Building the C++ example project

Pieter P

Downloading the example project

The example is just a command line program that asks the user for his/her name using Boost.program_options and prints a *hello world* message.
Download it from GitHub:

```
$ mkdir -p ~/GitHub
$ cd ~/GitHub
$ git clone https://github.com/tttapa/RPi-Cross-Cpp-Development.git
$ cd RPi-Cross-Cpp-Development
```

Customize the paths

Edit the CMake toolchain file to point to the correct root filesystem path (the value of the `CMAKE_SYSROOT` variable):

```
$ sed -i 's/schroot-name/rpizero-buster/' cmake/armv6-rpi-linux-gnueabihf.cmake
```

You might want to change the toolchain prefix or the architecture-specific flags as well.
If you're using a 64-bit toolchain, edit the `aarch64-rpi3-linux-gnu.cmake` file instead.

Installing the dependencies

Thanks to the `sbuild` development tools, managing dependencies is really easy, you can just install them to the Raspberry Pi OS root filesystem using the familiar `apt-get install` command. We use the `sbuild-apt` tool, and specify the name of the root filesystem we created on the previous page.

```
$ sudo sbuild-apt rpizero-buster-armhf apt-get install libboost-all-dev
```

Now Boost is installed on our build computer, but not yet on the Raspberry Pi itself. Let's do that now, using the standard `apt install` command over SSH:

```
$ ssh RPi0 sudo apt install -y libboost-all-dev
```

Strictly speaking, we don't need all development libraries on the Pi, so to save some time and space, you could install just the libraries you need, e.g.

```
$ ssh RPi0 sudo apt install -y libboost-program-options1.67.0
```
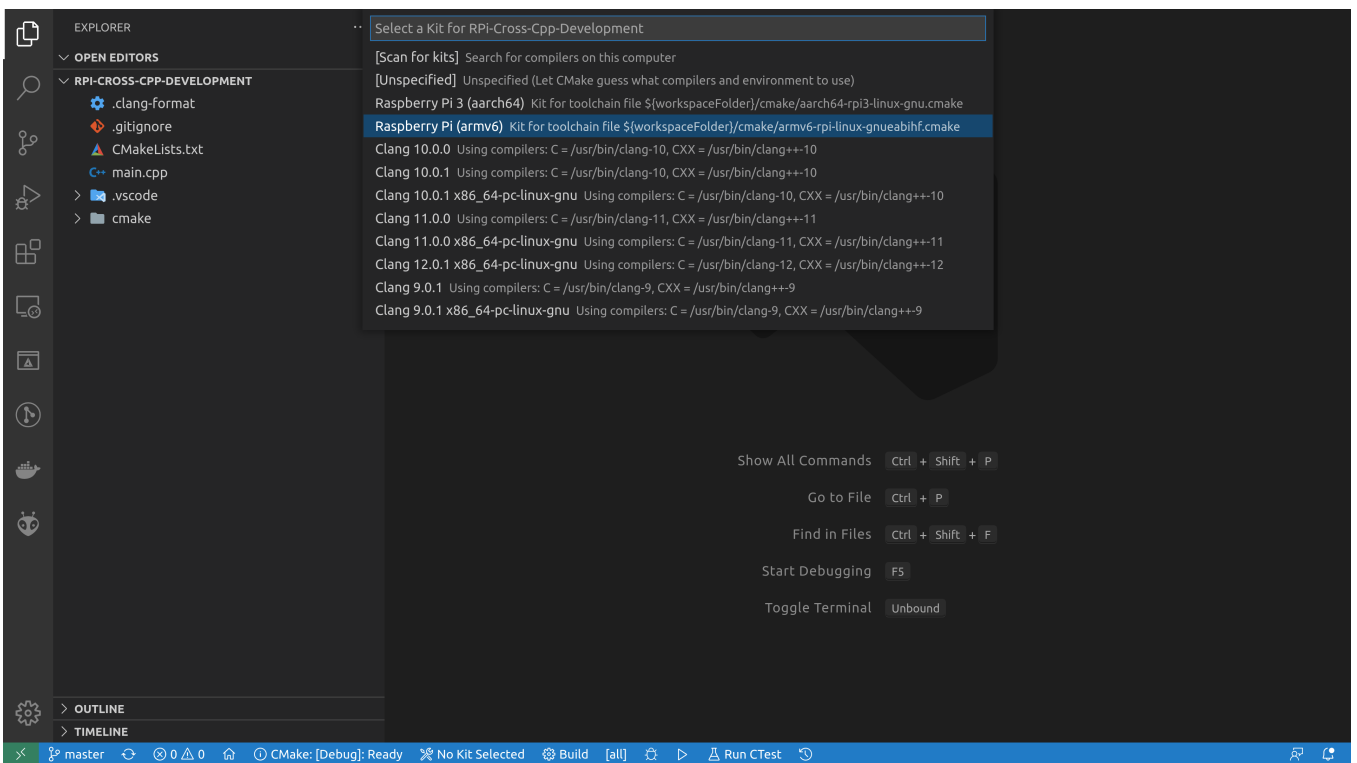
Configuring and building the project

