Software Development Practice 1

Instructor: RSP <rawat.s@eng.kmutnb.ac.th>

Exploring Raspberry Pi SBC and OS

Objectives

- Learn how to flash a Raspberry Pi OS image file to a microSD card.
- Learn how to boot and configure the Raspberry Pi (RPi) using the headless configuration method (without a keyboard, mouse, or HDMI display attached).
- Learn how to share Internet connectivity from a Windows or Ubuntu machine to the RPi via LAN.
- Learn how to remotely access the RPi via SSH.
- Learn how to install the C/C++ Extension Pack and configure the VS Code IDE to build and debug C program on the RPi.
- Learn how to write Python script to scan nearby BLE devices.

Expected Learning Outcomes

- Students will be able to correctly install and set up the Raspberry Pi OS.
- Students will be able to access and operate the RPi remotely using SSH.
- Students will be able to build, run, and debug C programs on the RPi using the VS Code IDE via Remote SSH.

Task 1: Boot the RPi with Raspberry Pi OS from a microSD

1. Download and install **the RPi Imager** (for Windows or Linux).
   - For Windows: https://downloads.raspberrypi.org/imager/imager_latest.exe
2. Download the Raspberry Pi OS image file.
   - URL: https://www.raspberrypi.com/software/operating-systems/
   - Download Options:
     - Raspberry Pi OS with desktop vs. Raspberry Pi OS Lite
     - 32-bit vs. 64-bit ARM architecture
   - Choose **Raspberry Pi OS Lite (64-bit)**.
     - File: `2025-05-13-raspios-bookworm-arm64-lite.img.xz`

3. Use the **Raspberry Pi Imager** on the Window machine to write (or flash) the .img.xz file to a microSD (storage capacity of 8GB or more).

- You need a microSD card writer (external, via USB port) for this task.
- In order to configure and flash the image file properly, use the following settings:
  - Choose Raspberry Pi Device: Raspberry Pi 4B
  - Choose OS: Use Custom OS (select the .img.xz file downloaded previously)
  - Choose Storage: Use the microSD device
- Specify a **unique hostname**. In the example, we use 'rpi4b-demo'.
- Make sure that the **SSH server is enabled**.
- Use **default user and password** (pi : raspberry)
- Use the 'passwd' command to change the password of the 'pi' user.

4. Share the Internet connectivity from your computer to the RPi.

- Assume that your computer is connected to a WiFi router with Internet access.
- You want to share the Internet with the RPi via the Ethernet/LAN port.
- Configure the host computer (e.g. windows 10/11 or Ubuntu) to share Internet connectivity via LAN.
- Connect the RPi and your computer directly using a network cable.

Note:

- If your computer doesn't have a LAN / RJ45 port, you can use a USB-to-LAN adapter.
- **Alternative method**: To avoid using a network cable, you can configure the Wi-Fi settings using the Raspberry Pi Imager before flashing the OS to the microSD card. In this case, you can use your smartphone to act as a Wi-Fi hotspot to share Internet connectivity to both RPi and your computer.

5. Boot the RPi from the microSD card with the installed Raspberry Pi OS.

- Use a 5V Adapter with a USB Type-C connector to provide DC power supply to the RPi. The red LED on the board is turned on, indicating the SBC is powered on.

Note:

- Some RPi boards available for the hands-on lab are equipped with a metal heatsink and a 5VDC-powered small cooling fan, while others only have have a metal heatsink.

6. Ping the RPi and connect your RPi using a SSH client. For example:

```
ping -4 rpi4b-demo.local
ssh pi@rpi4b-demo.local
```

7. After logging in via SSH, do the following tasks:

- Check the IP addresses for the network interfaces: `eth0` and `wlan0` using the '`ifconfig`' command.
- Check the Internet connectivity using the '`ping 8.8.8.8`' command (used to access one of the Google public DNS servers).
- Use the '`sudo raspi-config`' command to configure RPi system settings.
    - Use Tab and Arrow key to select the menu options.

