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:

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

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
```

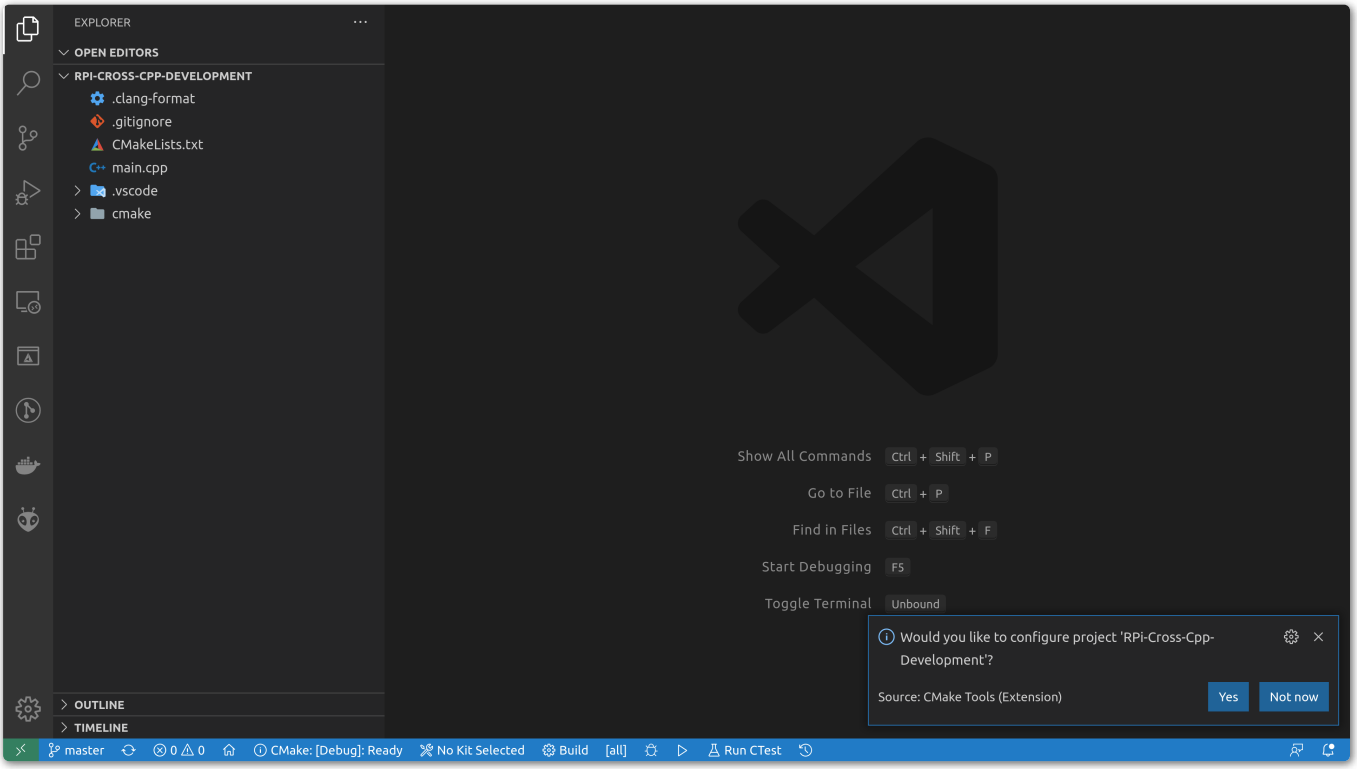
Strictly speaking, we don't need all development libraries on the Pi, so to save some time and space, you could install just the libraries you need, e.g.

