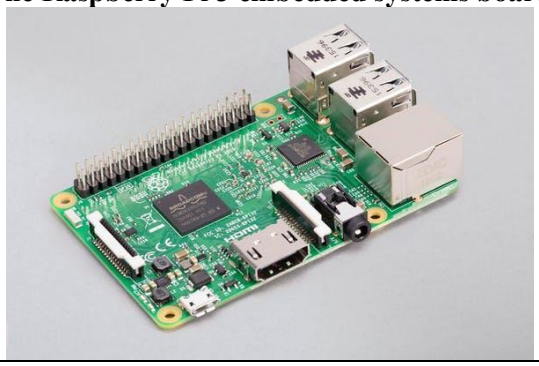
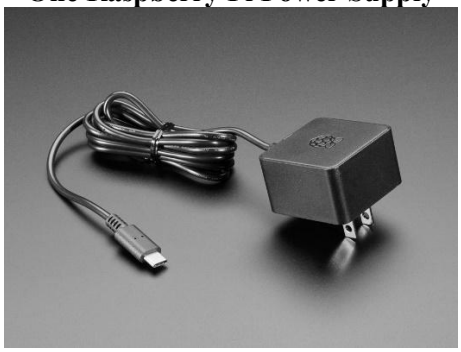
# SWE4211 Lab 1: Getting started with Raspberry Pi Development

## 1 Introduction

This is your first lab with the Raspberry Pi. In this lab, you will learn a bit about how the Raspberry Pi works, as well as start some basic development with the system.

Note that for this lab, you will be working with a lab partner. You will have one Pi setup, but both of you should follow the instructions to create an image and to setup the virtual machine (VM).

## 2 Laboratory Equipment Needed (For each group)

| | |
|---|---|
| **One Raspberry Pi 3 embedded systems board**  | **One Raspberry Pi Power Supply**  |
| **2 Ethernet networking cables**  | **1 USB uSD Card Reader (Note: May physically look slightly different)**  |
| **1 Trendnet 2.0 USB Ethernet 10/100 Adapter**  | |

## 3 AI Policy Statement

For this assignment, the usage of AI is **PERMITTED**. This assignment does not have any specific needs for the usage of AI, but from a pedagogical standpoint, there is no justification to forbid the usage of AI on this assignment.

# 4 Lab Objectives

- Create a microSD card image of an operating system
- Configure a Linux virtual machine for cross compilation

# 5 Deliverables / Submission

This lab has primarily been focused on learning about the environment. The lab report is therefore brief about what you found and what you learned. You are to submit this with your lab partner. One submission between the two of you is fine.

Each person will be responsible for submitting one report with the following contents:

1. Introduction
   a. **What are you trying to accomplish with this lab? This section shall be written IN YOUR OWN WORDS. DO NOT copy directly from the assignment. Make sure it is in multiple, grammatically correct sentences.**
2. Screen captures
   a. **What was the output of the architecture commands when installing / printing foreign architecture?**
   b. **What was the output of the ls command for the g++ compilers?**
   c. **Include screen captures showing the execution of the first demo program (test1) on the Raspberry Pi and the uname command.**
   d. **Include a screen shot of the test1 program trying to run on the host and showing the file size.**
   e. **Include a screen shot of the Lab1Part3 program running on the Pi as well as on the host.**
3. Questions
   a. **Explain what is the difference between foreign architecture and simply an architecture?**
   b. **What was the binary that the g++ cross compiler linked to? Why is this structure advantageous to Linux?**
   c. **What is the size of the test1 binary when cross compiled? What was the size of the test2 binary when cross compiled?**
   d. **Why does the test1 file not run in your vm environment?**
   e. **What does the -static option do to the compiler/linker? Why might this be advantageous? What problems might this cause?**
   f. **What does the size of the pointer have a value of 8 mean? What would it mean if the size of the pointer was 4?**
   g. **Why is debugging harder on the raspberry pi using gdb than it was when you debugged code with the debugger in your C/C++ Programming course?**
   h. **What is meant by multi-architecture in terms of gdb?**
   i. **What does the gdb service do on the raspberry pi?**
4. Conclusions
   a. **What background material from previous courses that would have been useful to this lab do you feel you were missing?**
   b. **What have you learned with this experience?**
   c. **What improvements can be made in this experience in the future?**

This material should be submitted as a single pdf file.

If you have any questions, consult your instructor.

# 6 Introduction

In this lab, you are going to setup the Raspberry Pi and associated development systems.

In this lab writeup, and in future lab writeups, a convention will be used to help guide you through the exercises. In the lab writeups, instructions that are to be typed are color coded for whether they are to be typed into a plain Linux terminal or a terminal which is connected to the Raspberry Pi. Commands that appear in **RED** are to be entered in the shell of your VM and are to be executed by the VM itself. For example, `ls -al` will list all files in the current directory on Linux VM. Commands which are in **BLUE** are to be executed on the Raspberry Pi. To do this will require a ssh connection to your Pi. For example, **chmod a+x hello.sh** will change the permissions for the file hello.sh on the raspberry Pi to be executable for everyone.[1]

# 7 Prelab

Prior to the lab session, you will want to try and accomplish as many of these tasks as is possible, as these are setup tasks. Sometimes they go by very quickly, but other times, there may be downloads that take time to complete. If possible, your professor will provide materials to you in advance of lab potentially on flash drives.

## *7.1 Virtual Machine setup*

The Linux distribution you are using will execute using the VirtualBox virtual machine software. You will need to download this and install the VirtualBox software on your laptop. (Note: This is a newer version than you may have used for previous courses, and you should update to the latest version for best operation on your laptop machines.) You will need to install the appropriate version of the software, which is 64-bit for the MSOE machines. The download for VirtualBox will be present in Canvas. Once you have done this, download and install the Oracle Virtual Machine Extension Packs. This will allow full operation of USB devices and other things that require advanced virtualization technology. (Note: If you do not install the extensions, strange behaviors may be noted in later assignments.)

When you have completed installing Virtual box, you will need to download and install the Linux virtual machine image from Canvas. The virtual machine will run Debian. Debian has become the defacto standard for those who wish to work with the Raspberry Pi when the Raspberry Pi is running an embedded Debian distribution. As part of the installation process, you will need to create a folder on your c drive called lwshare, which is a folder that will be shared between Linux and your Windows operating system. You should create this before installing and running the VM

Once the Virtual machine is downloaded and setup, log onto the operating system. The default user is **swestudent**, and the default password is **M$OEStudentPwd**. At this point, if you would like to share another folder in addition to the default lwshare on the c drive, this would be the ideal time to set it up.[2]

## *7.2 Changing the name of your Virtual Machine*

One of the things we need to do is to differentiate our virtual machines. Right now, if all students were to log on at the same time, we'd see all of your VM's using the same name. That is not ideal for troubleshooting. So, what we want to do is to rename our VM so that we can figure out what is going on. Luckily, this is a very easy process.

To start, open a shell on your virtual machine. In the shell, enter the command `hostnamectl`. This will show you information about the given virtual machine. Next, enter the command `sudo hostnamectl set-hostname <newname>` where <newname> is the name you are giving your machine. Use something that is obvious for this course

[1] These examples should look familiar from your Operating Systems course, likely taken last year.
[2] This is done using the Shared Folders segment of VirtualBox. There are numerous tutorials online about this.

(i.e. your MSOE email id, your last name and first initial, etc., making sure to add in that it is the swe4211. For example, swe4211-debian-schilling).

