Cross-Compiling the C++ Example Project

Pieter P

This page gives an overview of the included example project and has instructions on how to cross-compile it.

Overview of the example project

The Greeter library

For this example, we'll create a very simple library with a single function that just takes a name and an output stream as arguments, and that prints a greeting message to this stream. It's basically a "Hello, World!" example, but as a library for demonstration purposes.

The structure of the library will be as follows:

```
greeter
├── CMakeLists.txt
├── include
│   └── greeter
│       └── greeter.hpp
├── src
│   └── greeter.cpp
└── test
    ├── CMakeLists.txt
    └── greeter.test.cpp
```

This structure is very common for C++ libraries: the function prototypes/declarations will be in the header file `greeter.hpp`. The implementations for these functions are in the corresponding implementation file `greeter.cpp`.
The `CMakeLists.txt` file in the `greeter` directory specifies how the library should be compiled, and where to find the headers.
Additionally, there's a `test` folder with unit tests in `greeter.test.cpp`. The `CMakeLists.txt` file in this folder specifies how to compile and link the tests executable.

`greeter.hpp`

```cpp
 1  #pragma once
 2
 3  #include <iosfwd> // std::ostream
 4  #include <string> // std::string
 5
 6  namespace greeter {
 7
 8  /**
 9   * @brief   Function that greets a given person.
10   *
11   * @param   name
12   *          The name of the person to greet.
13   * @param   os
14   *          The output stream to print the greetings to.
15   */
16  void sayHello(const std::string &name, std::ostream &os);
17
18  } // namespace greeter
```

`greeter.cpp`

```cpp
 1  #include <greeter/greeter.hpp>
 2  #include <iostream> // std::endl, <<
 3
 4  namespace greeter {
 5
 6  void sayHello(const std::string &name, std::ostream &os) {
 7      os << "Hello, " << name << "!" << std::endl;
 8  }
 9
10  } // namespace greeter
```

`CMakeLists.txt`

```cmake
1   # Add a new library with the name "greeter" that is compiled from the source
2   # file "src/greeter.cpp".
3   add_library(greeter
4       "src/greeter.cpp"
5       "include/greeter/greeter.hpp"
6   )
7
8   # The public header files for greeter can be found in the "include" folder, and
9   # they have to be passed to the compiler, both for compiling the library itself
10  # and for using the library in a other implementation files (such as
11  # applications/hello-world/hello-world.cpp). Therefore the "include" folder is a
12  # public include directory for the "greeter" library. The paths are different
13  # when building the library and when installing it, so generator expressions are
14  # used to distinguish between these two cases.
15  # See https://cmake.org/cmake/help/latest/command/target_include_directories.html
16  # for more information.
17  # If you have private headers in the "src" folder, these have to be added as
18  # well. They are private because they are only needed when building the library,
19  # not when using it from a different implementation file.
20  target_include_directories(greeter
21      PUBLIC
22          $<INSTALL_INTERFACE:include>
23          $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
24      PRIVATE
25          $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/src>
26  )
27  # Enable C++17
28  target_compile_features(greeter PUBLIC cxx_std_17)
29
30  # Add an alias with the proper namespace to prevent collisions with other
31  # packages.
32  add_library(greeter::greeter ALIAS greeter)
33
34  # Include the rules for installing the library
35  include(cmake/Install.cmake)
36
37  # Include the tests in the "test" folder.
38  add_subdirectory("test")
```

The unit tests

The test file only contains a single unit test, and just serves as an example. It uses the Google Test framework.

The tests can only be run on the build computer if we're not cross-compiling, that's why the call to `gtest_discover_test(...)` is conditional.

`greeter.test.cpp`

```cpp
1   #include <greeter/greeter.hpp>
2   #include <gtest/gtest.h>
3   #include <sstream>
4
5   /**
6    * @test
7    *
8    * Check that the output of the greeter::sayHello function matches the
9    * documentation.
10   */
11  TEST(greeter, sayHello) {
12      std::ostringstream ss;
13      greeter::sayHello("John Doe", ss);
14      EXPECT_EQ(ss.str(), "Hello, John Doe!\n");
15  }
```

test/CMakeLists.txt

```
1   find_package(GTest MODULE REQUIRED)
2
3   # Add a new test executable with the name "greeter.test" that is compiled from
4   # the source file "greeter.test.cpp".
5   add_executable(greeter.test
6       "greeter.test.cpp"
7   )
8
9   # The test executable requires the "greeter" library (it's the library under
10  # test), as well as the Google Test main function to actually run all tests.
11  target_link_libraries(greeter.test
12      PRIVATE
13          greeter
14          GTest::gtest_main
15  )
16
17  # Only look for tests if we're not cross-compiling. When cross-compiling, it's
18  # not possible to run the test executable on the computer that's performing the
19  # build.
20  if (NOT CMAKE_CROSSCOMPILING)
21      include(GoogleTest)
22      gtest_discover_tests(greeter.test)
23  endif()
```