Note: For network configuration on the RPi, you can use the command: '`sudo nmtui`'

8. Access your RPi using certificate-based authentication (instead of password-based authentication)

- Find out how to setup **SSH key authentication** from Windows PowerShell and copy the public key file to RPi.
- Transfer an exisitng file (named file.txt as an example) to or from the RPi.
    - For example: `scp file.txt pi@rpi4b-demo.local:/home/pi/`
- Execute a Linux command on the RPi.
    - For example: `ssh pi@rpi4b-demo.local "ls -l /home/pi"`

Questions

1. What information do the outputs of the following commands show on the RPi?

```
$ vcgencmd measure_temp | cut -f2 -d=

$ vcgencmd measure_clock arm | awk -F"=" '{printf ("%0.0f", $2 / 1000000); }'

$ awk '{printf ("%0.0fHz\n",$1/1000); }' \

  </sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq

$ vcgencmd measure_volts | cut -f2 -d= | sed 's/000//'

$ for id in core sdram_c sdram_i sdram_p ; do \

  echo -e "$id:\t$(vcgencmd measure_volts $id)" ; done
```

2. Which Linux commands can be used to check whether the SSH server is running on the RPi?

Task 2: SSH and Remote Development using VS Code IDE

1. Open **VS Code IDE** on your computer (e.g. Windows, Ubuntu platforms) and install the **Remote Development Pack for VS Code**.

2. Use the VS Code IDE to connect the RPi board on the same network via SSH.

   - You have to specify the user name and hostname of the RPi and the password to log in. For example: `ssh pi@rpi4b-demo.local`
   - The first time the RPi is connected from VS Code IDE remotely via SSH, the `VS Code server` will be installed automatically on the RPi.

3. Install **C/C++ Extension pack for VS Code IDE** on the RPi.

   - Create and open a folder for a new project.
   - Create the `main.c` code and write your C code.
   - Run the following commands to compile and run the C program on RPi.

     ```
     $ gcc -g main.c -o main && ./main
     ```

   - Show information about the `./main` executable file.

     ```
     $ file ./main | sed 's/, /\n/g'
     ```

   - Add the configuration files: `launch.json` and `tasks.json` under the `.vscode` directory of the project.
   - `Ctrl+Shift+B` to build the C source code and `Ctrl+Shift+D` to run and debug the executable file of the C program.

4. Use WSL2 Ubuntu 2 to cross-compile `main.c` targeting 64-bit ARM CPU.

   - Install the GCC cross-compilation toolchain for `aarch64`.

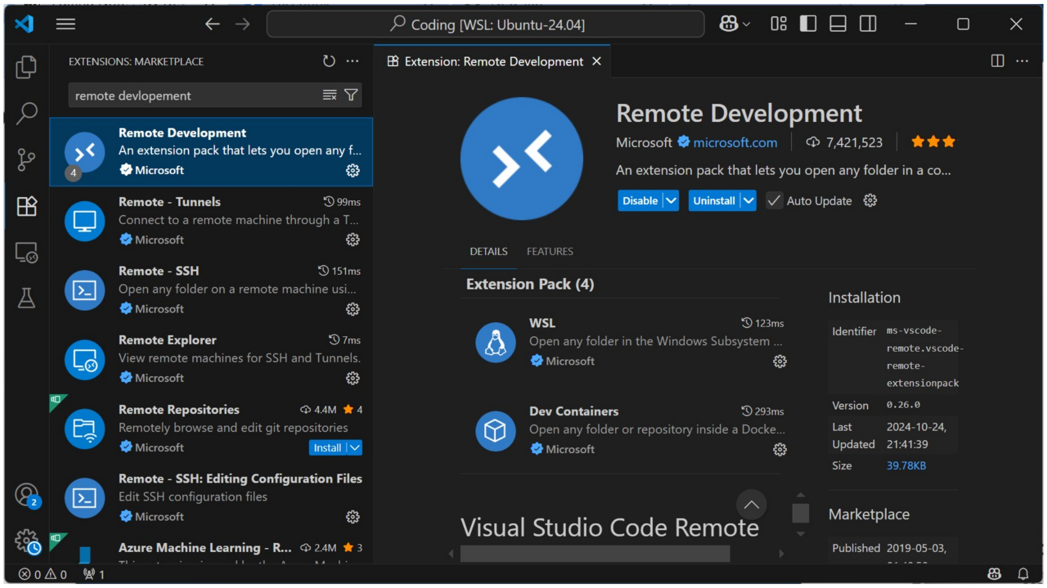
     ```
     $ sudo apt update
     $ sudo apt install -y gcc-aarch64-linux-gnu
     ```

   - Compile `main.c` for 64-bit ARM, with all used libraries statically linked.