Open the `~/GitHub/RPi-Cross-Cpp-Development` folder in Visual Studio Code (using the `Ctrl+K+O` shortcut or "Open Folder").

The CMake Tools extension will now ask you to configure the project. Click "Yes".
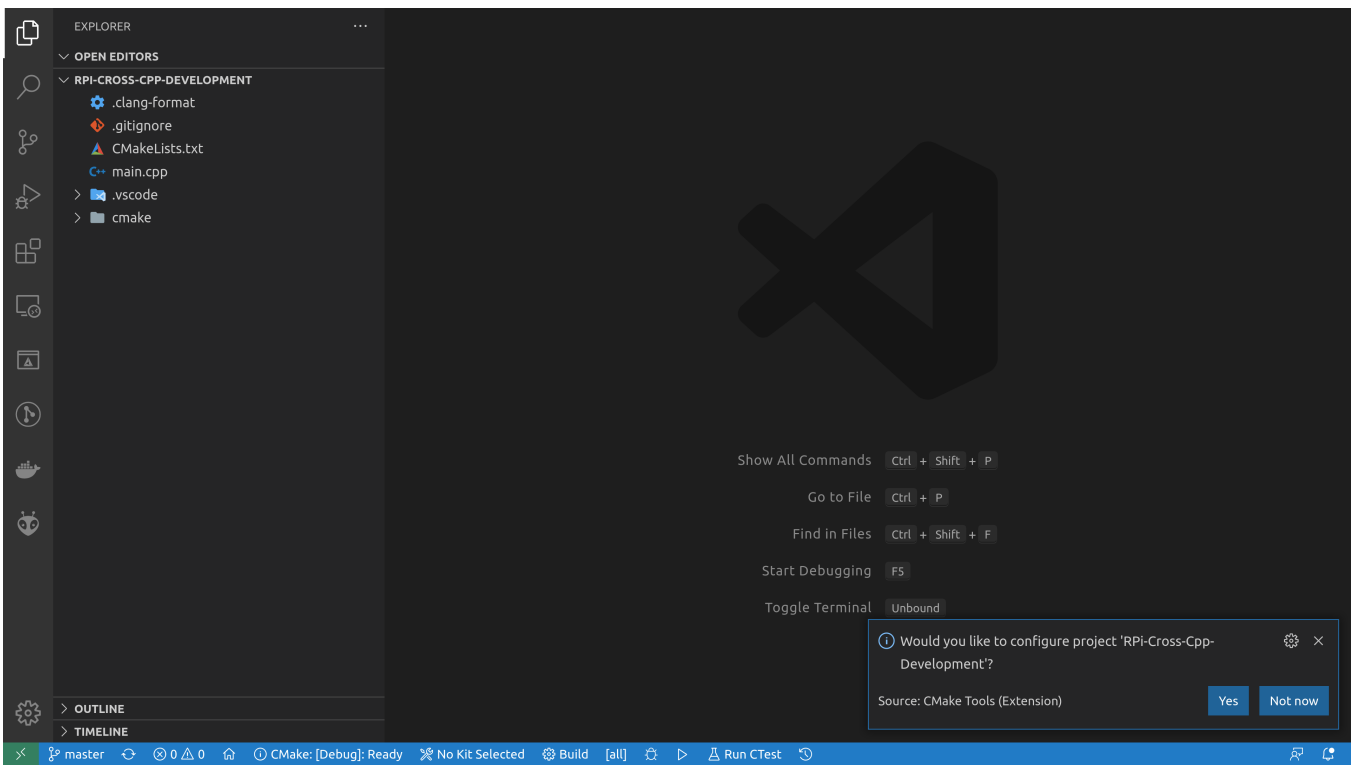
Then select the correct toolchain, in this case, we want the "Raspberry Pi (armv6)" toolchain.