```
$ ssh RPi0 sudo apt install -y libboost-program-options1.67.0
```

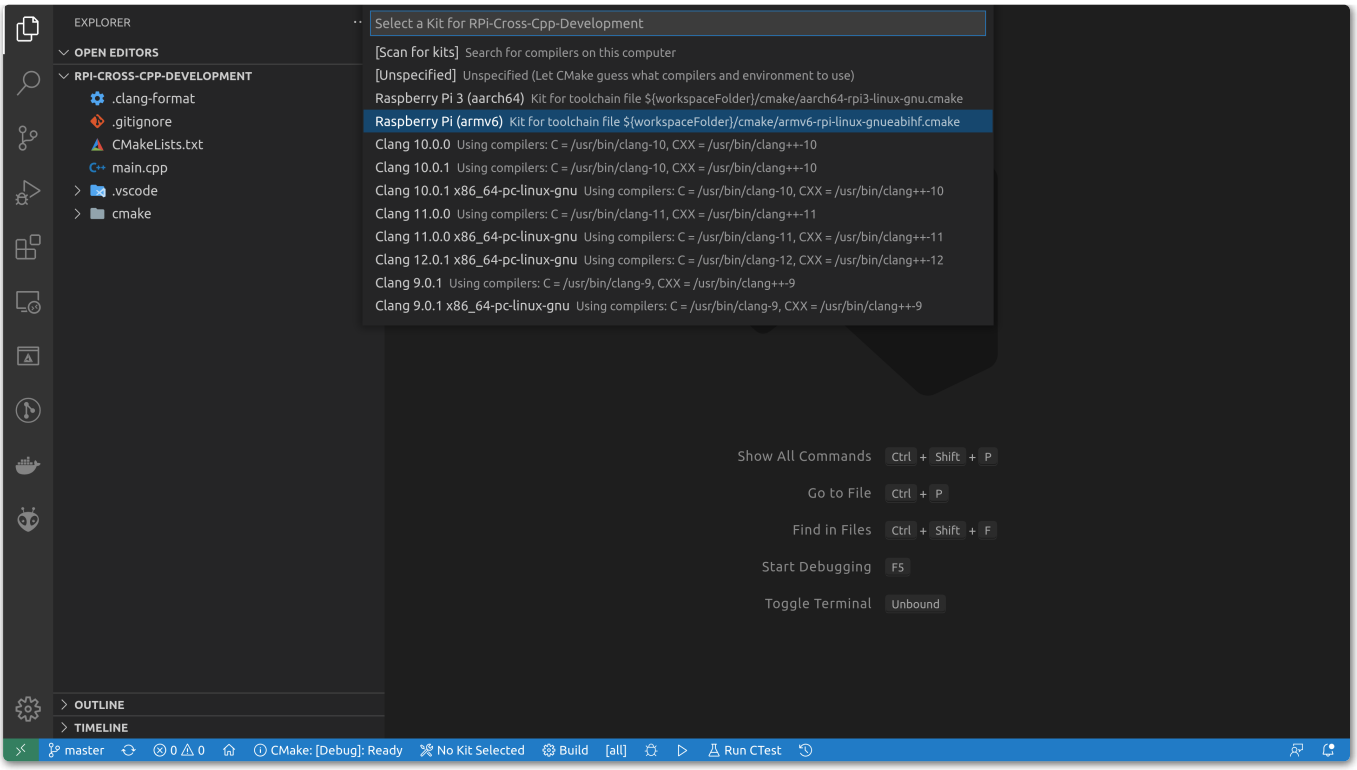
Configuring and building the project

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

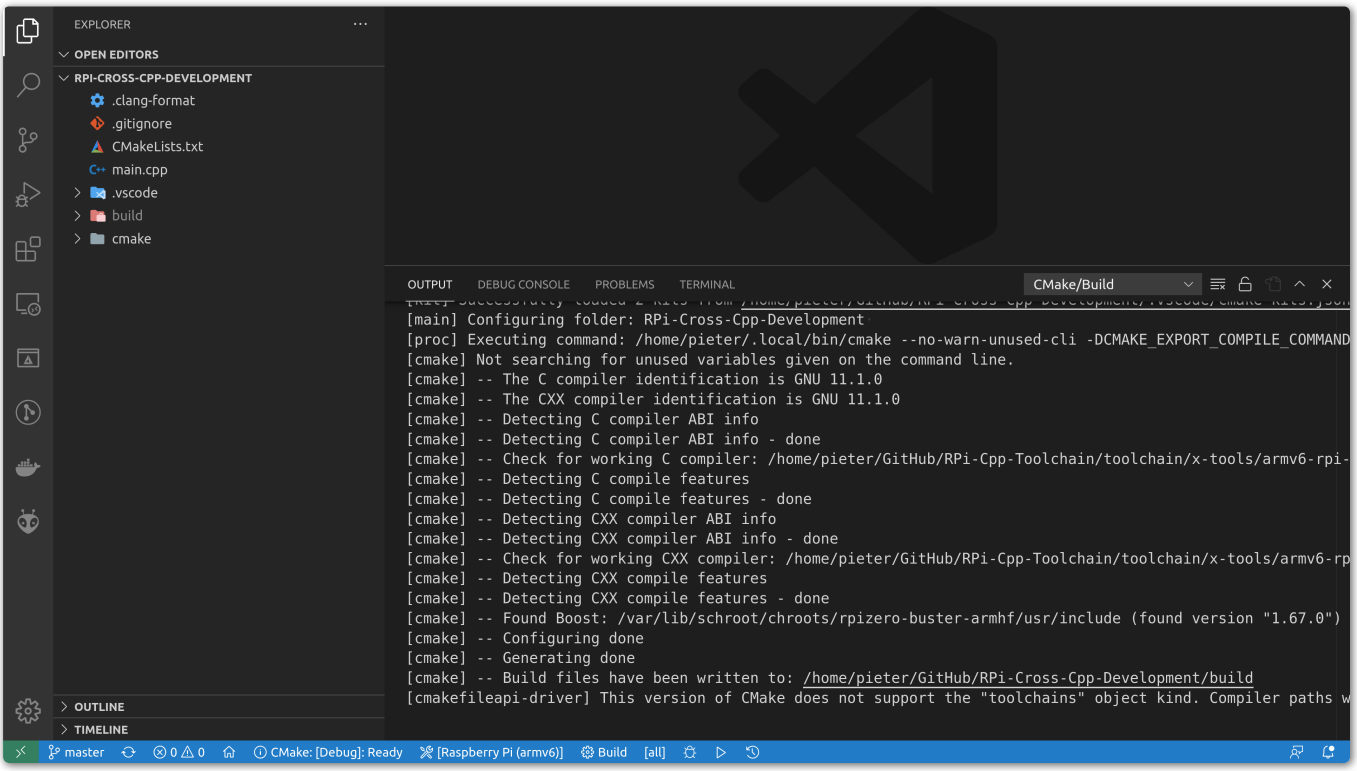
The CMake Tools extension will now ask you to configure the project. Click "Yes".



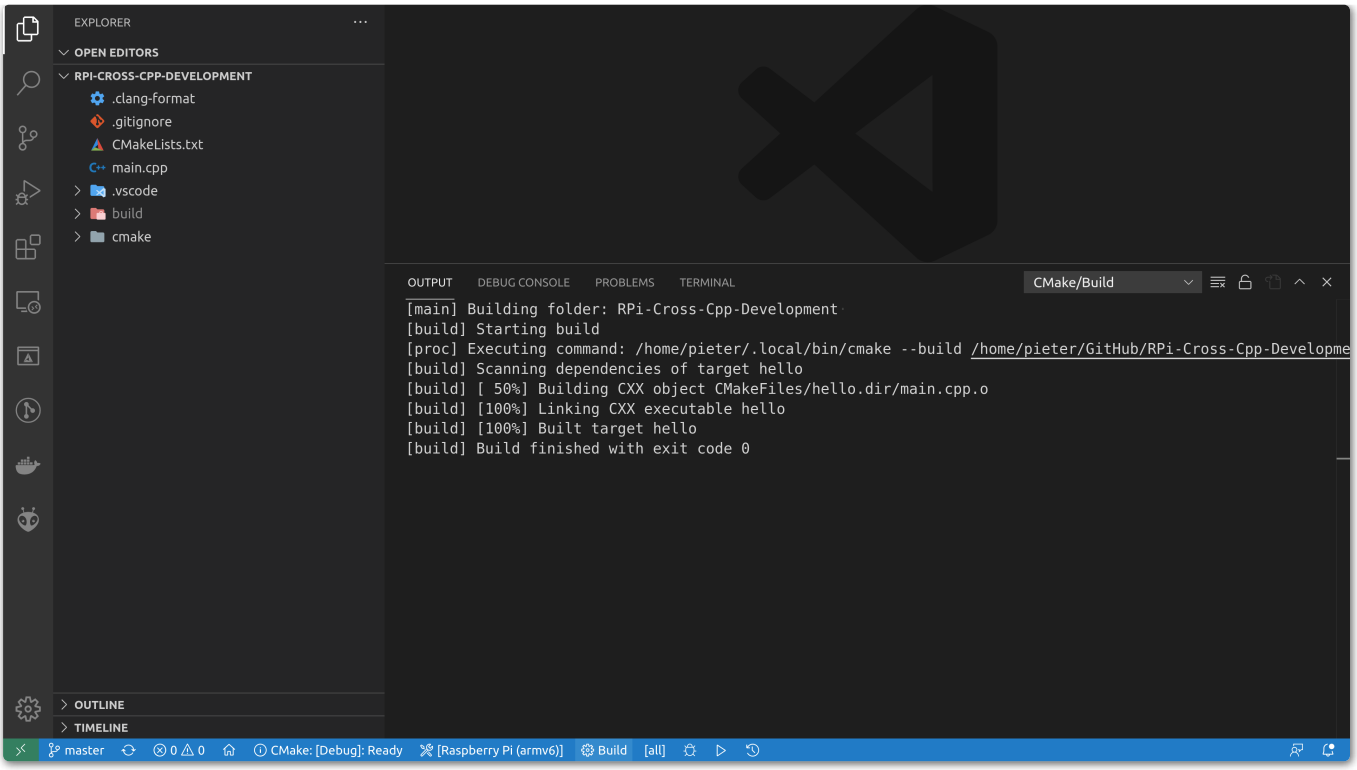
Then select the correct toolchain, in this case, we want the "Raspberry Pi (armv6)" toolchain.