Next, issue the command **`sudo nano /etc/hosts`**.[3] This will open up the file that is responsible for storing some information about the hostname for the given machine and other networking information. Change the name of the given machine to match the hostname you entered previously. When finished, save the file (**`Ctrl-O`** and then **`enter`**) and exit the editor (**`Ctrl-X`**). Next, cat the file out to the console to make sure your changes were saved appropriately using the command **`sudo cat /etc/hosts`**

Once you have done this, issue the command **`sudo reboot now`** and log onto the machine once the reboot completes. Open another shell and you should see the changed name when you type the command **`hostnamectl`**. These steps are shown in Figure 1.

```
swestudent@swe4211-debian-base:~$ hostnamectl
 Static hostname: swe4211-debian-base
       Icon name: computer-vm
         Chassis: vm
      Machine ID: 659ac051cd4542aab6ae8fdf86865fe4
         Boot ID: 6599e79d5cce4cbea793ba5ace3bcefa
  Virtualization: oracle
Operating System: Debian GNU/Linux 13 (trixie)
          Kernel: Linux 6.12.101+deb13-amd64
    Architecture: x86-64
 Hardware Vendor: innotek GmbH
  Hardware Model: VirtualBox
Firmware Version: VirtualBox
   Firmware Date: Fri 2006-12-01
    Firmware Age: 19y 8month 2w 6d
swestudent@swe4211-debian-base:~$ sudo hostnamectl set-hostname swe4211-debian-schilling
[sudo] password for swestudent:
swestudent@swe4211-debian-base:~$ sudo nano /etc/hosts
sudo: unable to resolve host swe4211-debian-schilling: Name or service not known
swestudent@swe4211-debian-base:~$ sudo nano /etc/hostname
swestudent@swe4211-debian-base:~$ sudo cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       swe4211-debian-schilling

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
swestudent@swe4211-debian-base:~$
```

**Figure 1 Resetting the name of your VM.**

[3] This instruction uses nano as the text editor. However, you are free to use any editor (vim, emacs, etc.) that has been installed on the VM environment.

# 8 Setting up the toolchain[4]

The virtual machine you are being provided with is a somewhat stock VM, in that it is designed to run Linux for the target platform, generally a 64 bit Intel platform. However, to work with the Raspberry Pi, we need our platform to be able to handle compilation for the arm platform. To do this, we will install a feature called foreign architectures. Foreign architectures, when used with a tool called multiarch allows us to compile targeting different architectures.

To do this, start by adding the appropriate arm architecture as a foreign architecture using the command **`sudo dpkg --add-architecture arm64`**. When this finishes, issue the commands **`dpkg --print-architecture`** and **`dpkg --print-foreign-architectures`**. This will show the architecture of the vm as well as the installed foreign architectures. Save a screenshot for your report.

Next, we need to update the VM with any new package sources and install the cross compiler. The cross compiler will allow us to compile code for the defined platform. However, before we can do this, we need to remove a source. Issue the command **`sudo nano /etc/apt/sources.list`**. In the editor, remove any references that may exist referencing "cdrom" as the source. When finished, save the file. Then, issue the command **`sudo apt update`** and **`sudo apt install crossbuild-essential-arm64`**. This will update the sources used for the OS as well as install the appropriate ARM cross compiler for the Raspberry Pi.

Next, issue the command **`clear`** and then **`cd /usr/bin`** to change into the path where the tools have been installed. Issue the commands **`ls -l *g++*`** and **`aarch64-linux-gnu-g++ -v`** to list the g++ compilers and the compiler version. If the result of the ls command is a link, try to identify the base binary that is being linked to. Save a screenshot for this report showing the outputs.

When finished, shutdown your virtual machine with the command **`sudo shutdown now`**.

[4] Based upon Molloy, D. (2016). Exploring Raspberry Pi: Interfacing to the Real World with Embedded Linux. New York: Wiley, 978-1119188687, 720pp.

# 9 Creating your bootable flash image on a uSD Card

There are many ways to create uSD images. We will follow what is becoming the de facto method when working with the Raspberry PI. However, other methods may also work, depending upon your experience and environment.

## *9.1 Burning the Image with Windows*

To begin, download the Raspberry PI imager software from Canvas and install it on your laptop. Place your uSD card into an external USD to USB adapter and then plug it into your laptop. When you have completed these steps, start the program. For the device, select the Raspberry Pi 3, choose an other operating system and then raspberry pi lite, and then select your device. **REALLY MAKE SURE YOU SELECT THE RIGHT DEVICE. MISTAKES HERE CAN BE UNRECOVERABLE. IT IS RECCOMENDED THAT YOU REMOVE ALL OTHER USB DRIVES FROM YOUR MACHINE WHEN DOING THIS ACTION.** Then click on the Next button. Set the hostname to a unique hostname which incorporates your MSOE student name / student id and then setup the localization to be in the central US. Note that setting this wrong can have very adverse impacts on the Wi-Fi setup. For the username select pi and then provide a simple, easy to remember password that you can readily type. In selecting your password, use something simple that you would not be afraid of sharing should it be necessary to help with debugging.[5]

Next, enter the Wi-Fi network. For the Wi-Fi network, consult canvas for details about the names of the network or the plackards on the walls of the cyber lab, which list various networks and credentials. When entering credentials, make sure they are correct, as significant issues can occur if they are not correct.

After setting up ssh, do not enable raspberry connect and then proceed to burn the uSD card. These steps are shown in Figure 2. Then click on Write. Writing to the image will take roughly 5-10 minutes or so, depending upon your USB speed and laptop.

[5] Note that this somewhat goes against normal security practice, but in the class, there may be times when we will need to share a password to access devices.