CMake will now configure the project. If you followed the instructions on the previous pages correctly, it finds the `armv6-rpi-linux-gnueabihf` toolchain we installed, as well as the Boost libraries in the Raspberry Pi OS root filesystem.

If CMake raises an error, see the Troubleshooting section below.
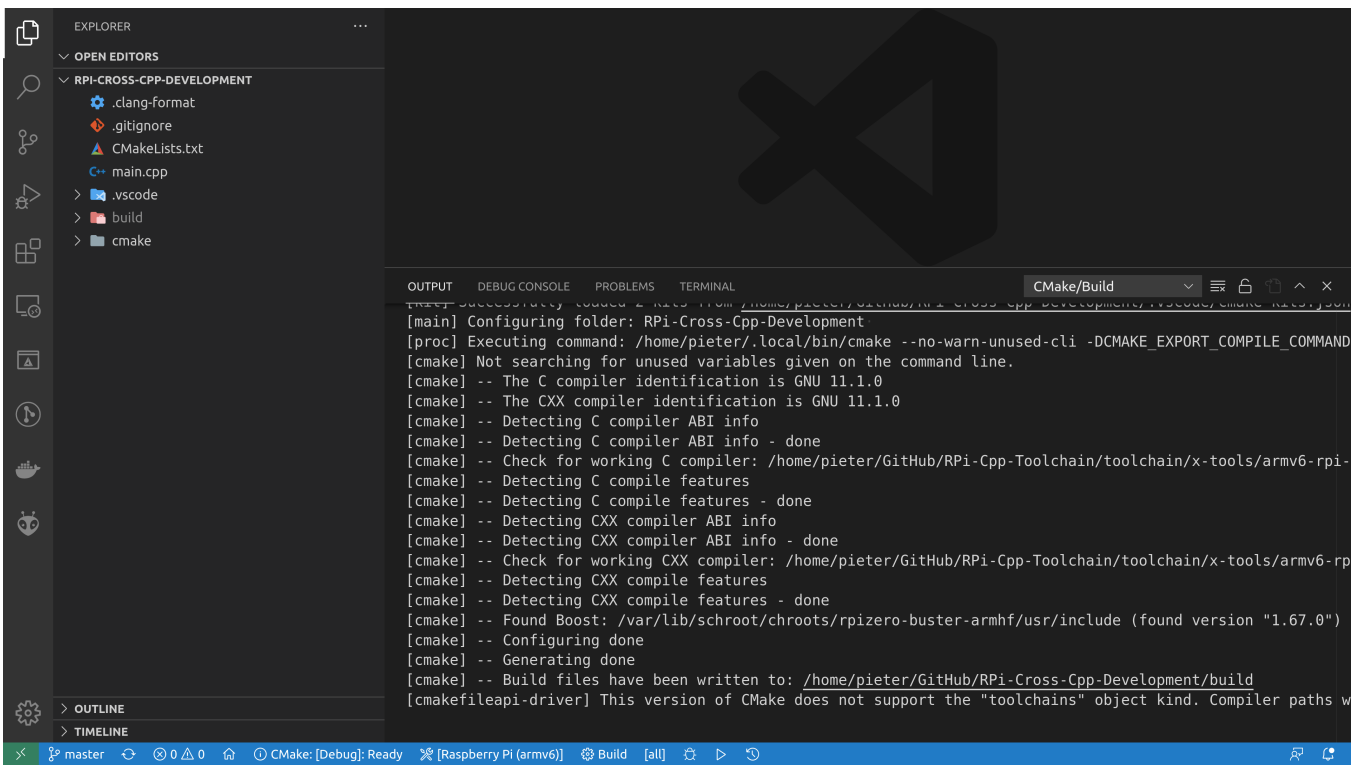
Finally, click the ⚙ Build icon to actually compile the *hello world* program.
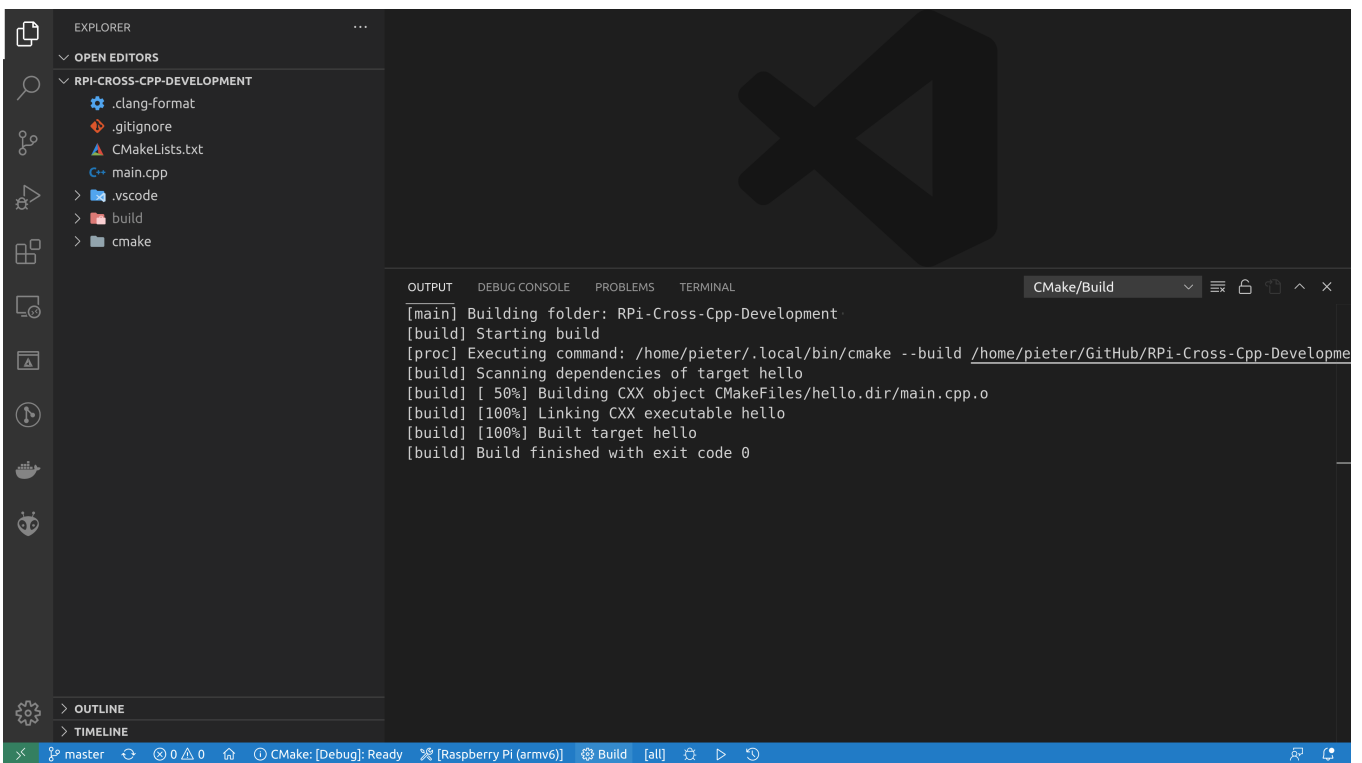
You can verify that everything worked correctly using the `file` command:

```
$ file build/hello
build/hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-
armhf.so.3, for GNU/Linux 3.2.0, with debug_info, not stripped
```