The main Hello World program

Finally, the Greeter library can be used to create a simple Hello World program.

hello-world.cpp

```
1   #include <greeter/greeter.hpp> // Our own custom library
2
3   #include <iostream> // std::cout, std::cin
4   #include <string>   // std::getline
5
6   int main(int argc, char *argv[]) {
7       std::string name;
8       if (argc > 1) {     // If the user passed arguments to our program
9           name = argv[1]; // The name is the first argument
10      } else {            // If not, ask the user for his name
11          std::cout << "Please enter your name: ";
12          std::getline(std::cin, name);
13      }
14      greeter::sayHello(name, std::cout); // Greet the user
15  }
```

CMakeLists.txt

```
1   # Add a new executable with the name "hello-world" that is compiled from the
2   # source file "hello-world.cpp".
3   add_executable(hello-world
4       "hello-world.cpp"
5   )
6
7   # The "hello-world" program requires the "greeter" library.
8   # The target_link_libraries command ensures that all compiler options such as
9   # include paths are set correctly, and that the executable is linked with the
10  # library as well.
11  target_link_libraries(hello-world
12      PRIVATE
13          greeter::greeter
14  )
15
16  include("cmake/Install.cmake")
```

Compiling the example project

Using Visual Studio Code

1. Open this repository (`RPi-Cpp-Toolchain`) in Visual Studio Code (e.g. using `Ctrl+K O`).

2. You will be prompted *"Would you like to configure project 'RPi-Cpp-Toolchain'?"*. Click "Yes".
   (If this prompt doesn't appear automatically, click the **"⚒ No Kit Selected"** button at the bottom of the window.)

3. Select the configuration that matches your specific board, e.g. *Raspberry Pi 3 (AArch64)*. CMake will now configure the project for you.

