Raspberry Pi 4 Environment Configuration

Jaidon Lybbert, University of Washington

January 24, 2024

Abstract

This report documents the setup and configuration of a portable development environment for a headless Raspberry Pi 4. The environment consists of a Windows 10 laptop, iPhone 13 Pro, and Raspberry Pi 4B. The Pi is pre-configured with the SSID and password to connect to the iPhone hotspot, and accessed from the laptop through SSH. A second home WiFi network is added with priority. The Pi is set up with the Neovim editor for on-device development, with language support for Python, Rust, C, C++, and Zig. For debugger support, the Windows laptop is set up with Visual Studio and VSCode with extensions for the same languages.

1 Introduction

This document is organized as follows. In Section 2 I describe my experience with embedded systems and software, motivations and intent. I describe the setup of my environment in Section 3 and perform some benchmarking tests. I show the benchmark results in Section 4, and discuss the results along with the failures and hurdles I faced in Section 5. I conclude in 6 by discussing the improvements I would like to make in the future.

2 Background

2.1 Programming Experience

I graduated with my BS in Electrical Engineering at Eastern Washington University in 2022, with a concentration in Embedded Systems. As part of

my coursework, I programmed an FPGA in VHDL to implement a processor supporting a subset of the MIPS instruction set. I did several small projects using the TI ARM® Cortex®-M4F Based TM4C123G LaunchPad™ MCU evaluation board [6], and was introduced to real-time operating systems using the Arduino Mega which I used to make a kitchen timer[9] with polling and pre-emptive scheduling.

While at EWU, I was an officer in the IEEE Student Chapter, and made a controller for a skittle-sorting machine, again using the TM4C123G [11].

For two years, I worked at an aerospace startup developing real-time control software for a pressurized gas system using an Allen-Bradley PLC.

I spent the summer of 2022 doing an internship at SpaceX, where I developed tools in Python to automate workflows and manage parts data.

My graduate studies have been in high-performance parallel computing on the NVIDIA Cuda platform, with some graphics programming in OpenGL and Vulkan, and some deep neural networks. My most recent work has been developing a parallel QR decomposition algorithm with Amazon Lab126 for in-home robotics applications[8].

2.2 Experience with the Raspberry Pi

I have done some projects with the Raspberry Pi 3B, and Raspberry Pi-ZeroW. For a time, I had the Raspberry Pi 3B set up as an ad-blocking DNS server using the Pi-hole project [1], and I used the ZeroW for a weather station project as an undergrad in IEEE [5].

2.3 Motivation

My experience with bare-metal programming and parallel programming on GPUs has given me a great appreciation for on-device debuggers, and memory management. My recent interests have been in programming languages themselves, and the tradeoffs they offer. At the same time, I have been becoming more mobile, and work in a variety of places from my laptop, leaving the desktop, two monitors, and mouse at home. The single low-res screen and trackpad have led me down the path of keyboard shortcuts, macros, and Neovim to ergonomically switch between windows and edit code without using the trackpad.

These motivations form the basis of the requirements for my development environment (plus version control as a given).

- Portability
- Version control
- Support for several languages
- On-device debugging
- Compatibility with Neovim

3 Methods

3.1 Installation

The Raspberry Pi 4B is a relatively simple device to setup following the documentation [4]. I used a 16GB SD card and flashed the 64-Bit Raspberry Pi OS "Bookworm" to it on my Windows 10 laptop with the provided Raspberry Pi Imager tool. I took a screenshot of the hostname and username configuration, so I wouldn't forget it.

3.2 Portability

In advanced settings in the Imager tool, the pre-configured WiFi network was automatically filled, along with the public SSH key from my laptop. I changed the SSID and password for the WiFi network to my iPhone hotspot so I have the option to physically bring my Pi with me and connect with SSH.