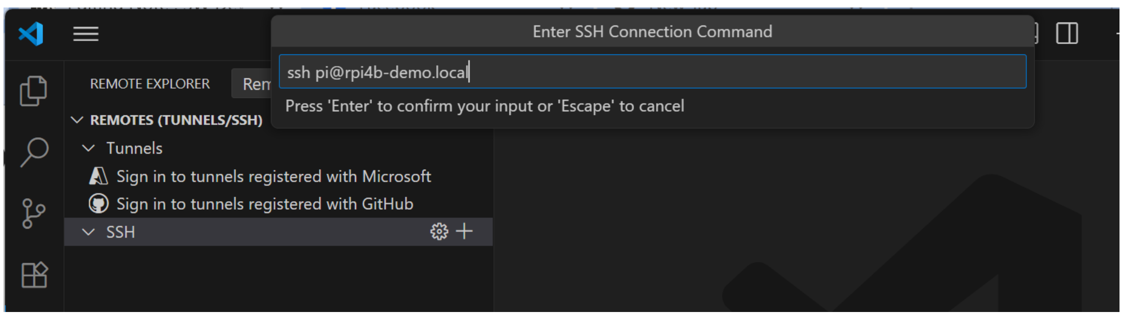
     ```
     $ aarch64-linux-gnu-gcc main.c -o main_aarch64 -Wall -static
     ```

   - Copy and run the executable file on RPi. WSL2 Ubuntu might not resolve the `.local` hostname (such as `rpi4b-demo.local`) via mDNS, so use the RPi’s IPv4 address instead.

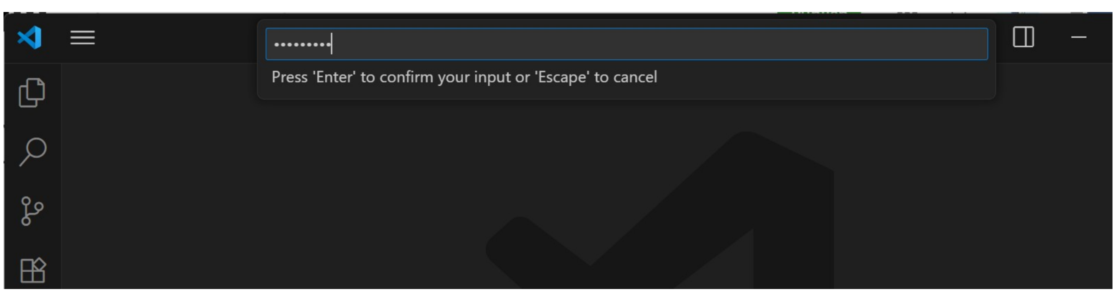
     ```
     $ scp ./main_aarch64 pi@192.168.100.60:/home/pi
     $ ssh pi@192.168.100.60 "/home/pi/main_aarch64"
     ```



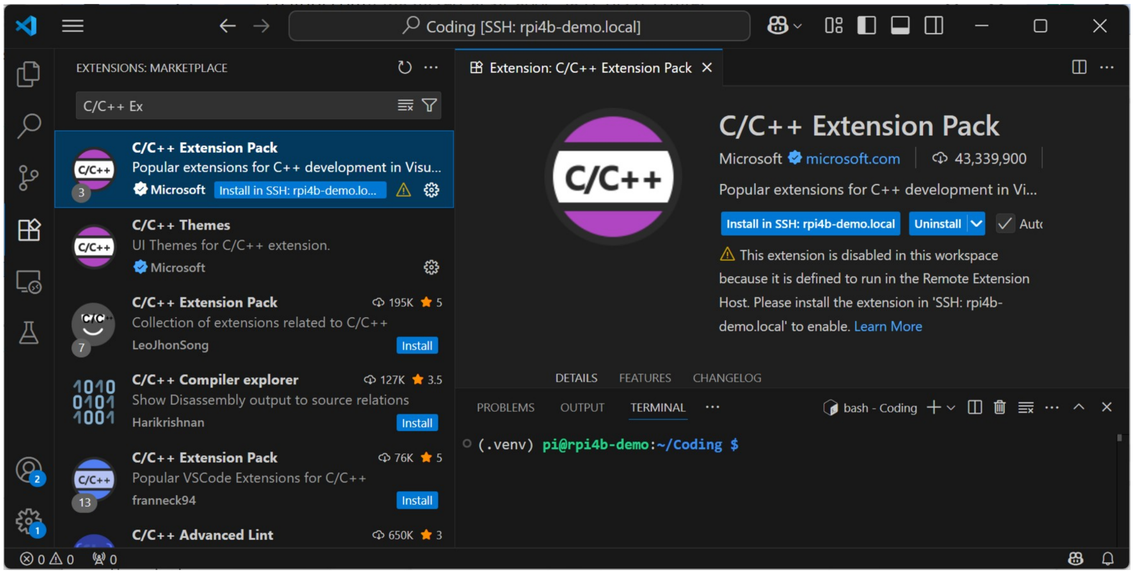
Install Remote Development Pack for VS Code IDE.