4. Click the **"⚙ Build"** button at the bottom of the window to compile the library, tests and examples.
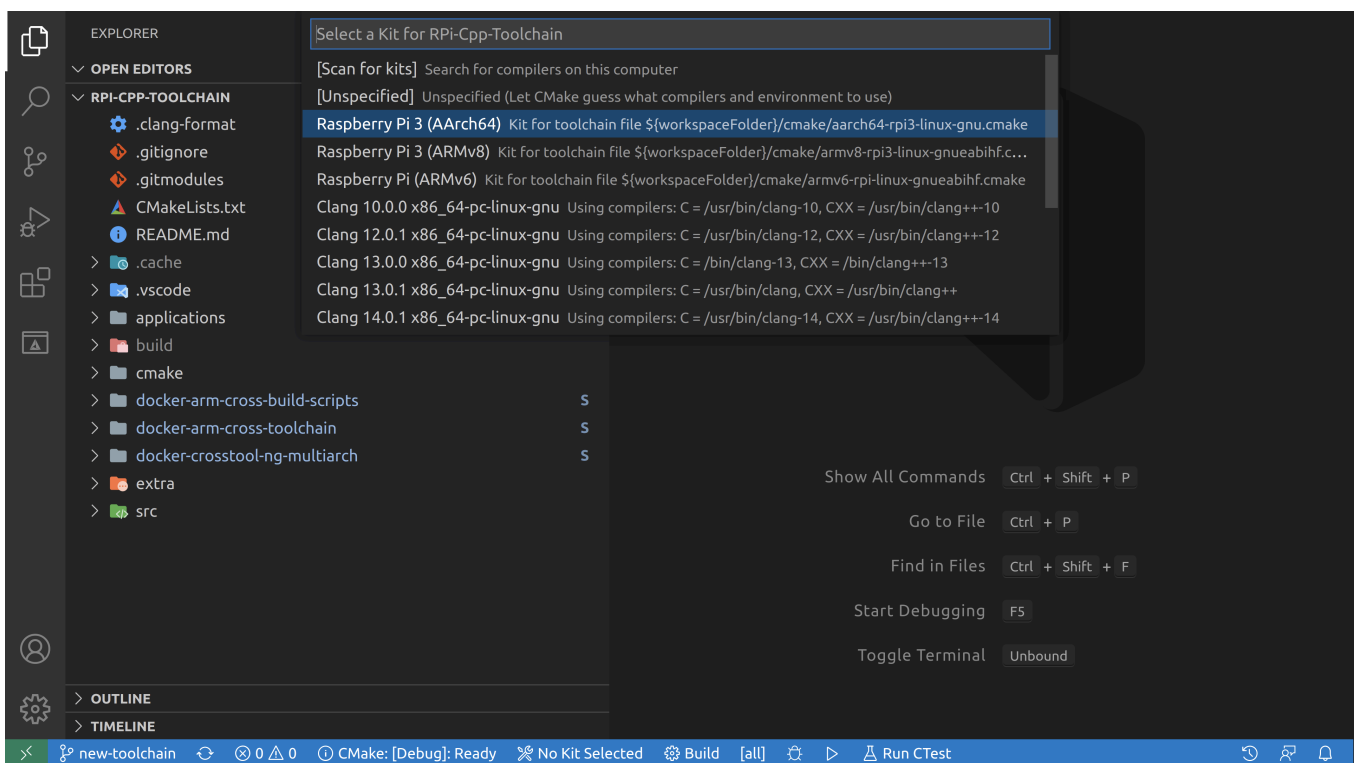