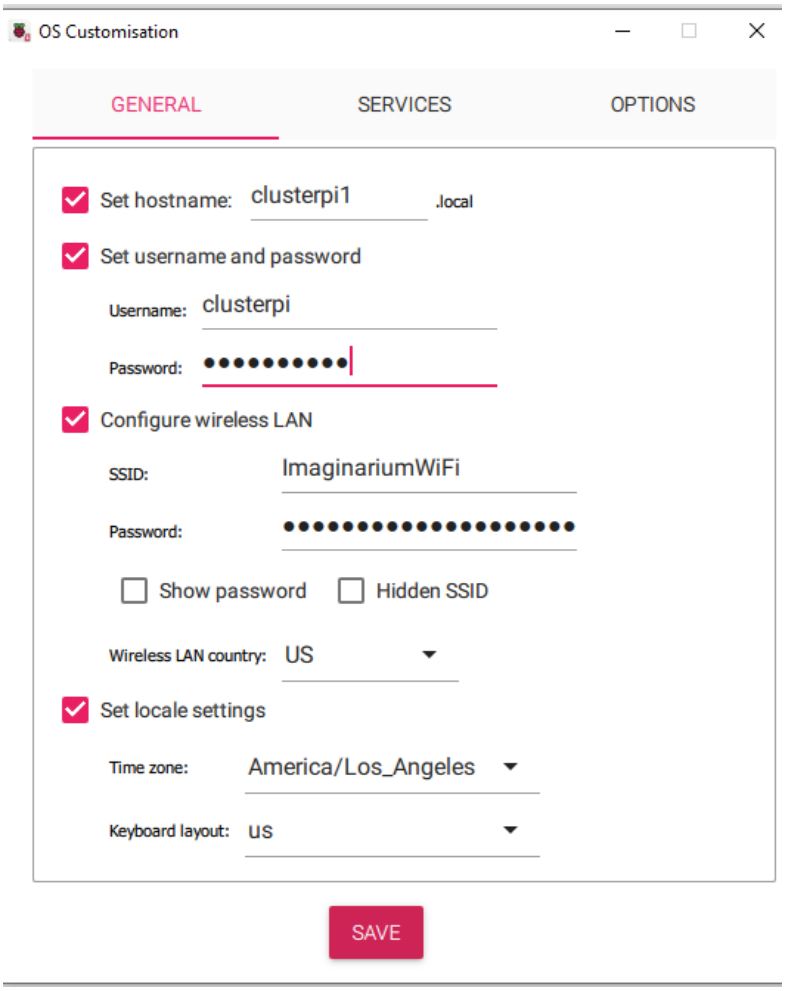

Figure 1: The Raspberry Pi Imager showing hostname and username configuration.

The Pi connected to the hotspot on boot without issue, and I was able to SSH into it by the hostname. I immediately scanned for my home WiFi network, and added it with higher priority using the Network Manager CLI. Switching networks caused the SSH session to hang, but switching networks on my laptop and restarting the SSH connection went without issue.

I updated the Pi with APT through my home network.

```
clusterpi@clusterpi1:~ $ nmcli --fields autoconnect-priority,name connection
AUTOCONNECT-PRIORITY  NAME
10                    6027 WiFi
0                     lo
0                     preconfigured
-999                  Wired connection 1
```

Figure 2: WiFi network prioritization in Network Manager. The mobile hotspot was labeled "preconfigured" by default, the home network is named by the SSID "6027 WiFi".

3.3 Version Control

Once my packages were up to date, I generated an SSH key pair with `ssh-keygen` and copied the public key to my personal Github account. I created a new repository [10] and cloned it to the Raspberry Pi.

cd git clone git@github.com:jaidonlybbert/EEP522A-W24.git

3.4 Language Support

At this point I checked the remaining space on my SD card with the `df -H` command. I would keep an eye on it as I began installing packages.

```
clusterpi@clusterpi1:~/.ssh $ df -H
Filesystem      Size  Used Avail Use% Mounted on
udev            1.8G     0  1.8G   0% /dev
tmpfs           398M  1.3M  397M   1% /run
/dev/mmcblk0p2   16G  4.7G  9.6G  33% /
tmpfs           2.0G  144k  2.0G   1% /dev/shm
tmpfs           5.3M   17k  5.3M   1% /run/lock
/dev/mmcblk0p1  535M   76M  459M  15% /boot/firmware
tmpfs           398M   37k  398M   1% /run/user/1000
```

Figure 3: Free space on the SD card after basic installation. The filesystem has 9GB available.