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.



Finally, click the Build icon to actually compile the *hello world* program.



You can verify that everything worked correctly using the `file` command:

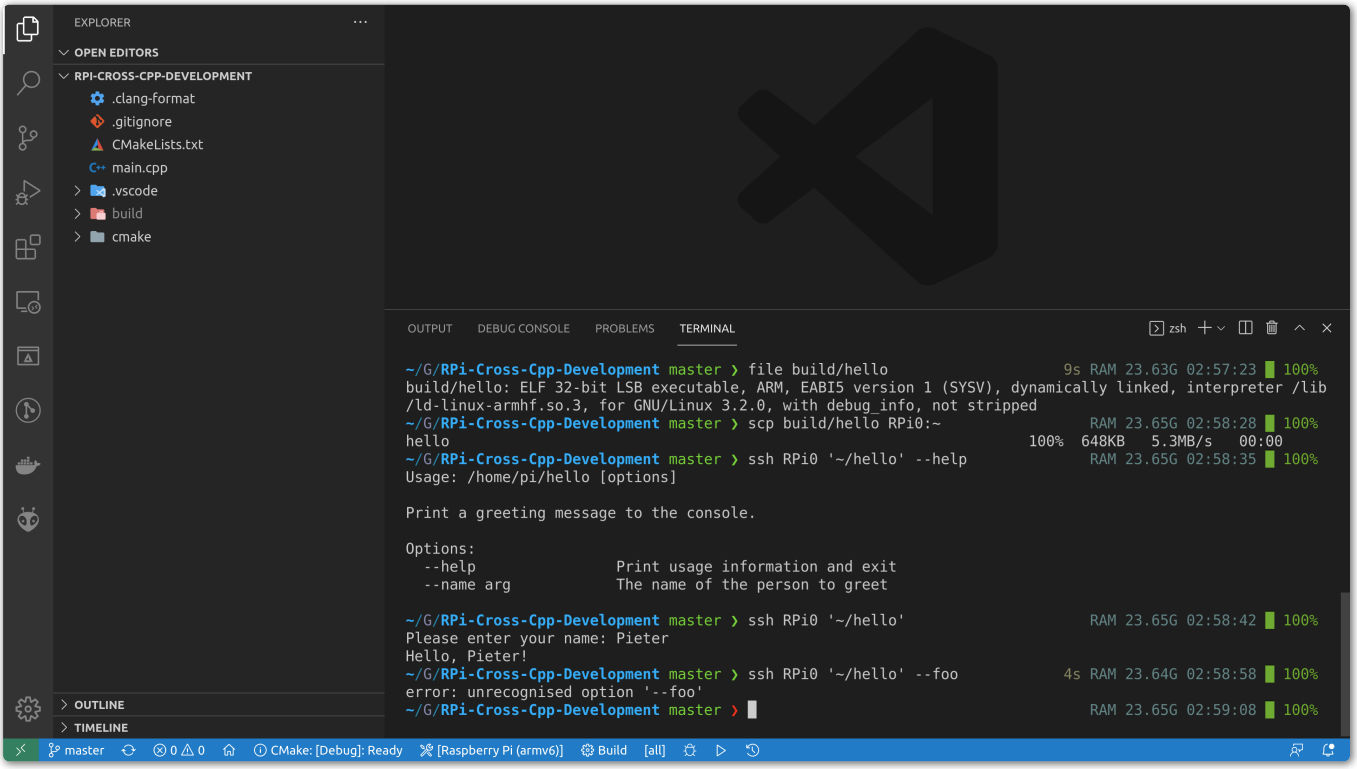
```
$ file build/hello
  build/hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-
  armhf.so.3, for GNU/Linux 3.2.0, with debug_info, not stripped
```

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!

A closer look at the build process

There's only one source file, `main.cpp`, we won't go into 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.

`CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.16)
project(hello VERSION 0.1.0)

find_package(Boost REQUIRED COMPONENTS program_options)

add_executable(hello main.cpp)
target_link_libraries(hello PRIVATE Boost::program_options)
```