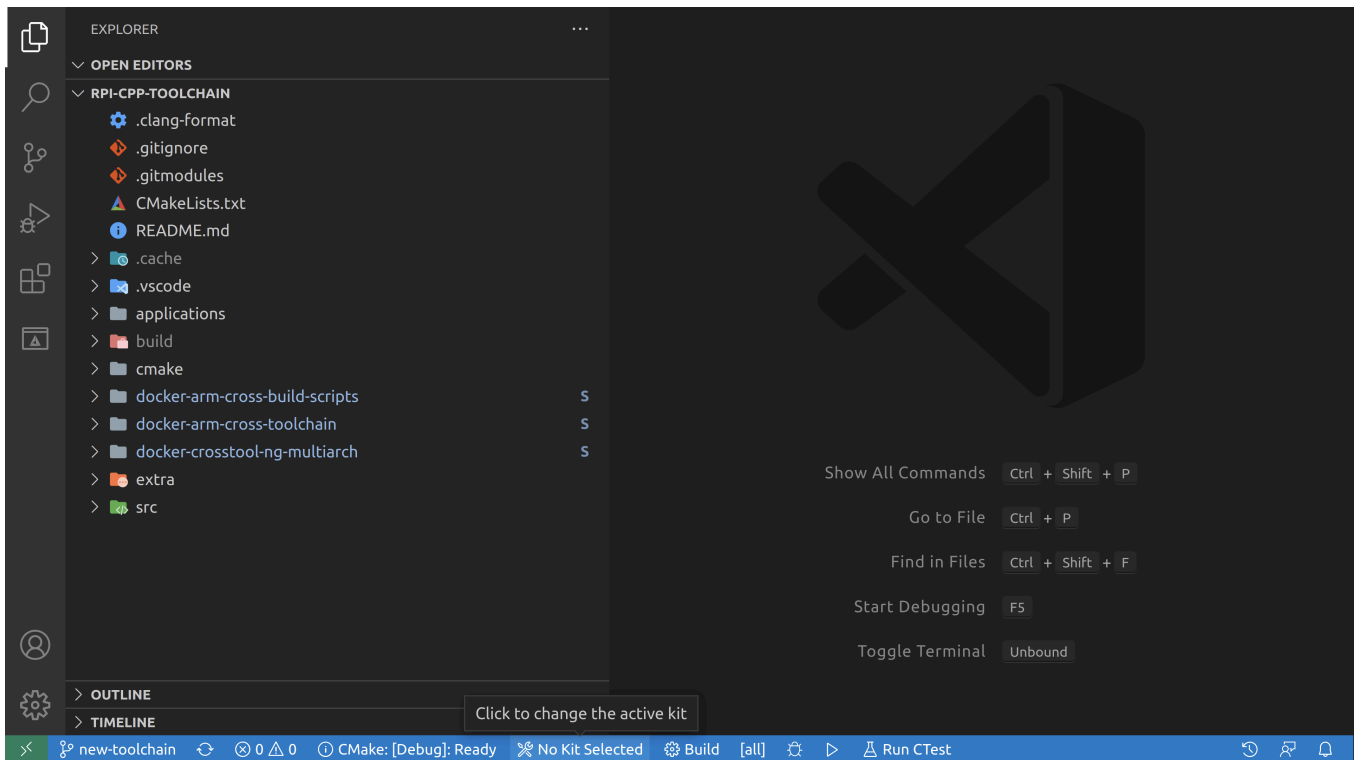
5. Package the project.
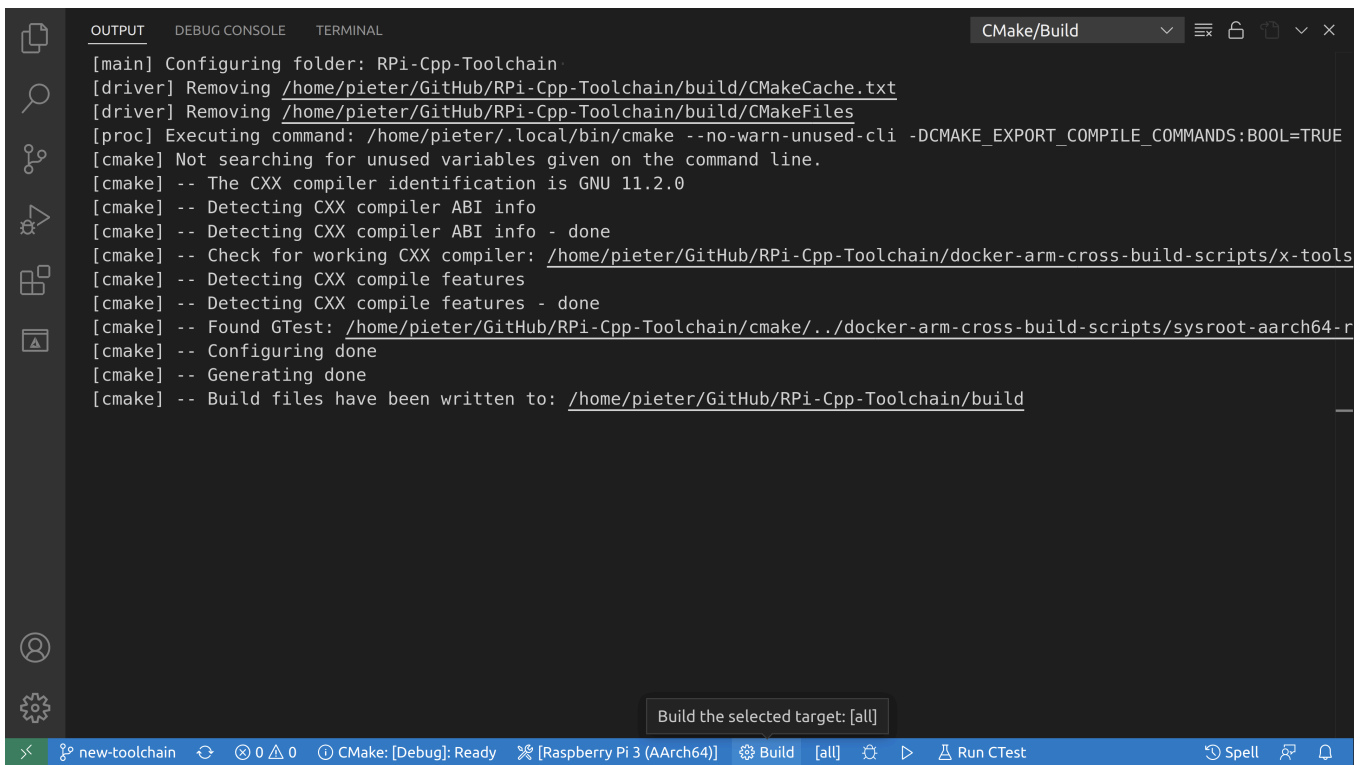
```
pushd build; cpack; popd
```