The full set of languages I want to support are:

- Python

- Rust
- C
- Zig
- C++

I found GCC and Python came pre-installed with the OS. I installed Rust by fetching and executing the installer with the one-liner

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.
   rustup.rs | sh
```

as suggested by the docs [3].

I attempted to compile Zig from source using the zig-bootstrap [7] repository. The only dependency missing to compile Zig was CMAKE, which I planned to install anyway. I installed Ninja as well to speed up the build process.

I got the triplet for the raspberry pi using

```
gcc -dumpmachine
```

I set the CMAKE environment variable to use Ninja instead of Make.

```
export CMAKE_GENERATOR=Ninja
```

Then ran the build script for Zig

```
./build aarch64-linux-gnu native
```

It occured to me as the build spooled up that this was the first real workload I put on the device. I gave the small adhesive heatsink I put on the SoC an ill-advised finger test and determined it was getting hot. As it turns out the Pi is quite slow relative to the size of the job of compiling Zig. It would take several hours. After looking at the repository again, I realized that I wasn't just compiling Zig, I was also compiling LLVM, LLD, Clang, zlib, and zstd. Ultimately, the build failed when my laptop went into hibernate mode and the SSH session disconnected.

I went the easy route and just downloaded the released binary for the aarch64 architecture.

```
wget https://ziglang.org/builds/zig-linux-aarch64
   -0.12.0-dev.2338+9d5a133f1.tar.xz
tar -xf zig-linux-aarch64-0.12.0-dev.2338+9d5a133f1.
   tar.xz
```

And added Zig to my PATH variable permanently by opening the bash configuration with Neovim.

```
nvim ~/.bashrc
```

Then pasting in the following, to append the PATH environment variable.

```
export PATH=$PATH:/home/clusterpi/zig/zig-linux-
   aarch64-0.12.0-dev.2338+9d5a133f1
```

3.4.1 Neovim Setup on Device

A development environment is set up on the device using Neovim with language support for Python, Rust, C, C++, and Zig. By default, Neovim supports syntax highlighting for these languages. For code completion, linting, and static analysis, language-specific plugins are added. Neovim supports the Lua scripting language for writing plugins and configuration.

To get a configuration quickly, I used the kickstart-nvim Github repository[2], which requires the latest version of Neovim, *not* the version found in the default package repositories on the Pi. I uninstalled the old version of Neovim, and compiled the latest version of Neovim from source.

```
sudo apt uninstall neovim
git clone https://github.com/neovim/neovim
cd neovim && make CMAKE_BUILD_TYPE=RelWithDebInfo
sudo make install
```

Neovim took a reasonable 10 minutes or so to compile. I then followed the kickstart-nvim instructions.