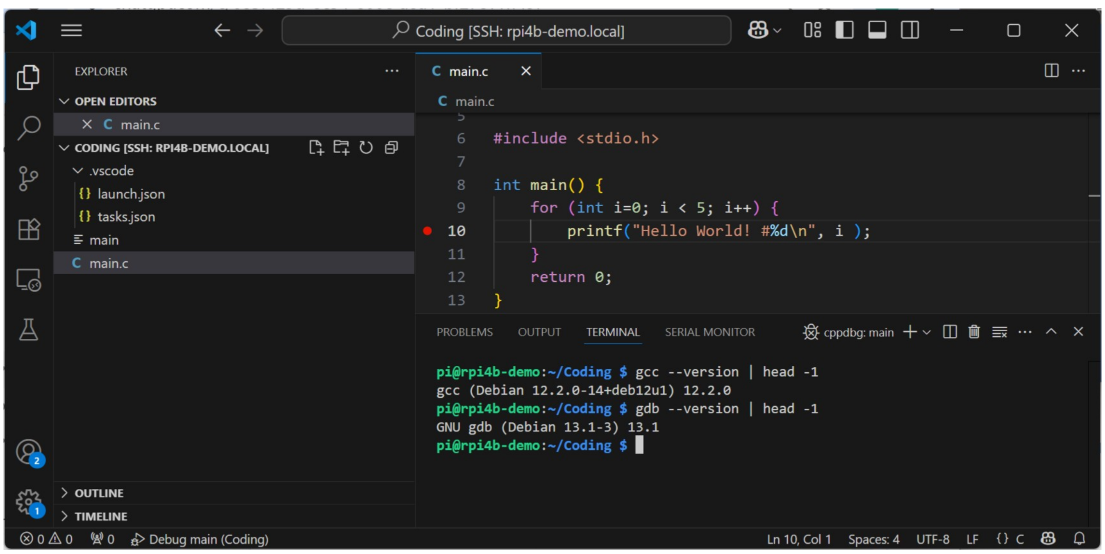
Use Remote-SSH to log in as the user pi on the RPi.



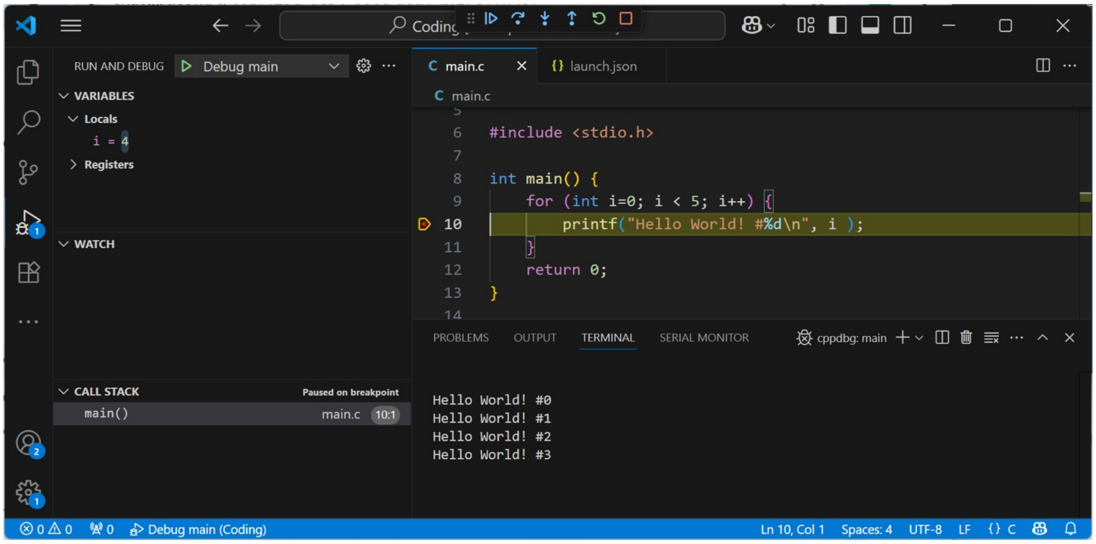
Enter the password for the user pi.



Install C/C++ Extension Pack on RPi.



Edit the C source code in the main.c file.



Run and Debug the C Program.