6. Copy the project to the Raspberry Pi.

```
ssh RPi3 tar xz < build/greeter-1.0.0-Linux-arm64.tar.gz
```

7. Run the hello world program on the Pi.

```
ssh -t RPi3 greeter-1.0.0-Linux-arm64/bin/hello-world
```
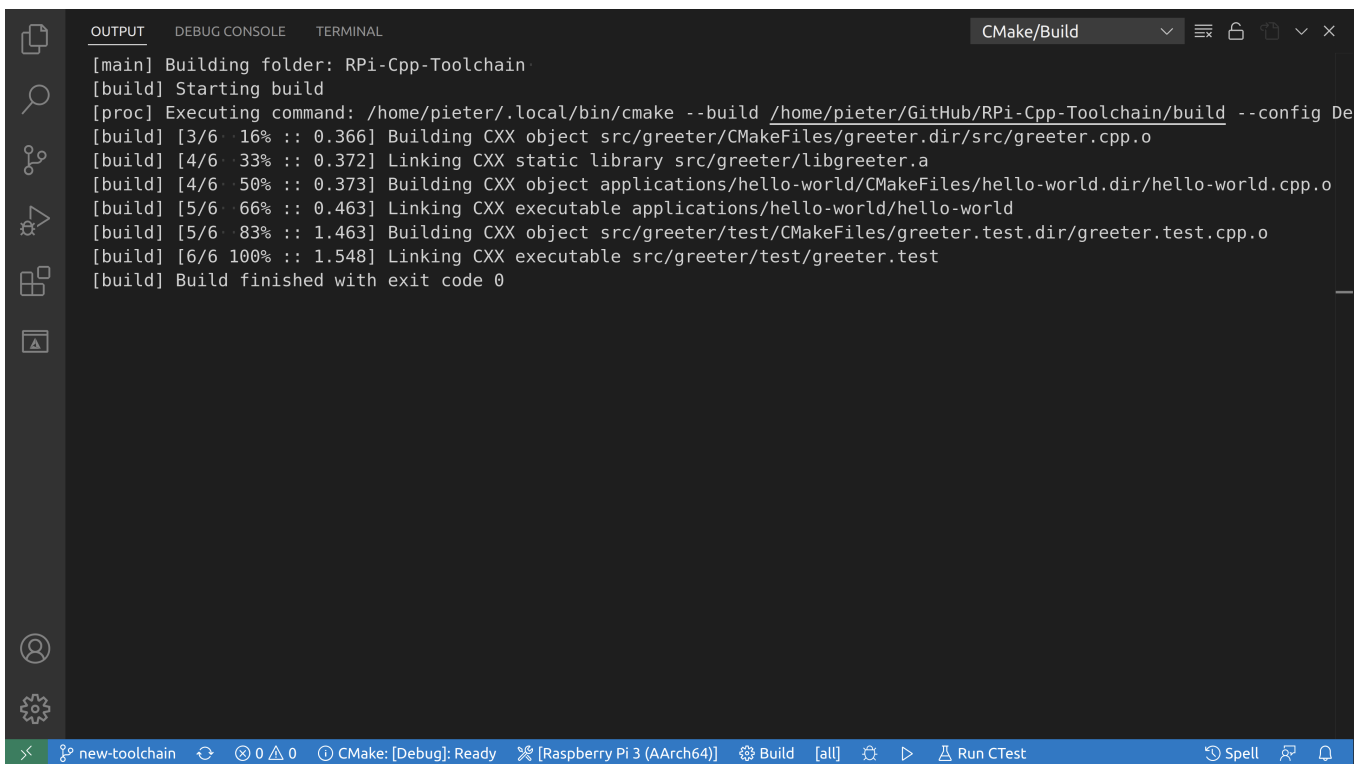
```
OUTPUT    DEBUG CONSOLE    TERMINAL                                          CMake/Build    ⌄    ☰ ⛶ ⛗ ⌃ ⌄ ✕
[main] Configuring folder: RPi-Cpp-Toolchain
[driver] Removing /home/pieter/GitHub/RPi-Cpp-Toolchain/build/CMakeCache.txt
[driver] Removing /home/pieter/GitHub/RPi-Cpp-Toolchain/build/CMakeFiles
[proc] Executing command: /home/pieter/.local/bin/cmake --no-warn-unused-cli -DCMAKE_EXPORT_COMPILE_COMMANDS:BOOL=TRUE
[cmake] Not searching for unused variables given on the command line.
