_xcb=ON -DFEATURE_xcb_xlib=ON -DQT_FEATURE_xlib=ON
```

Note: Both commands will do exactly the same thing: configuring Qt to be installed in folder

```
qt-raspi
```

on the host machine, with expectation that it will populate

```
usr/local/qt6
```

on the Raspberry Pi. If you already run the first command, then there is no need to run the second command.

The Qt installation from this build will be suitable to run Qt ARM64 applications using X11 as the

windowing system. You can check this page (https://doc.qt.io/qt-6/linux-requirements.html) from Qt Docs to learn more about xcb plugins.

In more advanced cases, check this page (https://doc.qt.io/qt-6/embedded-linux.html) from Qt Documentation if you want to use other windowing systems such as Wayland or EGLFS.

We can then proceed with building and installing.

```
$ cmake --build . --parallel 4
$ cmake --install .
```

After Qt is successfully installed on

```
~/qt-raspi
```

, we will then send this Qt installation back to the Raspberry Pi. To do this, we can use

```
rsync
```

as we did when building sysroot, or

```
scp
```

. Both will accomplish the same thing, and there should be no significant difference between both of them. To use

```
rsync
```

:

```
rsync -avz --rsync-path="sudo rsync" /path/to/qt-raspi/* <pi_username>@<pi_ip_address>:/usr/local/qt6
```

To use

```
scp
```

:

```
$ cd ~/qt-raspi
$ scp -r * <pi_username>@<pi_ip_address>:/usr/local/qt6
```

Final Configuration on Raspberry Pi

To ensure our Qt 6 installation can be run on Raspberry Pi, we need to setup a few environment variables. All of this can be done on the host device via SSH.

```
$ ssh pi@169.254.187.77

## enter your raspberry pi user password, if prompted

$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/qt6/lib/
```

This is the most important environment variable that we need to set, as it will enable Raspberry Pi to find our Qt 6 library path. If you intend to run the application through SSH from your host machine and see the application running on a monitor that is connected to the Raspberry Pi, you need to also set

```
DISPLAY
```

variable with

```
export DISPLAY=:0
```

in addition to the commands above.

Compiling and Running Qt Project

Now we can compile a Qt application and get a binary that is suitable to deploy on Raspberry Pi. There are two methods to achieve this: with Terminal or with Qt Creator. For both methods, we will use the analogclock example project.

With Terminal

```
$ cd ~/qt5/qtbase/examples/gui/analogclock/
$ ls ## make sure there is a CMakeLists.txt file
$ ~/qt-raspi/bin/qt-cmake CMakeLists.txt

### CMake is configuring here. After it is done without any error, we can proceed to build and install it

$ cmake --build . --parallel 4
$ cmake --install .
```

We now have the binary of our application ready to be run on the Raspberry Pi. We can then send the binary to our Raspberry Pi using either scp or rsync and run the application (either through SSH or directly on the Raspberry Pi with keyboard and mouse connected).

```
$ scp -r gui_analogclock <pi_username>@<pi_ip_address>:/home/pi
$ ssh <pi_username>@<pi_ip_address>
$ cd ~ ## or cd to the directory where you send the binary
$ ./gui_analogclock
```

With Qt Creator

Open Qt Creator. On the main toolbar, go to **Tools > Options**.

- Inside **Kits** menu, select **Qt Versions** then click **Add.** Locate the *qmake* file inside

```
~/qt-raspi/bin
```

  . Then click **Apply.**
- We also have to make sure that our cross-compiler is already detected by the Qt Creator. Select **Compilers** tab, then check if our compiler (*aarch64-linux-gnu-gcc* and *aarch64-linux-gnu-g++*) is already listed. If not, you must add a GCC compiler and locate the location of the compiler. Then click **Apply.**

Another component that we need to set in Qt Creator is our own device. Still in **Options** menu, select **Devices** on the sidebar.

- Under **Devices** tab, select **Add**. Inside the **Device Configuration Wizard Selection**, select **Generic Linux Device**, then click **Start Wizard.**
- Specify an identifiable name, your Raspberry Pi IP address that you use to SSH, and the user name of the Raspberry Pi.
- You will be prompted to deploy public key to the target device. This will speed up the deployment process as you don't need to input a password every time Qt Creator is attempting to send file to the target.
- Click **Next**, and then **Finish.** Qt Creator will test the connection upon finishing and will tell you whether a connection to the target device can be established.

After setting all the necessary components, we can now make a new kit. Go back to **Kits** menu on the sidebar.

- Click **Add.** Name your kit with a distinguishable name.
- On **Device type** option, select **Generic Linux Device.**
- On **Device** option, select the name of the device that you already set in previous step.
- On **Compiler** option, select the cross-compiler (in this guide, *aarch64-linux-gnu-gcc* and *aarch64-linux-gnu-g++*) for both C and C++ compilers.
- On **Qt version** option, select the name of the cross-compiled Qt version that you set in the previous step.
- On **CMake Configuration** option, click Change and add