File: main.c

```
#include <stdio.h>

int main() {
    for (int i=0; i < 5; i++) {
        printf("Hello World! #%d\n", i );
    }
    return 0;
}
```

File: .vscode/tasks.json

```json
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Build C Program",
      "type": "shell",
      "command": "gcc",
      "args": [
        "-g",
        "-Wall",
        "main.c",
        "-o",
        "main"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "problemMatcher": ["$gcc"],
      "detail": "Build an executable file from C source code"
    }
  ]
}
```

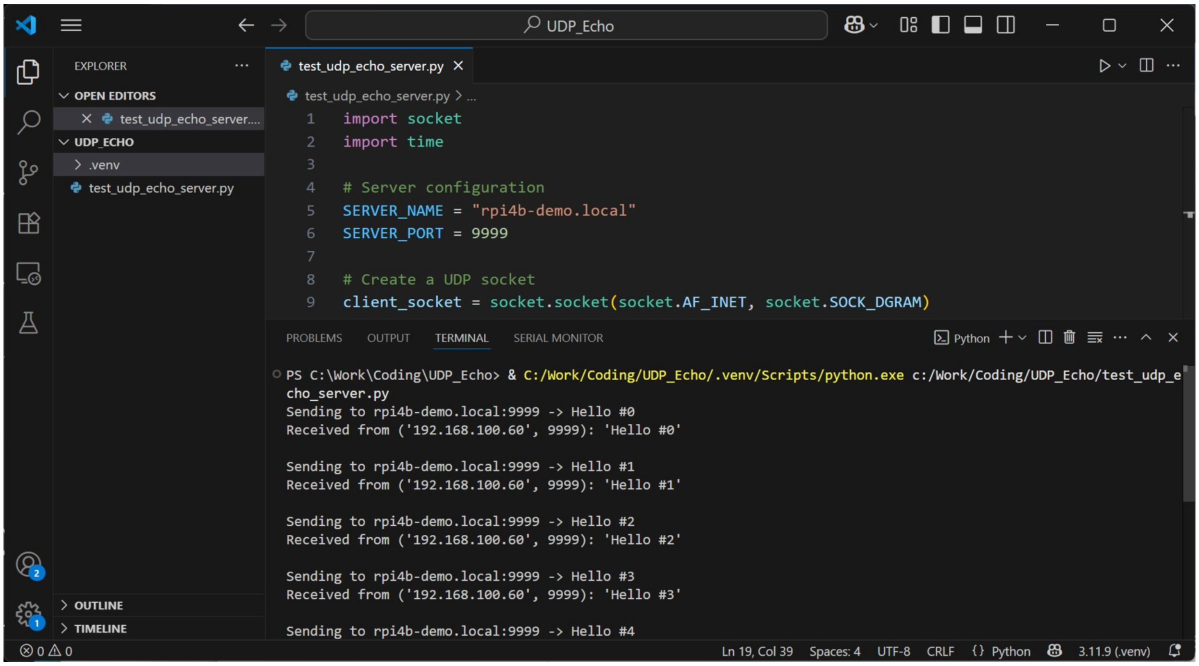
File: .vscode/launch.json

```json
{
  "version": "1.0.0",
  "configurations": [
    {
      "name": "Debug main",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/main",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "miDebuggerPath": "/usr/bin/gdb",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        }
      ]
    }
  ]
}
```

Task 3: Testing UDP Echo Server in C using Python Script

1. Compile the udp_echo_server.c file and run the executable file on the RPi.

2. Run the test_udp_echo_server.py file on the user’s computer (Windows) and observe the output messages.



File: test_udp_echo_server.py

```
import socket
import time

# Server configuration
SERVER_NAME = "rpi4b-demo.local"
SERVER_PORT = 9999

# Create a UDP socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

try:
    for i in range(10):
        message = f"Hello #{i}"
        # Send message to server
        print(f"Sending to {SERVER_NAME}:{SERVER_PORT} -> {message}")
        client_socket.sendto(message.encode(), (SERVER_NAME, SERVER_PORT))
        # Receive response from server
        data, server = client_socket.recvfrom(1024)
        print(f"Received from {server}: '{data.decode()}'\n")
        time.sleep(1.0)

except Exception as e:
    print(f"Error: {e}")

finally:
    client_socket.close()
    print('Done')
```

File: udp_echo_server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>      // for close()
#include <arpa/inet.h>   // for sockaddr_in, inet_ntoa()
#include <sys/socket.h>  // for socket(), bind(), recvfrom(), sendto()

#define PORT         9999
#define BUFFER_SIZE  1024

int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    char buffer[BUFFER_SIZE];
    socklen_t addr_len = sizeof(client_addr);
    ssize_t recv_len;

    // 1. Create a UDP socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    // 2. Configure server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;         // IPv4
    server_addr.sin_addr.s_addr = INADDR_ANY; // Listen on all interfaces
    server_addr.sin_port = htons(PORT);       // Server port