[cmake] -- The CXX compiler identification is GNU 11.2.0
[cmake] -- Detecting CXX compiler ABI info
[cmake] -- Detecting CXX compiler ABI info - done
[cmake] -- Check for working CXX compiler: /home/pieter/GitHub/RPi-Cpp-Toolchain/docker-arm-cross-build-scripts/x-tools
[cmake] -- Detecting CXX compile features
[cmake] -- Detecting CXX compile features - done
[cmake] -- Found GTest: /home/pieter/GitHub/RPi-Cpp-Toolchain/cmake/../docker-arm-cross-build-scripts/sysroot-aarch64-r
[cmake] -- Configuring done
[cmake] -- Generating done
[cmake] -- Build files have been written to: /home/pieter/GitHub/RPi-Cpp-Toolchain/build
```
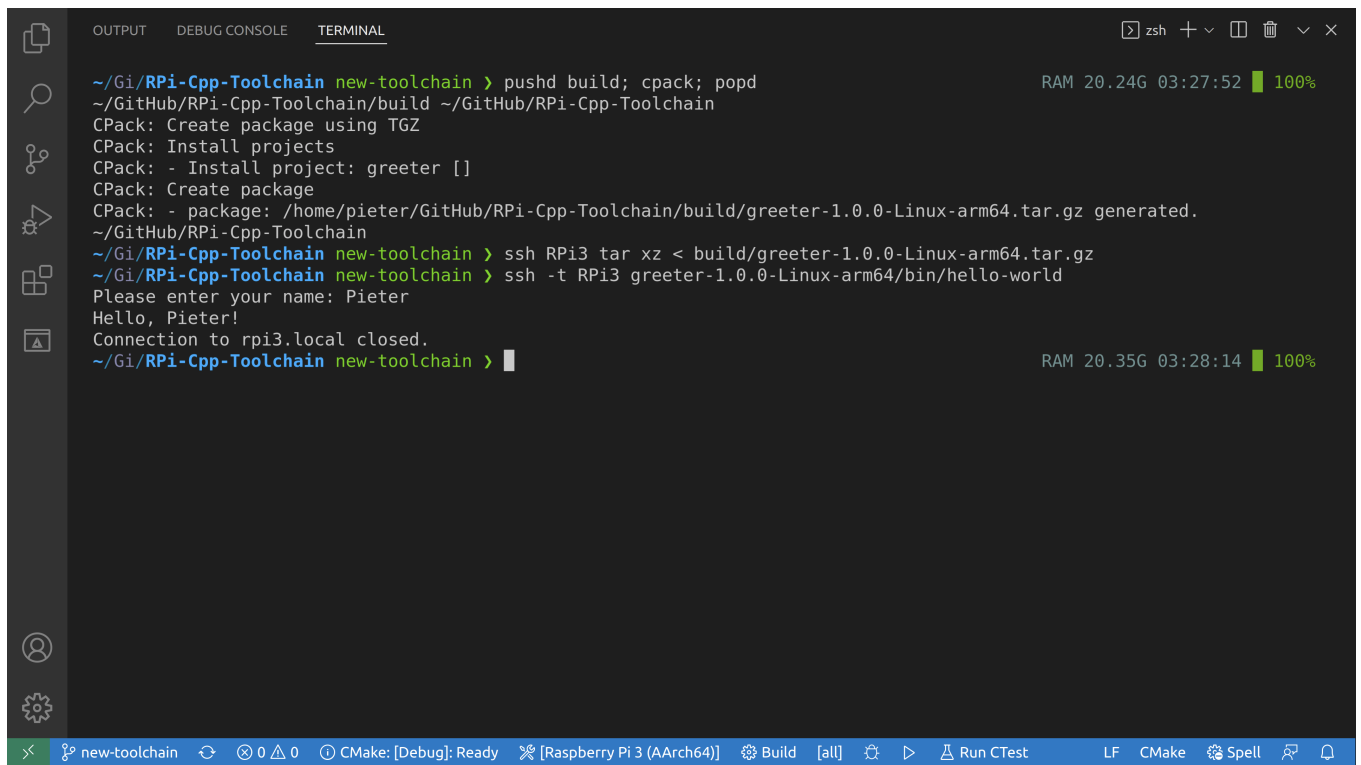
Build the selected target: [all]