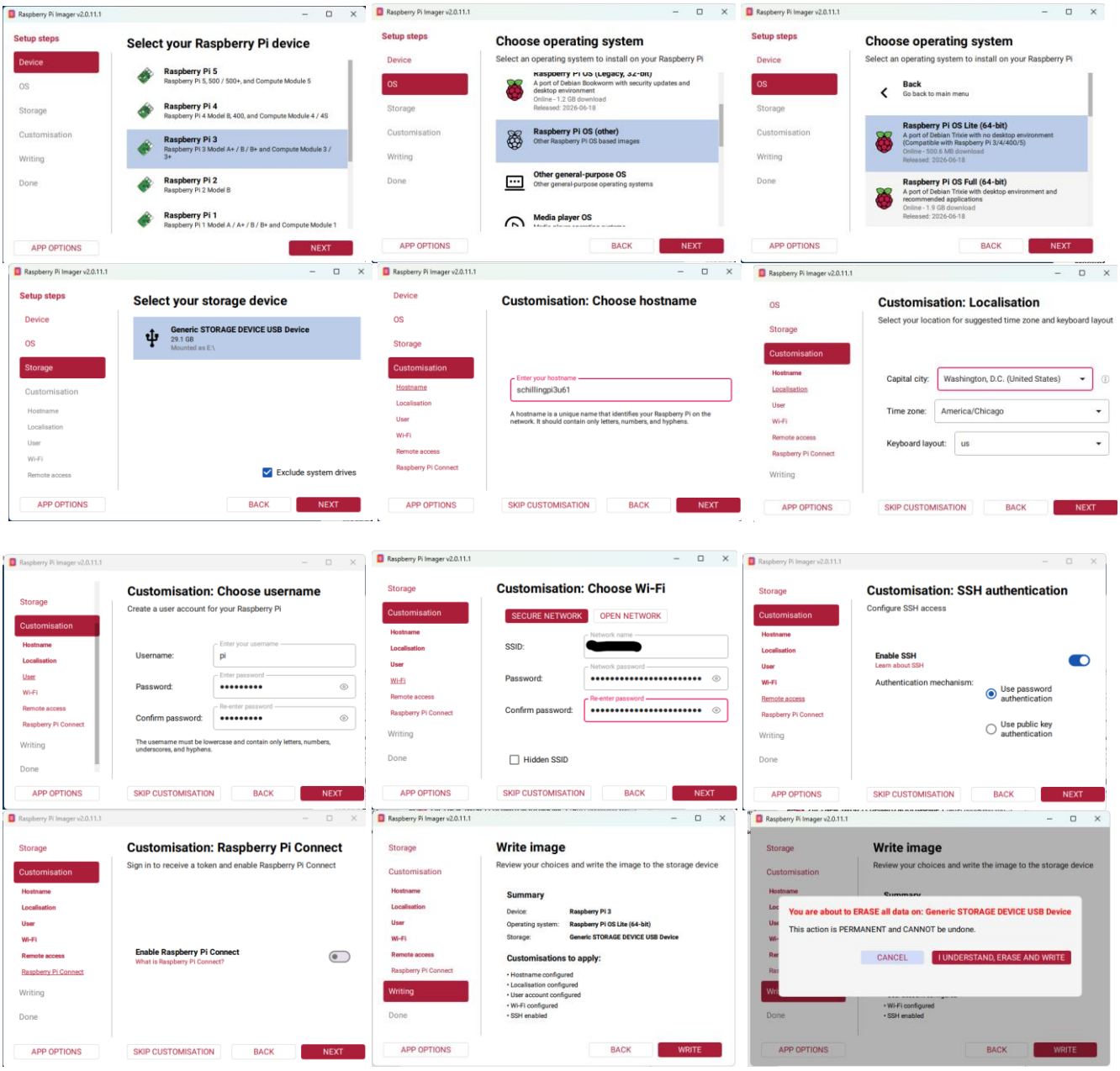



**Figure 2 Burning the Raspberry Pi image with the Raspberry Pi Imager. Note the customizations being applied to the uSD as it is being burned. You can use any password you desire. However, do not make it tremendously difficult, as there may be cases in lab where you will want to publicly disclose your Raspberry Pi's password to allow your professor to help you in lab by remoting in. While security is important, this is a lower risk environment.**

# 10 Working with the Pi

Right now, both the pi and your virtual machine are shutdown. We need to change that.

To start, plug in plug in your Trendnet adapter and connect to the network using the wired adapter. When completed, setup your virtual machine to use a Bridged connection to the network, connecting through the USB based Trendnet device. This is shown in Figure 3. (Note: Your laptop may use a different name as you may have different network capabilities on your machine.)

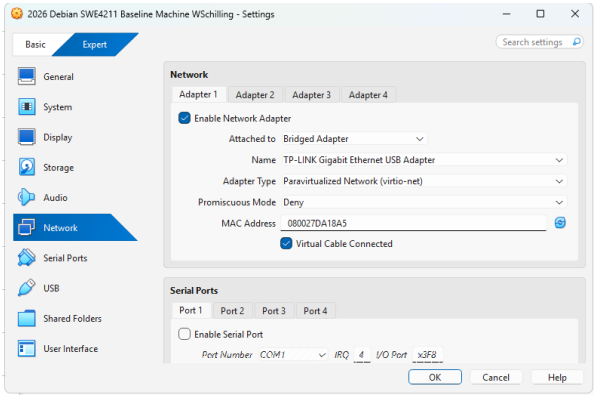



**Figure 3 Virtual machine settings. Note the name of the adapter may be different for you depending on your hardware.**

Next, boot the virtual machine up and log in, making sure you can get an IP address from the network. You can check network connectivity by running some pings.

## *10.1 Booting the Raspberry Pi and finding it on the network.*

Insert the microSD card into the Raspberry Pi. The slot is on the opposite end of the board from the Ethernet cable and the power supply adapter. Plug in the Ethernet cable to the network that your pc is attached to on the network and plug in the power supply. It will take about 30 seconds, but the Raspberry Pi will boot off the microSD card. You should be able to see the green light on the board flash as the device boots. A diagram of the board is shown in Figure 4.

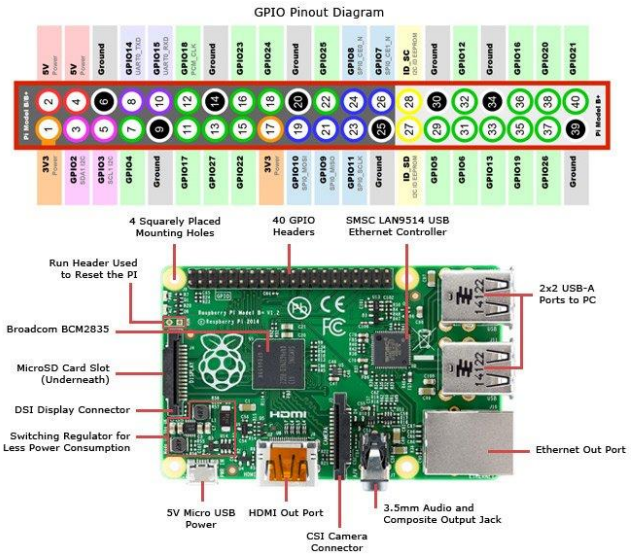



**Figure 4 Raspberry PI board. The green light next to the DSSI display connector and power supply will flash as the board boots. The red light indicates power is turned on to the board. (https://www.jameco.com/jameco/workshop/circuitnotes/raspberry-pi-circuit-note.html)**

Open up a terminal window on your virtual machine and enter the command **`ping <yourName>.local`**[6] where <yourname> is the name given to your raspberry pi.[7] This will result in your Linux virtual machine trying to ping the Raspberry Pi and returning the IP address that has been assigned by the DHCP server to the device. The screen capture shown below is an example. Notice that in this case, the router has assigned the IP address 192.168.4.155 to the Raspberry Pi board. Yours will most likely be different. You will need to remember this step, as if you have difficulty connecting to the Raspberry Pi, you will want to repeat this step again, and this will be a common command for you to enter when setting up your systems to verify connectivity. This is shown in Figure 5.

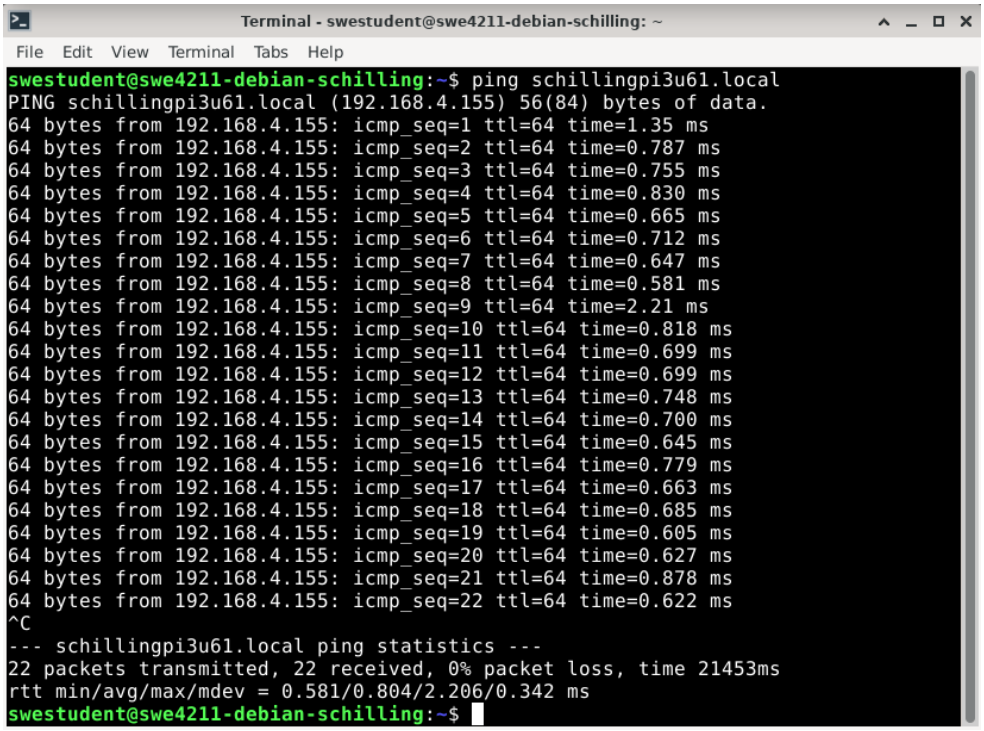



**Figure 5 Pinging your Raspberry Pi.**

[6] The .local is not always necessary, depending on how the mDNS service is setup on your router. You may be able to leave this .local off. But, you may also need it. If it works without .local, that is fine. If you need to add the .local, that is fine as well. It may change at home versus in the lab.