    // 3. Bind the socket to the port
    struct sockaddr *sock_addr = (struct sockaddr *)&server_addr;
    if (bind(sockfd, sock_addr, sizeof(server_addr)) < 0) {
        perror("bind failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }
    printf("UDP server listening on port %d...\n", PORT);

    // 4. Server loop: receive and echo
    while (1) {
        recv_len = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0,
                            (struct sockaddr *)&client_addr, &addr_len);
        if (recv_len < 0) {
            perror("recvfrom failed");
            continue;
        }
        buffer[recv_len] = '\0';  // Null-terminate the received message
        printf("Received from %s:%d: %s\n",
               inet_ntoa(client_addr.sin_addr),
               ntohs(client_addr.sin_port), buffer);
        // Echo the message back to the sender
        if (sendto(sockfd, buffer, recv_len, 0,
                   (struct sockaddr *)&client_addr, addr_len) < 0) {
            perror("sendto failed");
        }
    }
    // 5. Close the socket (unreachable in infinite loop)
    close(sockfd);
    return 0;
}
```

Task 3: BLE Device Scan on Windows and RPi machines

1. Install the **Python package** named 'bleak' on both the RPi and on your computer (Windows, not WSL2 Ubuntu) for using the onboard BLE device. This command installs 'bleak', which provides cross-platform BLE support, and shows its installed version.

```
pip3 install bleak && pip3 show bleak
```

2. Run the provided Python script on both the RPi and on your computer to scan nearby BLE devices. It outputs a JSON-formatted string showing information about discovered BLE devices.

3. Install **Nodejs v20.x** and the **Nodejs package** named 'noble' only on the RPi for using the onboard BLE device. This package will be used to scan nearby BLE devices.

```
# Remove any older Node.js.
$ sudo apt remove nodejs -y

# Add NodeSource repo for Node.js 20.x
$ curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -

# Install Node.js and npm.
$ sudo apt install -y nodejs

# Install noble package.
$ npm install @abandonware/noble
```

4. Run the provided Nodejs script on the RPi to scan nearby BLE devices. It outputs a JSON-formatted string showing information about discovered BLE devices.

File: ble_scan.py

```python
import asyncio
import json
from datetime import datetime
from bleak import BleakScanner

async def scan_ble():
    print("Scanning for nearby BLE devices...")
    devices = await BleakScanner.discover(timeout=5.0)

    results = []
    for device in devices:
        results.append({
            "address": device.address,
            "rssi": device.rssi,
            "name": device.name or "Unknown",
            "timestamp": datetime.now().isoformat()
        })

    print(json.dumps(results, indent=4))  # Pretty-print as JSON

if __name__ == "__main__":
    asyncio.run(scan_ble())
```

File: ble_scan.js

```
const noble = require('@abandonware/noble');
const devices = new Map();

noble.on('stateChange', async (state) => {
    if (state === 'poweredOn') {
        console.log("Scanning for nearby BLE devices...");
        await noble.startScanningAsync([], true);

        // Stop scan after 5 seconds
        setTimeout(async () => {
            await noble.stopScanningAsync();
            const result = Array.from(devices.values());
            console.log(JSON.stringify(result, null, 4));
            process.exit(0);
        }, 5000);
    } else {
        console.log(`Bluetooth state: ${state}`);
        await noble.stopScanningAsync();
    }
});

noble.on('discover', (peripheral) => {
    const { id, address, rssi, advertisement } = peripheral;
    const name = advertisement.localName || "Unknown";
    const timestamp = new Date().toISOString();
    const key = address && address !== 'unknown' ? address : id;
    devices.set(key, {
        address: address || id,
        rssi: rssi,
        name: name,
        timestamp: timestamp
    });
});
```

Task 4: Raspberry Pi Desktop Mode & RealView VNC

1. Upgrade the **Raspberry Pi OS Lite (64-bit)** to the **full Raspberry Pi Desktop mode.** Run the following commands to install the **X11 server**, **LightDM display manager**, **LXDE desktop session manager**:

```
$ sudo apt install -y raspberrypi-ui-mods rpi-update \
  lightdm xserver-xorg xinit lxterminal \
  gvfs gvfs-backends gvfs-fuse alsa-utils pavucontrol policykit-1 \
  gnome-disk-utility pcmanfm lxappearance lxsession chromium-browser \
  lxde-common lxpolkit