```
mkdir ~/.config/nvim
git clone https://github.com/nvim-lua/kickstart.nvim
   .git "${XDG_CONFIG_HOME:-$HOME/.config}"/nvim
```

Starting Neovim runs the configuration script, and downloads about 25 plugins greatly extending the functionality. I used the `:meson` command in Neovim to search and install language servers for Python, Rust, and Zig as shown in Figure 4. The popular language server for C and C++ is `clangd` which unfortunately did not install successfully on the Pi.

As an example, some of the power of the Rust Analyzer is demonstrated in Figure 5. After typing "ab" on a line of the Rust application in the Neovim editor, the language server guesses that you are typing "abort" and

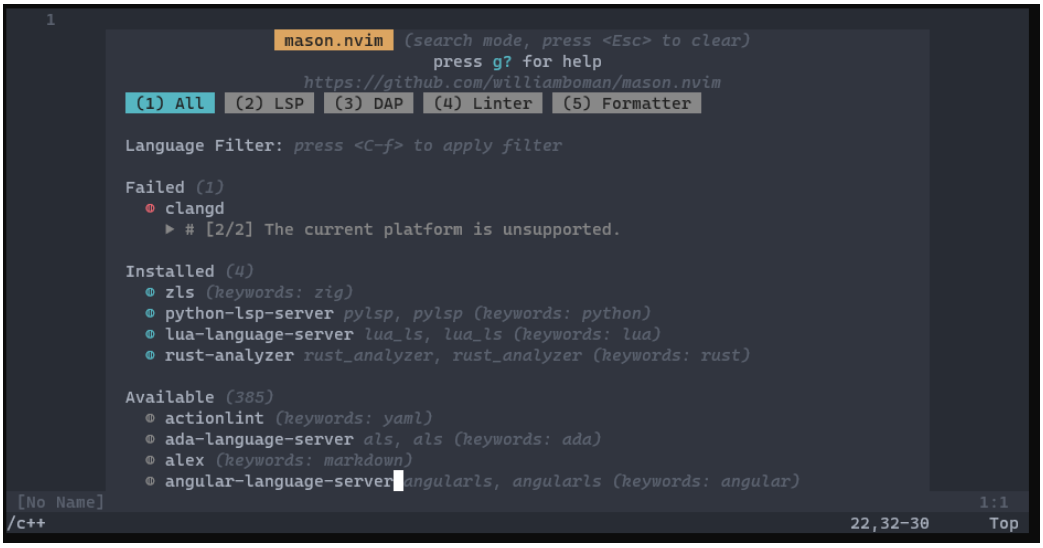

Figure 4: Language servers installed through the Mason package manager in Neovim.

gives you information about how the function works, and gives the option to autocomplete.

3.4.2 VSCode

A VSCode-based development environment is set up on the Windows 10 machine for instances when a debugger is necessary, with extensions for Python, Rust, and Zig. These extensions add syntax highlighting, language servers, and linting to VSCode.

- Python - Microsoft
- rust-analyzer - The Rust Programming Language
- Zig Language - ziglang
- Zig Language Extras - Igor Anic

Figure 5: The Rust Analyzer providing context and autocompletion in the Neovim editor running on the Raspberry Pi accessed through SSH.

3.4.3 Visual Studio

Visual Studio is used for C and C++, since those projects are built with MSBuild through CMAKE, which outputs Visual Studio projects. I installed the Visual Studio 2022 Community Edition.

3.4.4 Hello World!

I created a "src" directory in my repository, and a subdirectory for a "hello world" app in each language.

```
mkdir ~/EEP522A-W24/src
cd ~/EEP522A-W24/src
mkdir hello_python hello_c hello_rust hello_zig
```

From inside each folder I initialized a "hello world" project with their respective methods.

Python For Python I simply created a .py file with Neovim and put a single "hello world" print statement in it, then ran the file with the interpreter.

```
nvim hello_world.py
python hello_world.py
```

Rust Rust uses Cargo to build and run applications. Initializing a new Cargo package creates a "hello world" application by default.

```
cargo new hello_rust
cd hello_rust
cargo run
```

C For C, I asked Bing Chat to "create a template CMakeLists.txt for a single c file called 'hello.c'" and pasted the response with Neovim. I asked Bing Chat to "create a hello.c program in C which prints 'hello world!'" and pasted the response.

```
nvim CMakeLists.txt
nvim hello.c
```

Then ran CMake to build the project on the device.

```
mkdir build
cd build
cmake ..
cmake --build .
```

The project compiled with GCC and ran without issue.

```
./HelloWorld
```

I repeated the CMake build process on my Windows machine, and verified the build worked correctly with MSBuild.

Zig When you initialize a new project in Zig, some test code is created to print out a message. All that is needed is the following.

```
cd hello_zig
zig init
zig build run
```

The test code prints *All your codebase are belong to us.*

3.4.5 Debugger Test

I opened the ./src/ folder of my repository in VSCode on my Windows machine and set a breakpoint at the first line of each "hello world" program for Python, Rust, and Zig. I verified that the debugger was working for each one. For C, I opened the built project in Visual Studio, set a breakpoint and ran the project in Debug mode to verify the debugger was working.

3.4.6 Profiling Challenge

I created a new directory in my repository called "data".

```
mkdir ~/EEP522A-W24/data
cd ~/EEP522A-W24/data
```

I downloaded the `.zip` file from our class Canvas page to the repository on my Windows machine and pushed it to Github, then pulled the changes on the Pi.

I extracted the folder.

```
unzip ee_course.zip
```

I ran the shell script without changes to see the output.