[7] Note: **DO NOT USE** an underscore or other symbol in the name, as it will cause problems. The name can only be characters and letters.

# 11 Verifying that C programs compile and execute on the Pi

Once the tools have been installed, open an shell on your virtual machine. Enter the commands **`cd ~`** to change into the home directory for your username, **`mkdir temp`** to create a temp directory and **`cd temp`** to change into that directory. Then issue the command **`touch arm1.c`** to create an empty file arm1.c and **`nano arm1.c`** to open it in the nano editor.

In nano, enter the code given in Figure 6. This is a simple C program that will test out the cross compiler on your virtual machine. Unlike a traditional compiler, a cross compiler will write code for a different platform. In this case, you will be writing the code on your Linux Virtual machine, but it will be targeting the ARM architecture on the Raspberry Pi.

```
#define _GNU_SOURCE
#include <stdio.h>
#include<sys/utsname.h>
int main(int argc, char* argv[])
{
  struct utsname uname_pointer;
  uname(&uname_pointer);
  printf("Cross Compilation is fun!\n");
  printf("System name - %s\n", uname_pointer.sysname);
  printf("Nodename    - %s\n", uname_pointer.nodename);
  printf("Release     - %s\n", uname_pointer.release);
  printf("Version     - %s\n", uname_pointer.version);
  printf("Machine     - %s\n", uname_pointer.machine);
  printf("Domain name - %s\n", uname_pointer.domainname);
}
```

Figure 8: Sample C (arm1.c) program for cross compilation. Note that when coping and pasting, the quotation marks may cause compilation problems. To fix, simply overwrite the “ with another “.

**Figure 6 Sample C (arm1.c) program for cross compilation. Note that when coping and pasting, the quotation marks may cause compilation problems. To fix, simply overwrite the “ with another “.**

Once you have entered the program, (note: be careful of “, as copying and pasting them may yield a slightly wrong character that causes compilation errors) type the command

**`aarch64-linux-gnu-gcc -o test1 arm1.c`**

to compile the code. You should see that the code compiles successfully. Try and execute the program on your Virtual machine by issuing the command

**`./test1`**

Does the program run? It should not. But, do you understand why it does not run on the VM? Using the console and ls command, determine the size of the compiled binary. Screen shot these for the report.

Open up a second terminal window on your virtual machine and change to the directory where the code you are developing resides. Issue the command

**`sftp pi@<your Raspberry Pi>.local`**

to connect to your Raspberry Pi. The password is what you set. Type the command

```
put test1
```

to transfer the program to the Raspberry Pi. Now open a third terminal window and change into the directory where your code was developed. Issue the command

```
ssh pi@<your Raspberry Pi>.local
```

to open a secured connection with the Raspberry Pi. Log on using the your password. Issue the command

```
uname -a
```

on your PI. Next type the command

```
./test1
```

to run the program. With this program successfully running, we now know that C programs can be cross compiled to our embedded board. Screen capture this for your report, showing the program and the uname command.

Next, return to the original terminal and issue the command

```
aarch64-linux-gnu-gcc  -static -o test2 arm1.c
```

Again, list the size of the file. Is it different? If so, is it larger or smaller? What caused this change? Save this info for your report.

Upload the file to the pi and verify that the behavior is the same by running the program with the command **./test2**.

# 12 Cross Compiling with Eclipse[8]

In the virtual machine, start eclipse under Linux using the menu (Development -> Eclipse CDT) or from the shell by issuing the command **`eclipse &`**. Select a workspace that you would like to use in the Linux filespace (aka not the shared file space). Then click on Launch. In Eclipse, select "File", "New", "C/C++ Project", and "Classic C Project" before clicking next. Enter a project name of "Lab1Part3" and then select Cross GCC. Click on next until you get to the C Project screen. Enter the author and a greeting. Click on next and then Next again until you get to the Cross GCC Command. Enter the prefix for the compiler and the path, as is shown in Figure 7.

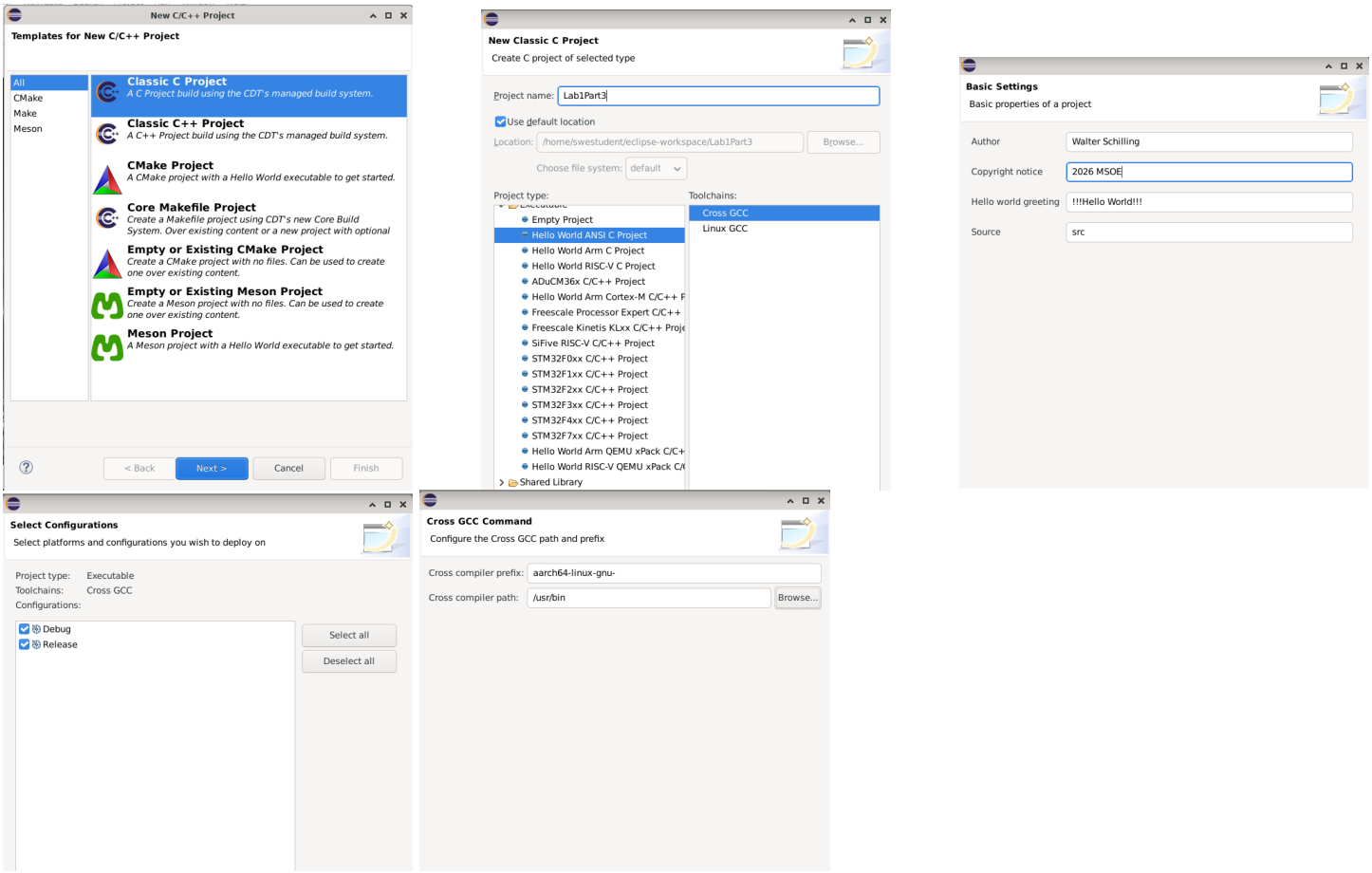



**Figure 7 Setting up our first Eclipse Project.**

Modify the code so that it matches that which is shown below in Figure 11.