$ sudo systemctl set-default graphical.target

$ sudo reboot
```

2. Use the '`sudo raspi-config`' command to enable the **VNC Server on the RPi.**

3. Download and install **RealVNC Viewer on Windows** from the following web site:

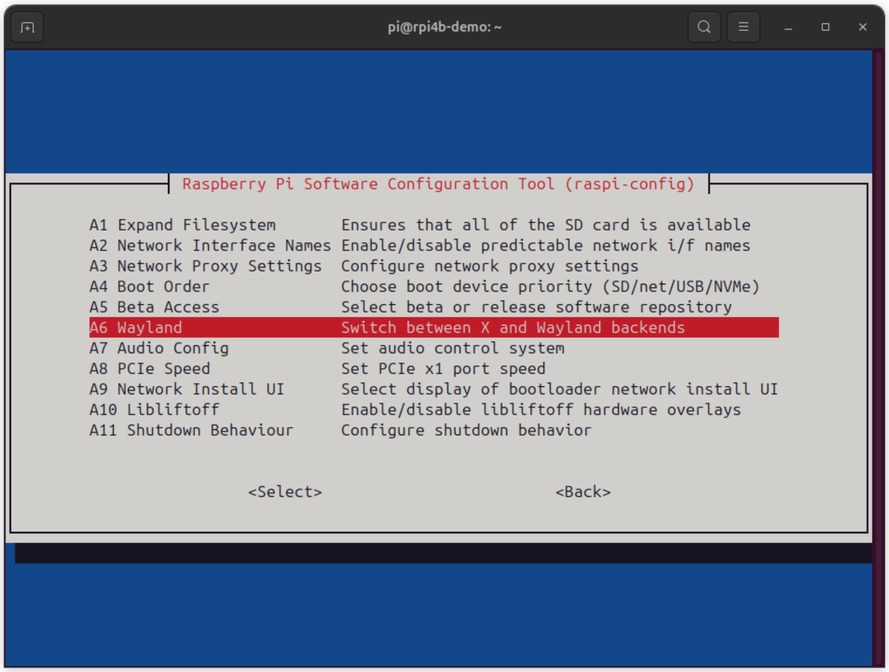
   - https://www.realvnc.com/en/connect/download/viewer/windows/

4. Open **RealVNC Viewer** to access the **VNC Server (default port 5900) on the RPi**.

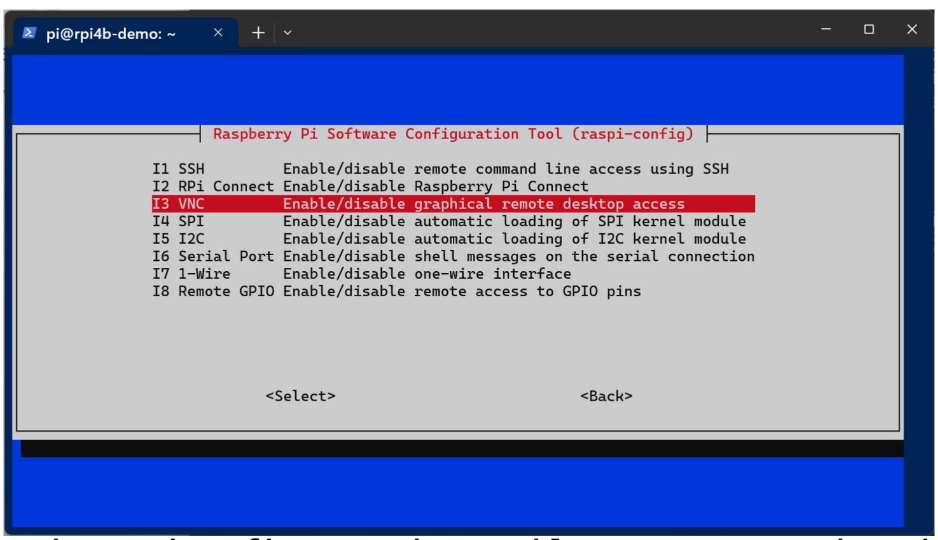
Note:

- The **RealVNC Viewer on Windows** may have some problems with the **VNC Server running Wayland.** Switching back to the **VNC Server for X11** may help solve the issue.

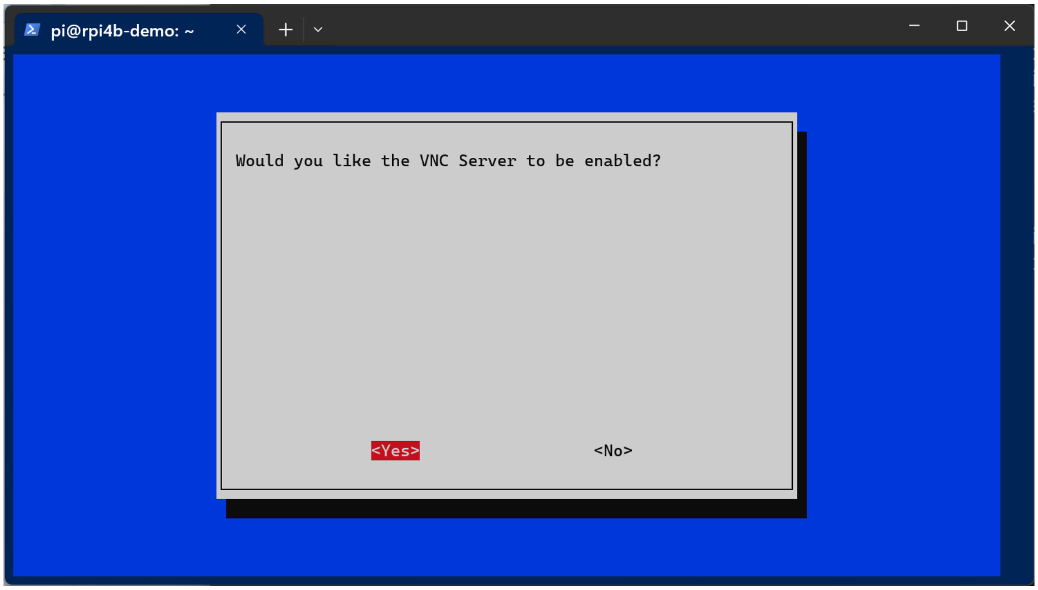
```
# Install RealVNC Server (X11-based)
$ sudo apt update
$ sudo apt install realvnc-vnc-server