```
./ee_course/run.sh
```

Following the guide in the assignment instructions I opened the `prototype.c` file in Neovim and jumped to the *void prototype translate information(char * model)* function. From the output of the program in it's broken state, I deduced that I needed to copy the board details, which are output by the command

```
cat /proc/cpuinfo
```

to the `g_core` struct.

I re-ran the shell script, pushed results to GitHub, and pulled them down to my laptop to look at.

4 Results

The images produced by the experiment are shown in Figures 7, 8, and 9.

```
// ** add more model types here
if (!strcmp(model, "c03115"))
{
g_core.model = 0xc03115;
g_core.memory_size_gb = 4;
g_core.revision = 1.5;
return;
}
```

Figure 6: Code added to 'prototype.c' for raspberry pi 4 support.

5 Discussion

5.1 Interpretation of Graphs

Figure 7 shows the correlation of time, temperature, and energy use relative to the number of memory tests performed. Although the units are not clear, what is clear is that temperature increases almost linearly with number of tests, while energy use is nearly exponential.

Figure 8 shows the difference in execution time that optimizing memory access patterns can make. By "working with the memory system" total execution time can be dramatically reduced.

Figure 9 shows execution time compared the number of tests performed. In this one, changing the y-axis scaling would improve readability. The obvious takeaway is that more tests take more time.

5.2 Issues Encountered

5.2.1 Initializing WiFi