```
-DCMAKE_TOOLCHAIN_FILE:UNINITIALIZED=/path/to/qt-raspi/lib/cmake/Qt6/qt.toolchain.cmake
```

  to the end of the line. Apply and then OK.

Click **Apply** and then **OK.** Now go to **File > Open File or Project**, then open *CMakeLists.txt* of the project (for this example, it is located in

```
~/qt5/qtbase/examples/gui/analogclock/CMakeLists.txt
```

). Qt Creator will try to configure the project using CMake.

Before running and deploying the application, we need to customise a few deployment/run settings. Go to Projects sidebar menu, then under the kit name click Run.

- Under **Run** section, on **X11 Forwarding** tick "Forward to local display" checkbox and input

  ```
  :0
  ```

  to the text field. This will forward the application window to be opened on host device rather than on Raspberry Pi monitor (the application instance is still running on Raspberry Pi).
- Sometimes Qt Creator does not fetch the device environment properly. This would cause the library of our Qt installation undetected by the Qt application. Under **Environment** section, click **Details** to expand the environment option. Click **Add**, then on **Variable** column type

  ```
  LD_LIBRARY_PATH
  ```

  . On the **Value** column, type the location of Qt installation inside the Raspberry Pi preceded by a colon (:), in our case it is

  ```
  :/usr/local/qt6/lib/
  ```

  .

After configuring these settings. run the application (Ctrl+R). The application window should be displayed on host's monitor display.

Known Issues

Issue with libdbus

During testing, we found a compilation error related to

```
libdbus
```

.

```
...
/usr/lib/gcc-cross/aarch64-linux-gnu/9/../../../../aarch64-linux-gnu/bin/ld: /home/uraihan/rpi-sysroot/usr/lib/
aarch64-linux-gnu/libdbus-1.a(libdbus_1_la-dbus-message.o): relocation R_AARCH64_ADR_PREL_PG_HI21 against symbol
`dbus_message_unref' which may bind externally can not be used when making a shared object; recompile with -fPIC
...
```

This could happen when the libdbus inside the initial sysroot is not a dynamically shared objects. This is normally fixable by doing a clean reinstall of Raspberry Pi OS (reformat microSD card, then install Raspberry Pi OS again through the Imager). Another recommendation is to try to reinstall

```
libdbus-1-dev
```

package. But if the problem persists and you don't need dbus, you can disable the dbus feature by adding

```
-DFEATURE_dbus=OFF
```

flag in the CMake configuration command.

Another walkaround worth trying is to change

from:

```
set(QT_LINKER_FLAGS "-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed")
```

to

```
set(QT_LINKER_FLAGS "-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed -ldbus-1")
```

Additionally make sure that link to rpi-sysroot/usr/lib/aarch64-linux-gnu/libdbus-1.so is valid. It should be linking to libdbus-1.so.3.19.13 in the same directory.

Note: You'll need DBus to have Bluetooth module working.

Issue with -fuse-ld=lld not found

Add "**-Wl,-fuse-ld=gold" to QT_LINKER_FLAGS.**

```
set(QT_LINKER_FLAGS "-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed -Wl,-fuse-ld=gold")
```

Issue with QtWayland not compiling:

Add -skip qtwayland to ./configure script like so:

```
$ ../qt5/configure -release -opengl es2 -nomake examples -nomake tests -skip qtwayland -qt-host-path $HOME/qt-host
-extprefix $HOME/qt-raspi -prefix /usr/local/qt6 -device linux-rasp-pi3-g++ -device-option CROSS_COMPILE=aarch64-
linux-gnu- -- -DCMAKE_TOOLCHAIN_FILE=$HOME/toolchain.cmake -DQT_FEATURE_xcb=ON -DFEATURE_xcb_xlib=ON -
DQT_FEATURE_xlib=ON
```

Environment variables are not permanent:

Environment variables are not persistent during ssh session if you use export command.

You can add this in ~/.bashrc file

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/qt6/lib/
```

Compiling app in QtCreator for device linux-rasp-pi3-g++ throws various errors:

Open qt-raspi/mkspecs/devices/linux-rasp-pi3-g++/qmake.conf

Remove -mfpu=crypto-neon-fp-armv8

Change include(../common/linux_arm_device_post.conf) to include(../common/linux_device_post.conf)

Various libraries not found when launching app on Raspberry Pi:

Make sure that directory **/usr/local/qt6/lib/** exists. You may need to change your LD_LIBRARY_PATH to **/usr/local/qt6/qt-raspi/lib/**

Further Readings

- Qt Docs:
  - Qt for Embedded Device (https://doc-snapshots.qt.io/qt6-6.3/embedded-linux.html) (alongside a guideline on Configuring an Embedded Linux Device (https://doc-snapshots.qt.io/qt6-6.3/configure-linux-device.html)).
  - XCB plugins for running Qt on X11 (https://doc.qt.io/qt-6/linux-requirements.html)
- CMake Docs: cmake-toolchains (https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html).

Retrieved from "https://wiki.qt.io/index.php?title=Cross-Compile_Qt_6_for_Raspberry_Pi&oldid=41748"

This page was last edited on 1 May 2024, at 04:06.