[8] The following instructions are based upon the material in Molloy, Derek. 2016. Exploring Raspberry Pi: interfacing to the real world with embedded Linux. starting on page 283

```
/*
 ============================================================================
 Name        : Lab1Part3.c
 Author      : Walter Schilling
 Version     :
 Copyright   : 2026 MSOE
 Description : This program tests address locations.
 ============================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int bss_var;             /* This is an uninitialized global variable. */
int data_var = 1;/* This is an initialized global variable. */
static int  lv;
static int lv2 = 1;

int main(int argc, char* argv[]) {
           void *stack_var; /* This is a local variable declared on the stack.*/

           char hostname[1024];  // This is an array declared on the stack which will hold the hostname.

           stack_var = (void*) main;
           printf("Executing Program: %s\n", argv[0]);

           puts("!!!Hello World!!!\n");
           printf("Hello World!  Main is executing at address %p\n", stack_var);
           printf("This address (%p) is in our stack frame\n", &stack_var);

           printf("The printf function is located at location %p.\n", printf);

           /* The bss section contains uninitialzed data. */
           printf("This address (%p) is our bss section\n", &bss_var);

           /* Data section contains initialized data */
           printf("This address (%p) is in our data section\n", &data_var);

           /* This is a static variable which is uninitialized. */
           printf("The address of the static variable which is uninitialized is %p.\n",  &lv);

           /* This is a static variable which is initialized. */
           printf("The address of an initialized lv variable is %p\n", &lv2);

           /* read the hostname. */
           gethostname(hostname, 1024);

           /* Print out the size of a pointer.*/
           printf("The size of a pointer is %ld.\n", sizeof(stack_var));

           /* Print out the hostname to the console.*/
           printf("Hostname: %s\n", hostname);
           return 0;
}
```

**Figure 8 Our first program (modified from Embedded Linux Primer: Second Edition)**

Right click on the project, properties, C++ Build, Settings, Cross GCC Linker, General, and check the static option, as is shown in Figure 9. (You may need to expand the project from the left hand segment or by clicking window->show view -> C/C++ Projects.) When finished click on Apply and Close. Build the code by clicking Project -> Build All from the menu. It should build cleanly with no errors or warnings.

Open up a terminal window and cd to the directory where your project lives. You should see two folders, a Debug and a src folder. The Debug folder is where the binary files live at. The src folder is where the source code lives at. Change into the Debug directory. Determine how large this file is (i.e. `ls -l` ). Transfer the binary to the Raspberry pi using sftp and then open up an ssh session. Execute the program on the raspberry Pi, taking note of the various addresses for main, the stack, the bss section, and the data section. Screen capture this for your report.

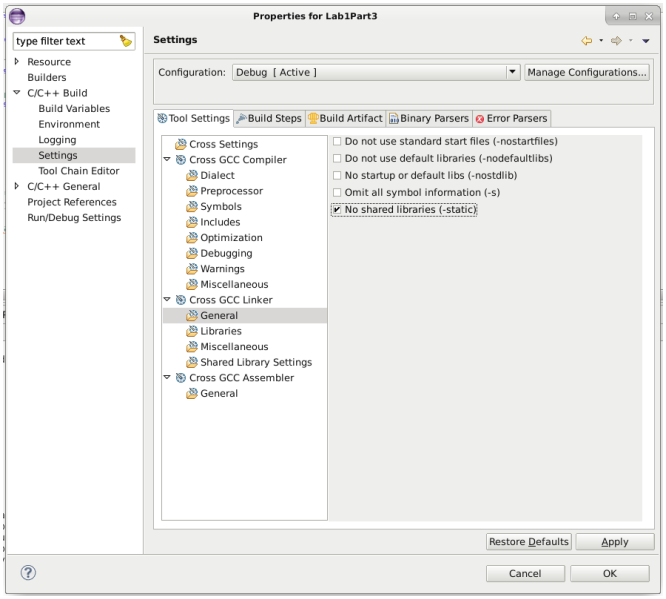



**Figure 9 Compiler Settings**

In the console on the Virtual Machine, change to the location where your project lives and into the src directory. This is most likely something like /media/sf_lwshare/workspace/Lab1Part3. On the console, issue the simple command **`gcc Lab1Part3.c -static -o host`**. Determine how large this file is (**`ls -l`**). Execute the same program on the host. What is significantly different, if anything? Why might the addresses be different between the virtual machine host and the Raspberry Pi? Why might they still be similar though not identical? What does the size of a pointer being 8 mean? Screen capture this for your report.

# 13 Remote Debugging

One of the things that is important to be able to do is to run remote debug sessions on your raspberry pi. While you have used the debugger in other classes, remote debugging is a bit more complex. You are running commands on your local machine which then need to be communicated to the raspberry pi for processing where the actual application is executing. To perform remote debugging, we need to be able to setup our environment for remote debugging. We will need to add a library on the raspberry pi and then we will need to setup our Eclipse project accordingly.

## 13.1 Installing the tools

To begin, open up an ssh connection to the Raspberry Pi using ssh. In the shell, issue the command:

**sudo apt-get install gdbserver**.

This command will install the gdbserver service on the Raspberry Pi.

> *"gdbserver is a control program for Unix-like systems, which allows you to connect your program with a remote GDB via target remote---but without linking in the usual debugging stub. gdbserver is not a complete replacement for the debugging stubs, because it requires essentially the same operating-system facilities that GDB itself does. In fact, a system that can run gdbserver to connect to a remote GDB could also run GDB locally! gdbserver is sometimes useful nevertheless, because it is a much smaller program than GDB itself. It is also easier to port than all of GDB, so you may be able to get started more quickly on a new system by using gdbserver. Finally, if you develop code for real-time systems, you may find that the tradeoffs involved in real-time operation make it more convenient to do as much development work as possible on another system, for example by cross-compiling. You can use gdbserver to make a similar choice for debugging."*[9]

This service allows one to remotely debug the raspberry pi. This will only need to be setup once and it will be there for all future installations. Note that you will need to remember this should you need to reimage your USD card.

Next, we need to make sure our host environment, namely the Debian Linux VM, is setup to allow gdb multiarch. GDB is the GNU debugger, the tool used for debugging source code under Linux. Multiarch is a specific variant of gdb that allows us to debug on different architectures than the host machine. In our case, we will want to be able to debug ARM Linux code under the Intel x86 host environment.

To do this, open up a terminal window on the VM and issue the command

```
sudo apt-get install gdb-multiarch
```

## 13.2 Why We Use an SSH Tunnel

Eclipse offers a "C/C++ Remote Application" launch type that will, in principle, log into the target over SSH, copy your program across, and start gdbserver for you, all automatically. It is convenient when it works. On current Raspberry Pi OS images, it does not work, and it is worth understanding why before we work around it. While the technical specifics are not glamorous, the situation shows the importance of configuration management and an understanding of the role of cybersecurity in modern development.

Eclipse ships its own SSH client library (JSch) rather than using the operating system's ssh command. The version bundled with Eclipse has not been updated in several years, and the only key exchange algorithms it can offer are based on SHA-1,

[9] https://ftp.gnu.org/old-gnu/Manuals/gdb-5.1.1/html_node/gdb_130.html

a hash function that has been considered broken since 2017. The Raspberry Pi is running OpenSSH 10.0, which has removed those algorithms entirely. Its key exchange list contains only modern constructions such as ML-KEM, Curve25519, and ECDH.