new-toolchain   ⟳   ⊗ 0 ⚠ 0   ⓘ CMake: [Debug]: Ready   ⚒ [Raspberry Pi 3 (AArch64)]   ⚙ Build   [all]   ⛭   ▷   ⚗ Run CTest   🕘 Spell   🖈   🔔


```
OUTPUT    DEBUG CONSOLE    TERMINAL                                          CMake/Build    ⌄    ☰ ⛶ ⛗ ⌃ ⌄ ✕
[main] Building folder: RPi-Cpp-Toolchain
[build] Starting build
[proc] Executing command: /home/pieter/.local/bin/cmake --build /home/pieter/GitHub/RPi-Cpp-Toolchain/build --config De
[build] [3/6  16% :: 0.366] Building CXX object src/greeter/CMakeFiles/greeter.dir/src/greeter.cpp.o
[build] [4/6  33% :: 0.372] Linking CXX static library src/greeter/libgreeter.a
[build] [4/6  50% :: 0.373] Building CXX object applications/hello-world/CMakeFiles/hello-world.dir/hello-world.cpp.o
[build] [5/6  66% :: 0.463] Linking CXX executable applications/hello-world/hello-world
[build] [5/6  83% :: 1.463] Building CXX object src/greeter/test/CMakeFiles/greeter.test.dir/greeter.test.cpp.o
[build] [6/6 100% :: 1.548] Linking CXX executable src/greeter/test/greeter.test
[build] Build finished with exit code 0
```

new-toolchain   ⟳   ⊗ 0 ⚠ 0   ⓘ CMake: [Debug]: Ready   ⚒ [Raspberry Pi 3 (AArch64)]   ⚙ Build   [all]   ⛭   ▷   ⚗ Run CTest   🕘 Spell   🖈   🔔

```
OUTPUT   DEBUG CONSOLE   TERMINAL                                                    ⟩ zsh  + ∨  ▯  🗑  ∨  ✕

~/Gi/RPi-Cpp-Toolchain new-toolchain ❯ pushd build; cpack; popd          RAM 20.24G 03:27:52 ▌ 100%
~/GitHub/RPi-Cpp-Toolchain/build ~/GitHub/RPi-Cpp-Toolchain
CPack: Create package using TGZ
CPack: Install projects
CPack: - Install project: greeter []
CPack: Create package
CPack: - package: /home/pieter/GitHub/RPi-Cpp-Toolchain/build/greeter-1.0.0-Linux-arm64.tar.gz generated.
~/GitHub/RPi-Cpp-Toolchain
~/Gi/RPi-Cpp-Toolchain new-toolchain ❯ ssh RPi3 tar xz < build/greeter-1.0.0-Linux-arm64.tar.gz
~/Gi/RPi-Cpp-Toolchain new-toolchain ❯ ssh -t RPi3 greeter-1.0.0-Linux-arm64/bin/hello-world
Please enter your name: Pieter
Hello, Pieter!
Connection to rpi3.local closed.
~/Gi/RPi-Cpp-Toolchain new-toolchain ❯ ▌                                  RAM 20.35G 03:28:14 ▌ 100%
```

```
✕   ⌥ new-toolchain  ↻  ⊗ 0 ⚠ 0  ⓘ CMake: [Debug]: Ready  ✂ [Raspberry Pi 3 (AArch64)]  ⚙ Build  [all]  ☼  ▷  ⚖ Run CTest      LF  CMake  ⚙ Spell  ⟳  🔔
```

Using the command line

```
1   # See what toolchain files are available, use the one that matches your board.
2   ls cmake
3   # Configure the project using the correct toolchain.
4   cmake -S. -Bbuild \
5       -DCMAKE_TOOLCHAIN_FILE="cmake/aarch64-rpi3-linux-gnu.cmake" \
6       -DCMAKE_BUILD_TYPE=Debug
7   # Build the project.
8   cmake --build build -j
9   # Package the project.
10  pushd build; cpack; popd
11  # Copy the project to the Raspberry Pi.
12  ssh RPi3 tar xz < build/greeter-1.0.0-Linux-arm64.tar.gz
13  # Run the hello world program on the Pi.
14  ssh -t RPi3 greeter-1.0.0-Linux-arm64/bin/hello-world
```