I initially tried installing the Raspberry Pi OS using the same method I had used for the Raspberry Pi 3B several years ago. It used to be the case that you could pre-configure the WiFi connection by flashing the image to the SD card, then copying a `wpa_supplicant.conf` file to the boot partition before installing it to the device. This is no longer supported by the "Bookworm" Raspbery Pi OS and onwards [4].

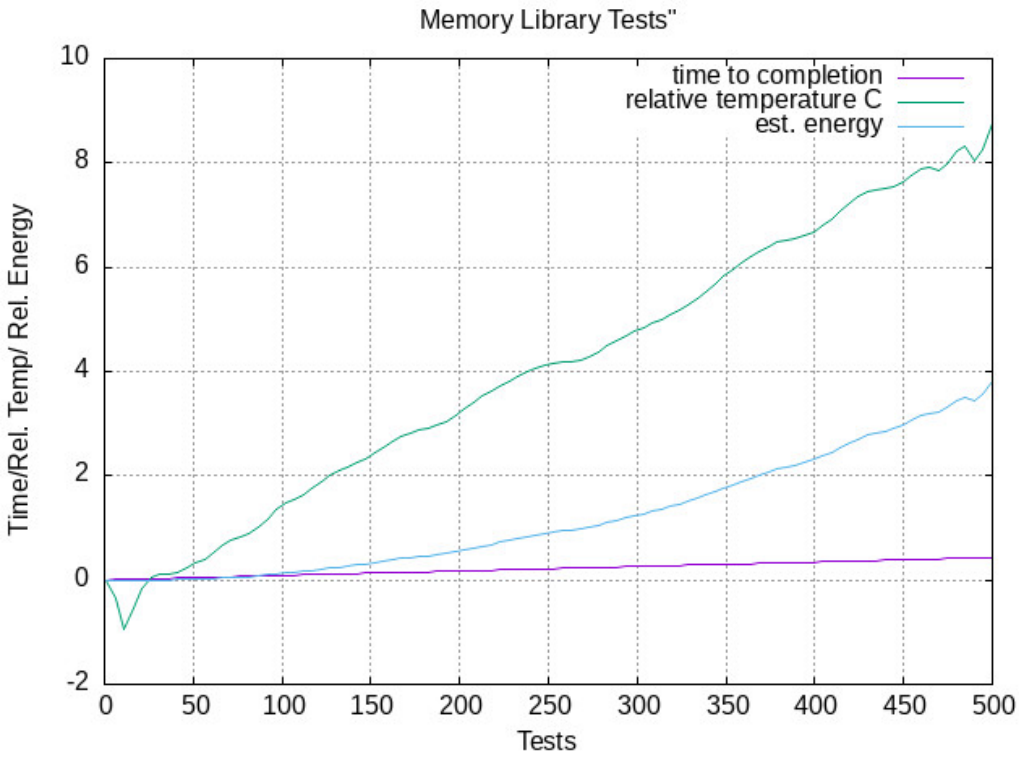

Figure 7: Output of the Memory Library test.

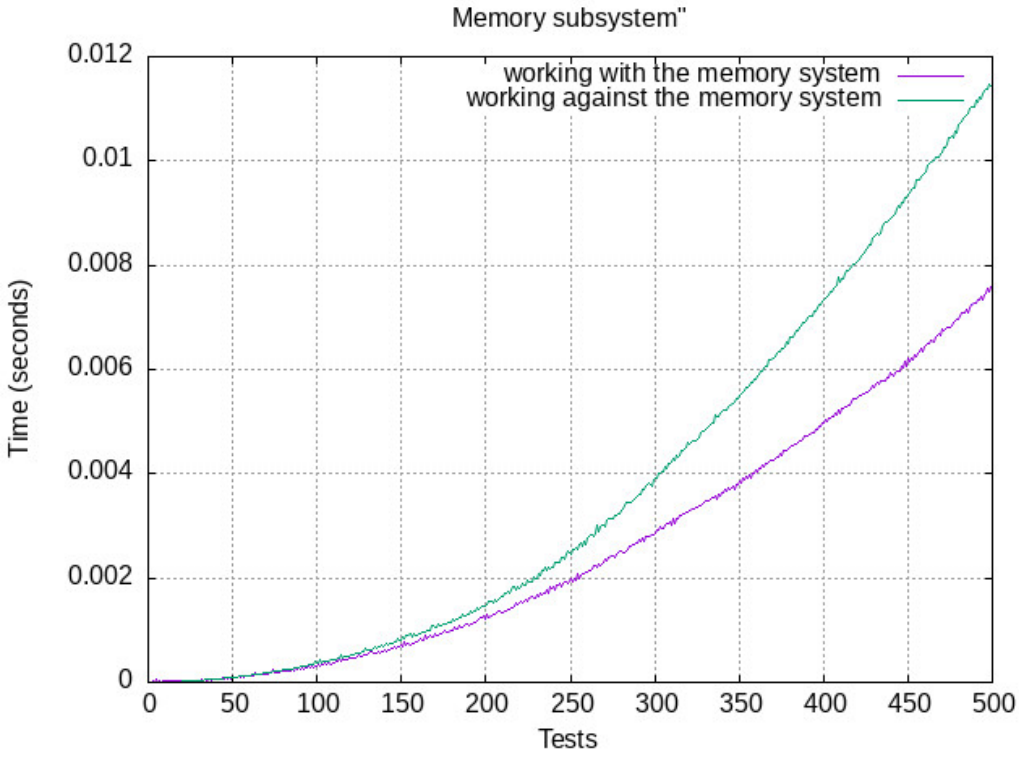

Figure 8: Output of the Memory Subsystem test.

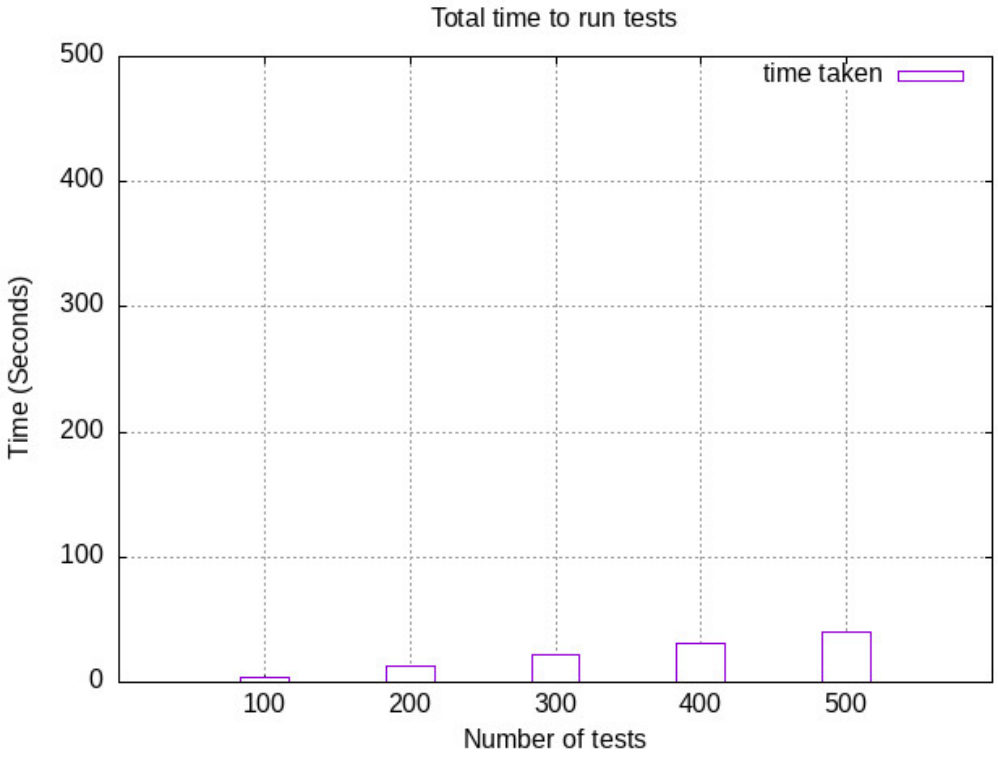

Figure 9: Output of the execution time test.

5.2.2 Compiling Zig from Source

Compiling Zig turned out to be more computationally intensive than I expected, and my SSH session was terminated by my host machine going into hibernation mode. If I were to try it again, I would install a fan on the Pi, and use a session manager like `tmux` to preserve the compilation process after disconnecting my SSH session.

6 Conclusion

The result of this work is a headless Raspberry Pi set up with prioritized WiFi connections to a mobile hotspot, and my home network. I am able to edit code with syntax highlighting, code completion, and "peak definition"-type functions using a modified configuration of Neovim on the device through an SSH connection. My code is synced to GitHub, and I am able to transfer and step-debug code on my Windows machine. Python, Rust, Zig, C, and C++ are supported by the environment.

I fell short of being able to do on-device debugging. This could become a problem if I start programming using significant device-specific code. In the future, I would like to explore using GDB directly through the shell for some basic debug functionality, and explore other options for dumping memory regions for analysis. I also noticed that there are plugins for Neovim supporting some debugging for some languages, which I will explore further. I saw with `clangd` that not all plugins are available on the Raspberry Pi.