When two SSH implementations connect, they must agree on a common set of algorithms. Here the client's list and the server's list have nothing in common at all, so the connection is refused before authentication is even attempted. Eclipse reports this as is shown in Figure 10.

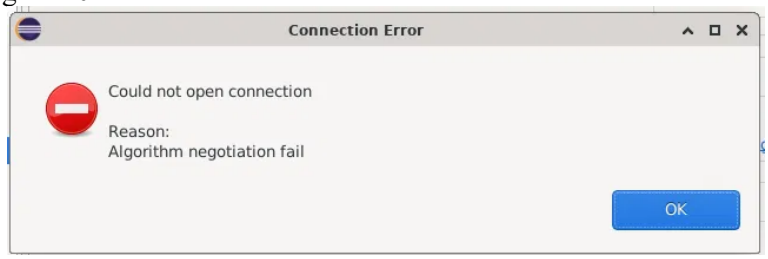



**Figure 10 Eclipse Error Message when remote connecting.**

You could "fix" this by re-enabling the deprecated algorithms on the Raspberry Pi. **Do not do this.** Weakening a server's cryptographic configuration to accommodate an out-of-date client is exactly the kind of expedient decision that produces long-lived vulnerabilities, and it would need to be repeated every time the Pi's SSH server is updated.

Instead, we will take Eclipse's SSH client out of the picture entirely. We will use the operating system's own ssh command which is current and negotiates modern cryptography without difficulty to build an encrypted tunnel between the VM and the Pi. Eclipse will then talk to one end of that tunnel using nothing but the GDB remote protocol. It will never attempt an SSH connection of its own. This arrangement has a second advantage: every piece of the mechanism is visible to you. You will start gdbserver yourself, open the tunnel yourself, and point the debugger at it yourself. When something breaks, you will know which of the three parts to examine. This behavior is shown graphically in Figure 11.

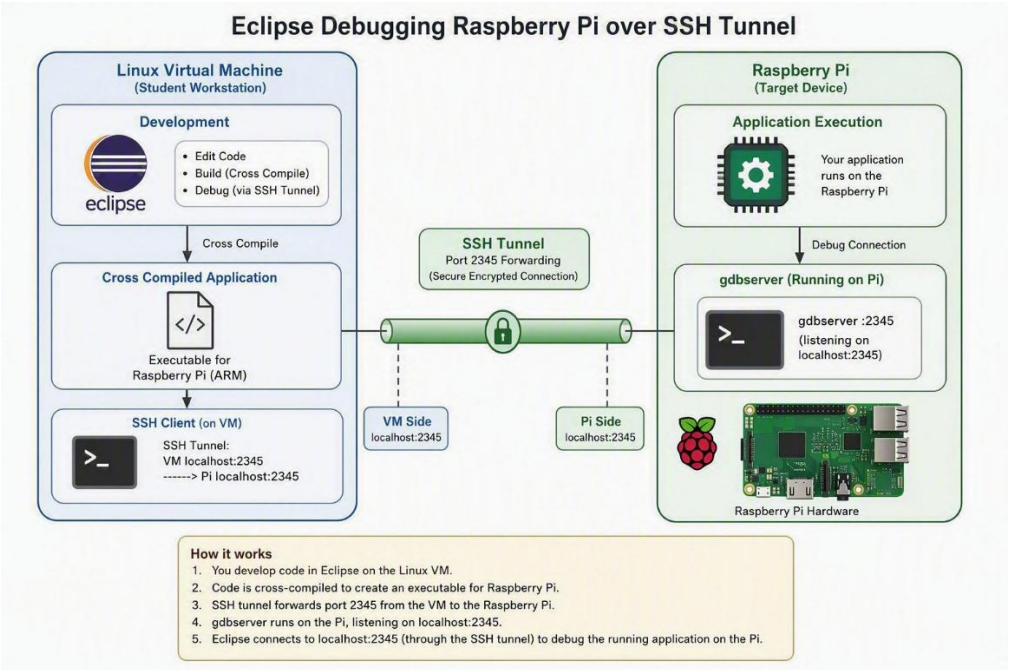



**Figure 11 Remote debugging on the Raspberry Pi using an ssh tunnel. (Generated with CoPilot)**

## 13.3 Building an Executable Suitable for Debugging

A debugger needs debug symbols. As you talked about in your C/C++ course and operating systems courses, this is the mapping between machine instructions and your source code. It also needs the compiler not to have rearranged your code beyond recognition, which can occur with optimization.

In Eclipse, open Project → Properties → C/C++ Build → Settings, and under your cross compiler's Debugging and Optimization categories confirm that the Debug Level is set to the maximum and the optimization level is set to 0. This is shown in Figure 12.

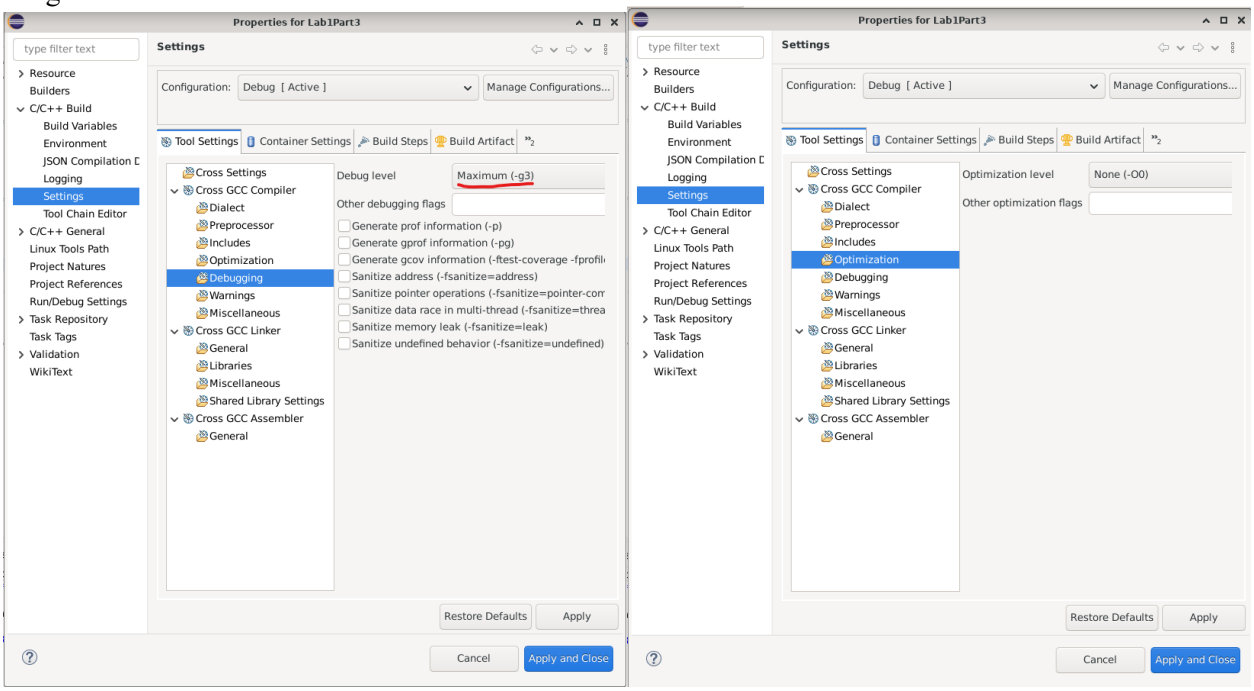



**Figure 12 Setting debugging and optimization levels in the tool.**

The Debug build configuration normally sets these already; the Release configuration deliberately does not. Make sure the active configuration is Debug before you build. If you attempt to debug an optimized binary, you will see execution jump backwards through your source, variables reported as "optimized out," and breakpoints that refuse to bind.