As you can see, `hello` is a 32-bit ARM executable, so the cross-compilation was successful.

Running the example program on the Raspberry Pi

All we have to do now is copy the `hello` file to the Raspberry Pi and run it. We'll copy it over SSH using the `scp` command, and then run it over SSH as well:

```
$ scp build/hello RPi0:~
$ ssh RPi0 '~/hello' --help
```


That's it, you have successfully executed your cross-compiled C++ program on the Raspberry Pi!

Using a staging directory

For larger projects, you usually don't want to have to dig around in the `build` directory to gather all the files you need to copy to the Raspberry Pi. Instead, you'll use `cmake --install` to let CMake install all the necessary files into a so-called staging directory on your computer, which you can then copy to the Pi.

After building the project as explained in the previous section, press `Ctrl+Shift+P` in VSCode and execute the `CMake: Install` command. You can then find the `hello` program in the `bin` subdirectory of `~/RPi-dev/staging-armv6-rpi`.

You'll usually compress the staging folder using `tar` and then deploy it to the Pi by extracting it to the `~/.local` or `/usr/local` folder.

A closer look at the build process

There's only one source file, `main.cpp`, and we'll focus on the build process, so we won't go into the code in detail here. The main `CMakeLists.txt` file is really basic: it just defines the project, looks for the Boost.program_options library, compiles `main.cpp` into an executable with the name `hello`, and then links this executable with the Boost library. Finally, an install rule is added to install the `hello` program to the `bin` folder of the staging directory.

CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.16)
2  project(hello VERSION 0.1.0 LANGUAGES C CXX Fortran)
3
4  find_package(Boost REQUIRED COMPONENTS program_options)
5
6  add_executable(hello main.cpp)
7  target_link_libraries(hello PRIVATE Boost::program_options)
8
9  include(GNUInstallDirs)
10 install(TARGETS hello
11         RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR})
```

The way we tell CMake to cross-compile this project for the Raspberry Pi is using a so-called toolchain file. You can find more information on https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html#cross-compiling.

cmake/armv6-rpi-linux-gnueabihf.cmake

```cmake
1   # https://cmake.org/cmake/help/book/mastering-cmake/chapter/Cross%20Compiling%20With%20CMake.html
2
3   # Cross-compilation system information
4   set(CMAKE_SYSTEM_NAME Linux)
5   set(CMAKE_SYSTEM_PROCESSOR arm)
6
7   # The sysroot contains all the libraries we might need to link against and
8   # possibly headers we need for compilation
9   set(CMAKE_SYSROOT /var/lib/schroot/chroots/rpizero-buster-armhf)
10  set(CMAKE_FIND_ROOT_PATH ${CMAKE_SYSROOT})
11  set(CMAKE_LIBRARY_ARCHITECTURE arm-linux-gnueabihf)
12  set(CMAKE_STAGING_PREFIX $ENV{HOME}/RPi-dev/staging-armv6-rpi)
13
14  # Set the compilers for C, C++ and Fortran
15  set(RPI_GCC_TRIPLE "armv6-rpi-linux-gnueabihf")
16  set(CMAKE_C_COMPILER ${RPI_GCC_TRIPLE}-gcc CACHE FILEPATH "C compiler")
17  set(CMAKE_CXX_COMPILER ${RPI_GCC_TRIPLE}-g++ CACHE FILEPATH "C++ compiler")
18  set(CMAKE_Fortran_COMPILER ${RPI_GCC_TRIPLE}-gfortran CACHE FILEPATH "Fortran compiler")
19
20  # Set the architecture-specific compiler flags
21  set(ARCH_FLAGS "-mcpu=arm1176jzf-s")
22  set(CMAKE_C_FLAGS_INIT ${ARCH_FLAGS})
23  set(CMAKE_CXX_FLAGS_INIT ${ARCH_FLAGS})
24  set(CMAKE_Fortran_FLAGS_INIT ${ARCH_FLAGS})
25
26  # Don't look for programs in the sysroot (these are ARM programs, they won't run
27  # on the build machine)
28  set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
29  # Only look for libraries, headers and packages in the sysroot, don't look on
30  # the build machine
31  set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
32  set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
33  set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
34
35  set(CPACK_DEBIAN_PACKAGE_ARCHITECTURE armhf)
```

First we set the system name and architecture, then we set some paths to the sysroot, so that CMake will be able to find the Boost library. If you gave your chroot a different name, you have to change these paths. The `CMAKE_STAGING_PREFIX` variable is useful for installing your project to a "staging" directory for deploying to the Pi. The `CMAKE_LIBRARY_ARCHITECTURE` variable helps CMake find libraries in Debian's Multiarch directory structure, e.g. in `/usr/lib/arm-linux-gnueabihf/`.

Next, we tell CMake to use the `armv6-rpi-linux-gnueabihf` cross-compilers, and we set some compiler flags. The `-mcpu` flag is a bit redundant here, since the toolchain is already configured to generate code for that specific CPU, but it serves as an example, you might want to add more specific flags.

Finally, we tell CMake never to run any programs it finds in the sysroot (because they are ARM binaries, you cannot run them on your computer without extra steps), and it should only look for libraries, headers and packages in the sysroot. We don't want CMake to find and use packages installed on our computer, because these are x86_64 libraries, not ARM.

The CMake Tools VSCode extension picks up these toolchain files using the following configuration file:

.vscode/cmake-kits.json

```json
1   [
2       {
3           "name": "Raspberry Pi 3 (aarch64)",
4           "toolchainFile": "${workspaceFolder}/cmake/aarch64-rpi3-linux-gnu.cmake"
5       },
6       {
7           "name": "Raspberry Pi 3 Clang (aarch64)",
8           "toolchainFile": "${workspaceFolder}/cmake/aarch64-rpi3-linux-gnu-clang.cmake"
9       },
10      {
11          "name": "Raspberry Pi (armv6)",
12          "toolchainFile": "${workspaceFolder}/cmake/armv6-rpi-linux-gnueabihf.cmake",
13          "environmentVariables": {
14              "PATH": "${env:HOME}/opt/x-tools/armv6-rpi-linux-gnueabihf/bin:${env:PATH}"
15          }
16      }
17  ]
```

You can see that there's a second toolchain file for newer 64-bit boards. If you need different configurations for different Pi models, you can add them here and easily switch between them by clicking the CMake Kit button in VSCode.

Manual build

If you don't want to use VSCode as your editor, you can also build the project from the command line:

```
$ cd ~/GitHub/RPi-Cross-Cpp-Development
$ rm -rf build
$ cmake -S . -B build -DCMAKE_TOOLCHAIN_FILE=cmake/armv6-rpi-linux-gnueabihf.cmake
$ cmake --build build -j
$ cmake --install build
```

Troubleshooting

This section covers some common problems that you might run into. After fixing them, you can re-run CMake by pressing Ctrl+Shift+P in VSCode and executing the CMake: Delete Cache and Reconfigure command.

Compiler not found

```
1    The CMAKE_C_COMPILER:
2
3      armv6-rpi-linux-gnueabihf-gcc
4
5    is not a full path and was not found in the PATH.
```

This indicates that you either didn't download and install the toolchain correctly, or that it has not been added to the PATH. See <u>the previous page</u> for more details.

Note that changing the PATH in `~/.profile` only has effect after logging out and back in again.
If you used `export PATH="..."` in a terminal, this only affects the current terminal, so unless you start VSCode from that terminal, it won't see the updated PATH.

Compiler is not able to compile a simple test program

```
1   -- Check for working C compiler: ~/opt/x-tools/armv6-rpi-linux-gnueabihf/bin/armv6-rpi-linux-gnueabihf-gcc - broken
2   CMake Error at ~/.local/share/cmake-3.22/Modules/CMakeTestCCompiler.cmake:69 (message):
3     The C compiler
4
5       "~/opt/x-tools/armv6-rpi-linux-gnueabihf/bin/armv6-rpi-linux-gnueabihf-gcc"
6
7     is not able to compile a simple test program.
8
9     It fails with the following output:
10
11       Change Dir: ~/GitHub/RPi-Cross-Cpp-Development/build/CMakeFiles/CMakeTmp
12
13       Run Build Command(s):/usr/bin/ninja cmTC_f7b02 && [1/2] Building C object CMakeFiles/cmTC_f7b02.dir/testCCompiler.c.o
14       [2/2] Linking C executable cmTC_f7b02
15       FAILED: cmTC_f7b02
16       : && ~/opt/x-tools/armv6-rpi-linux-gnueabihf/bin/armv6-rpi-linux-gnueabihf-gcc --sysroot=/var/lib/schroot/chroots/wrong-
    name-armhf -mcpu=arm1176jzf-s -mcpu=arm1176jzf-s  CMakeFiles/cmTC_f7b02.dir/testCCompiler.c.o -o cmTC_f7b02   && :
17       ~/opt/x-tools/armv6-rpi-linux-gnueabihf/bin/../lib/gcc/armv6-rpi-linux-gnueabihf/11.2.0/../../../../armv6-rpi-linux-
    gnueabihf/bin/ld.bfd: cannot find crt1.o: No such file or directory
18       ~/opt/x-tools/armv6-rpi-linux-gnueabihf/bin/../lib/gcc/armv6-rpi-linux-gnueabihf/11.2.0/../../../../armv6-rpi-linux-
    gnueabihf/bin/ld.bfd: cannot find crti.o: No such file or directory
19       collect2: error: ld returned 1 exit status
```

This error indicates that the path to the root filesystem is not correct. Make sure that the `CMAKE_SYSROOT` variable is set to the correct schroot you created on the <u>previous page</u>, double check the name (with the architecture as suffix), and check that the folder exists.

Boost not found

```
1   CMake Error at ~/.local/share/cmake-3.22/Modules/FindPackageHandleStandardArgs.cmake:230 (message):
2     Could NOT find Boost (missing: Boost_INCLUDE_DIR program_options)
3   Call Stack (most recent call first):
4     ~/.local/share/cmake-3.22/Modules/FindPackageHandleStandardArgs.cmake:594 (_FPHSA_FAILURE_MESSAGE)
5     ~/.local/share/cmake-3.22/Modules/FindBoost.cmake:2375 (find_package_handle_standard_args)
6     CMakeLists.txt:4 (find_package)
```

This error occurs when the Boost development package is not installed in your root filesystem. Follow the <u>Installing the dependencies</u> section to install it.