Note *All my code and project files for this and future reports, can be found on my GitHub repository [10].*

References

[1] Pi-hole: A black hole for internet advertisements. `https://pi-hole.net/`. Accessed: 2024-01-22.

[2] Christian Chiarulli. Kickstart.nvim: A neovim configuration template. `https://github.com/ChristianChiarulli/kickstart.nvim`, 2021. Accessed: 2024-01-25.

[3] The Rust Project Developers. Installation. https://www.rust-lang.org/learn/get-started. Accessed: 2024-01-23.

[4] Raspberry Pi Documentation. Configuration. https://www.raspberrypi.com/documentation/computers/configuration.html, 2021. Accessed on 2024-01-22.

[5] EWU-IEEE-Spokane. Weather-station. https://github.com/EWU-IEEE-Spokane/Weather-Station, 2024. Accessed: 2024-01-22.

[6] Texas Instruments. Ek-tm4c123gxl evaluation board. https://www.ti.com/tool/EK-TM4C123GXL. Accessed: 2024-01-22.

[7] Andrew Kelley and Zig Contributors. Zig bootstrap. https://github.com/ziglang/zig-bootstrap, 2021. Accessed: 2024-01-23.

[8] Jaidon Lybbert. Mixed-precision block qr. GitHub repository, 2020.

[9] Jaidon Lybbert. Kitchen timer. https://github.com/jaidonlybbert/EENG462/blob/main/Labs/Final/jjlybbertFinal/jjlybbertFinal.ino, 2023. Accessed: 2024-01-22.

[10] Jaidon Lybbert. Classwork for embedded and real-time systems, university of washington, winter 2024. https://github.com/jaidonlybbert/EEP522A-W24, 2024. Accessed: 2024-01-23.

[11] Jaidon Lybbert, Cody Birkland, sirchicdbruk, and mshel-don1986edu. Ewu-ieee-spokane/sorter-project. https://github.com/EWU-IEEE-Spokane/Sorter-Project, 2020. Accessed: 2024-01-22.