Build the project and note the path to the resulting executable on the VM, which can be found on the resource menu of the project properties.. It will be something like:
***/home/student/workspace/MyProject/Debug/MyProject***

You will need this path twice: once to copy the file to the Pi, and once to tell Eclipse where to find the symbols.

## 13.4 Copying the Executable to the Raspberry Pi

The Pi and the VM each need a copy of the executable, and the two copies must be byte-for-byte identical. GDB reads the symbol table from the local copy while gdbserver executes the remote copy. If they disagree, breakpoints will land at the wrong addresses and the session will behave nonsensically. Any time you rebuild, you must copy the executable image out to the raspberry pi.

From a bash terminal on the VM, enter the following command, noting that the file path and the raspberry pi name will be different.:

```
scp ~/workspace/MyProject/Debug/Lab1Part3 pi@schillingpi3u61.local:~/
```

Substitute your own project path and your own Pi's hostname throughout these instructions. There are ways that this can be automated, as we will see in the future.

## 13.5 Starting gdbserver on the Raspberry Pi

Open an SSH session to the Pi:

```
ssh pi@schillingpi3u61.local
```

Start gdbserver, telling it to listen on port 2345 and to run your program:

```
gdbserver localhost:2345 ./Lab1Part3
```

You should see something similar to that which is shown in

```
pi@schillingpi3u61:~ $ gdbserver localhost:2345 ./Lab1Part3
Process ./Lab1Part3 created; pid = 3681
Listening on port 2345
```

**Figure 13 The Raspberry Pi waiting to execute the program.**

Note carefully that we wrote **localhost:2345** and not **:2345**. The **localhost** prefix instructs gdbserver to accept connections only from the Pi itself. This matters, as the GDB remote protocol has no authentication and no encryption whatsoever, and anyone who can reach an open gdbserver port can execute arbitrary code on that machine as the user who started it. Binding to loopback means the only way in is through the SSH tunnel we are about to create, which does provide both authentication and encryption.

Leave this terminal open. gdbserver has started your program but has not run it. The process is suspended at its entry point, waiting for a debugger to connect and tell it to proceed. Keep this window handy. This is where program console output will go.

## 13.6 Opening the SSH Tunnel

In a new terminal window on the VM, not on the Pi, issue the following command

```
ssh -L 2345:localhost:2345 pi@schillingpi3u61.local
```

The **-L** option requests local port forwarding, and its argument reads left to right as: listen on port 2345 of this machine; forward anything that arrives there, through the encrypted SSH connection, to port 2345 of localhost as seen from the far end. Because "the far end" is the Pi, localhost in that middle position means the Pi itself. This is exactly where the gdbserver is listening. The practical effect is that port 2345 on your VM now behaves as though it were port 2345 on the Pi. Anything connecting to it is transparently carried across the network inside the SSH session.

You will be dropped at a shell prompt on the Pi. The tunnel exists only while this session is open. Leave the window alone for the duration of your debugging session; closing it or letting it time out will drop the tunnel and disconnect the debugger.

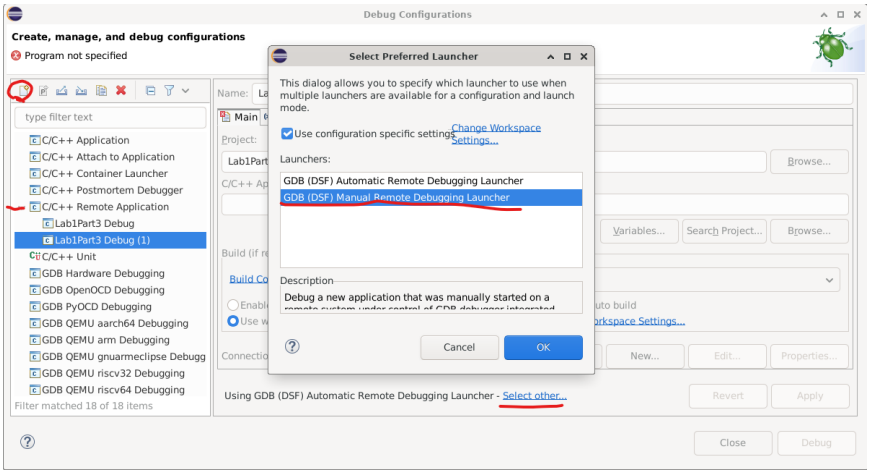
You should now have three things running simultaneously:

- A terminal on the Pi running gdbserver, waiting for a connection.
- A terminal on the VM holding the SSH tunnel open.
- Eclipse, which we are about to configure.

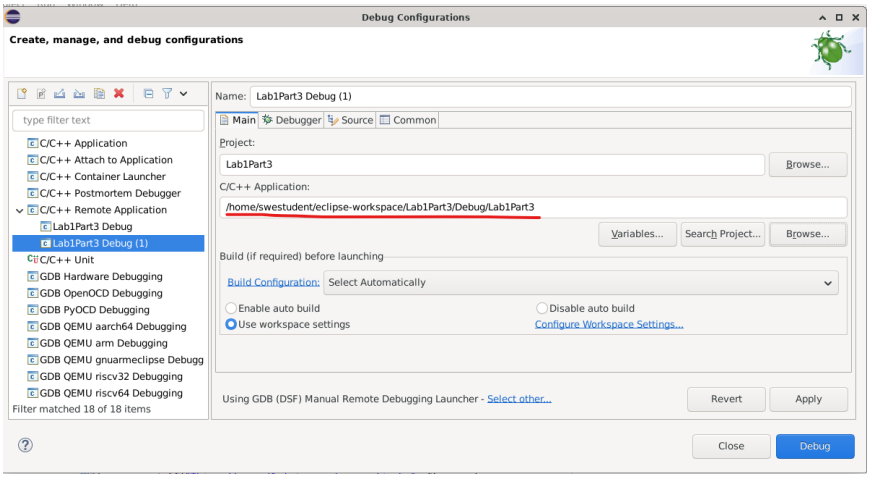
## 13.7 Configuring the Eclipse Debug Configuration

In Eclipse, select Run → Debug Configurations…. In the list on the left, select C/C++ Remote Application and click the New Configuration button to create a configuration for your project. You will start by selecting the correct launcher. This is the step that keeps Eclipse's SSH client out of the way, and it is easy to overlook.

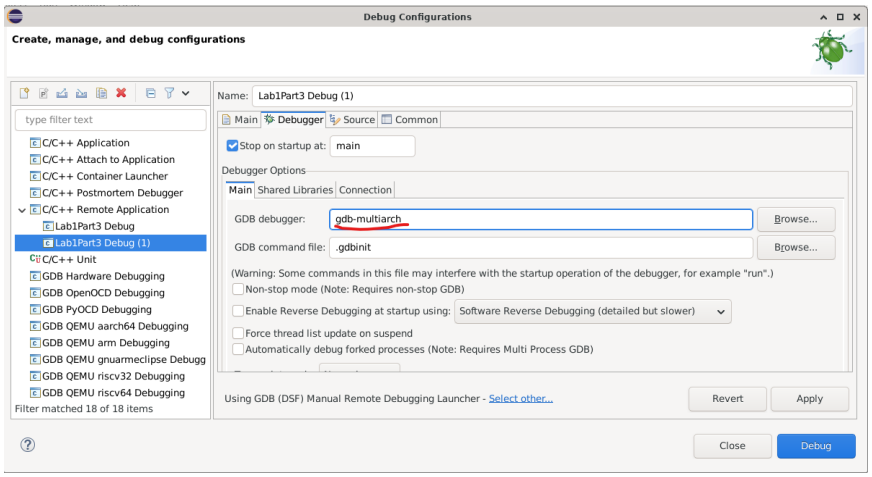
At the bottom of the Main tab is a line reading "Using GDB (DSF) Automatic Remote Debugging Launcher" followed by a Select other… link. Click that link. In the dialog that appears, check Use configuration specific settings and then select GDB (DSF) Manual Remote Debugging Launcher from the list. Then click OK. The word Manual is what matters. The Automatic launcher is the one that tries to SSH into the target on your behalf and fails with the negotiation error. The Manual launcher assumes you have already started gdbserver yourself and simply opens a TCP connection to it.