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`

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_SYSROOT /var/lib/schroot/chroots/rpizero-buster-armhf)
SET(CMAKE_FIND_ROOT_PATH ${CMAKE_SYSROOT})
set(CMAKE_STAGING_PREFIX $ENV{HOME}/RPi-dev/staging-armv6-rpi)
set(CMAKE_LIBRARY_ARCHITECTURE arm-linux-gnueabihf)

set(cross "armv6-rpi-linux-gnueabihf")
set(CMAKE_C_COMPILER ${cross}-gcc)
set(CMAKE_CXX_COMPILER ${cross}-g++)

set(ARCH_FLAGS "-mcpu=arm1176jzf-s")
SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${ARCH_FLAGS}")
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${ARCH_FLAGS}")

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(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 might want to change these paths. The `CMAKE_STAGING_PREFIX` variable is useful for installing your project to a "staging" directory for deploying to the Pi. It is not used in this simple project. 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`

```
[
{
    "name": "Raspberry Pi 3 (aarch64)",
    "toolchainFile": "${workspaceFolder}/cmake/aarch64-rpi3-linux-gnu.cmake"
},
{
    "name": "Raspberry Pi (armv6)",
    "toolchainFile": "${workspaceFolder}/cmake/armv6-rpi-linux-gnueabihf.cmake"
}
]
```

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.

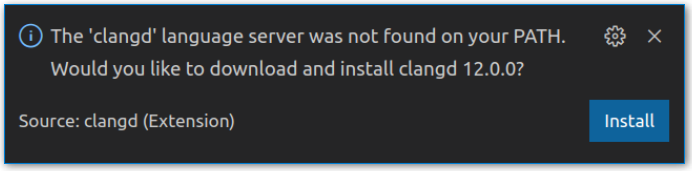
Manual build

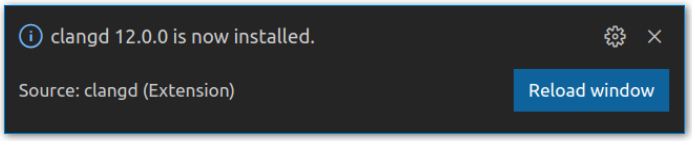
If you don't want to use VSCode as your editor, you can also build the project from the command line:

```
$ 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
```

Installing clangd

If this is the first time you're using the clangd extension, you'll have to install the language server. When you open a C++ file for the first time, the extension will automatically give you a prompt:





Once the language server is installed, you get all features you'd expect from an IDE, such as semantic syntax highlighting, go-to-definition, autocomplete, documentation, refactoring options, etc.