You will notice that the Connection and remote-path fields disappear from the Main tab once you switch launchers. That is expected. Those fields belonged to the SSH-based workflow we are no longer using. In the main tab, set the C/C++ application to be the executable we will be running. This is the copy on the VM, not the copy on the Pi. Eclipse reads debug symbols from this file; the Pi executes its own copy.

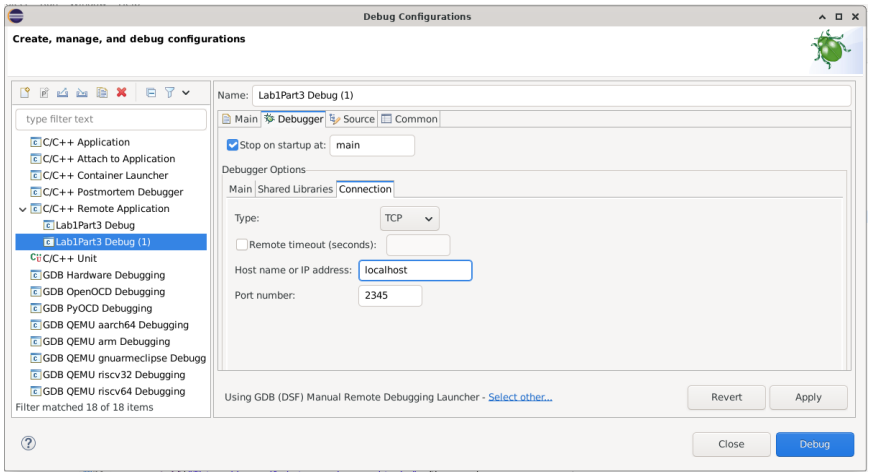



Next, setup the debugger on the debugger tab. Set the gdb debugger to be gdb-multiarch and leave the command file at its default.




This is where the package you installed earlier comes into play. The default gdb on your x86-64 VM understands only x86-64 instructions and will reject an ARM binary. gdb-multiarch is compiled with support for many architectures and will detect AArch64 from your executable automatically.

Next, click on the connection subtab. Set the type to be tcp, enter the localhost for your Raspberry Pi, and enter the port number 2345.




localhost is correct here and is not a mistake. Eclipse connects to the local end of the tunnel on the VM; the tunnel carries the connection to the Pi. Eclipse itself has no idea the Pi exists.

When finished, click Apply, then Debug. Eclipse will switch to the Debug perspective. In the terminal running gdbserver on the Pi, a line will appear reporting a remote debugging connection from 127.0.0.1. Execution will halt at main(), and the Debug view will show your program's stack.

```
pi@schillingpi3u61:~ $ gdbserver localhost:2345 ./Lab1Part3
Process ./Lab1Part3 created; pid = 3681
Listening on port 2345
Remote debugging from host ::1, port 47120
```

From this point forward, debugging proceeds exactly as it does for a local program. To set breakpoints, simply double click on the line of code where you want the breakpoint to be at. Step over will step over a line of code. Step into will step into

the method invoked on that line of code. Step with F5 and F6, resume with F8, and inspect variables in the Variables view. The only difference is where the instructions are physically executing.

Note that your program's output does not appear in the Eclipse Console. Standard output and standard error belong to the process on the Pi, so they appear in the terminal where you started gdbserver. Keep that window visible.

By default, gdbserver exits when the debugging session ends, as is shown in Figure 14. Before starting a second session you must restart it on the Pi:
**gdbserver localhost:2345 ./Lab1Part3**

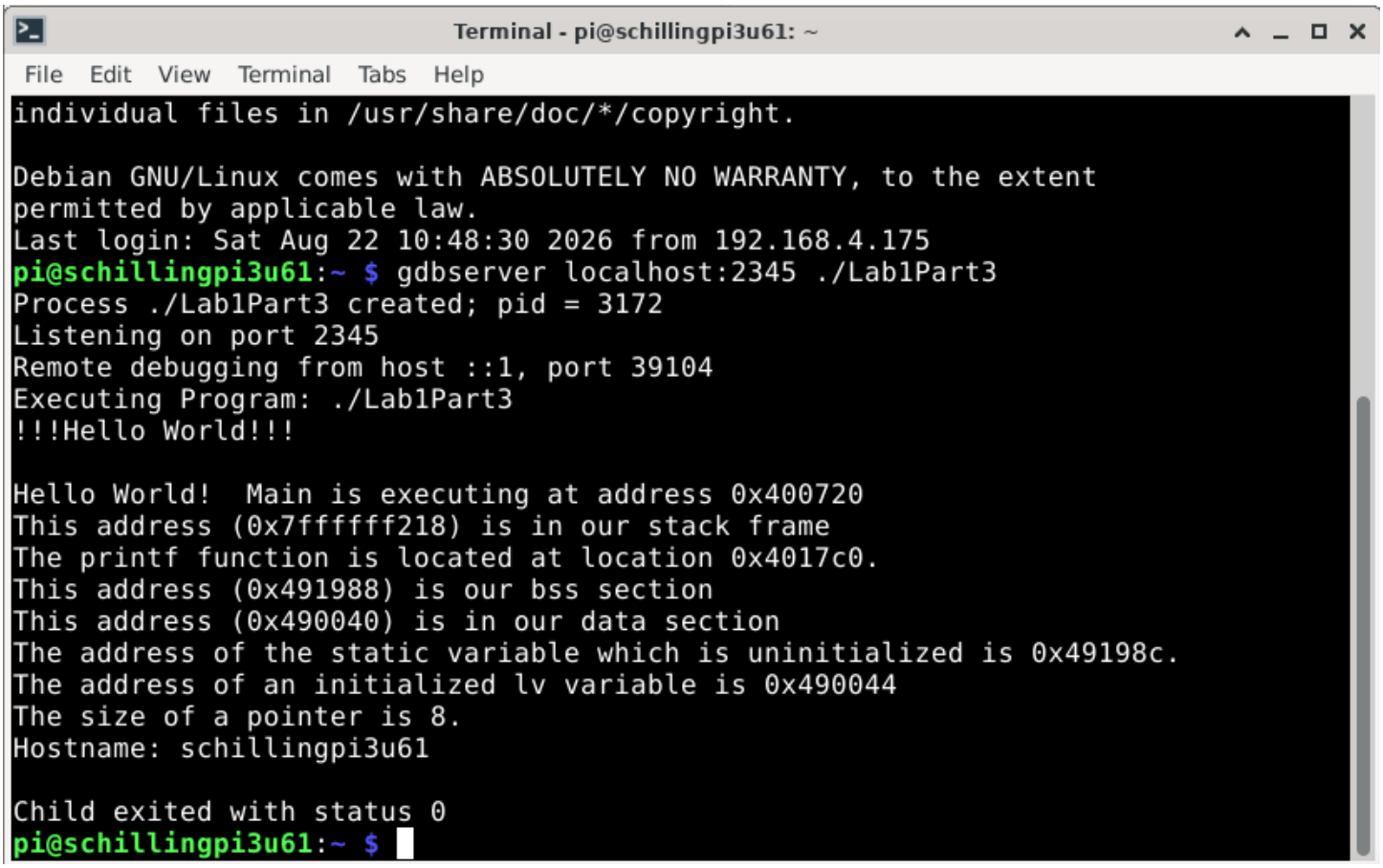



**Figure 14 The debug session ended.**

At this point, you have now completed lab 1, and you should be able to remotely execute code on your Raspberry Pi. In future labs, we will streamline this a bit, but you now have the fundamentals down.