# Check whether the  X11-based VNC server service is running.
$ sudo systemctl status vncserver-x11-serviced.service
```

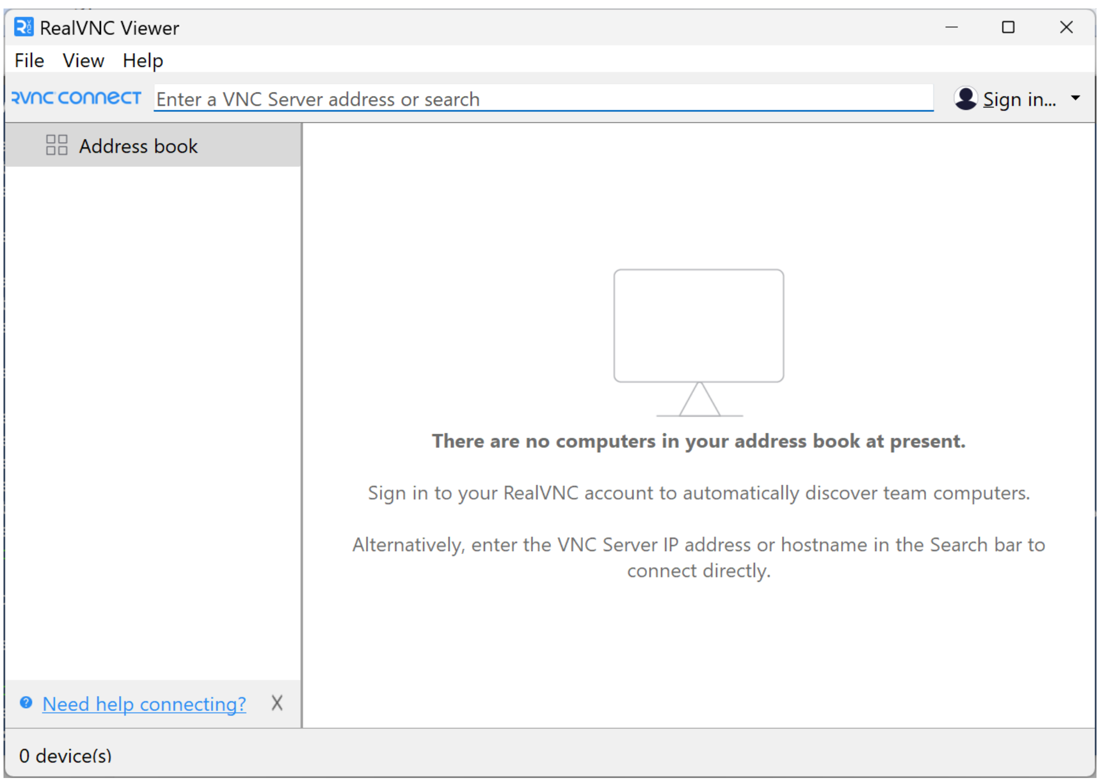


Switch between Wayland and X11 graphics backends

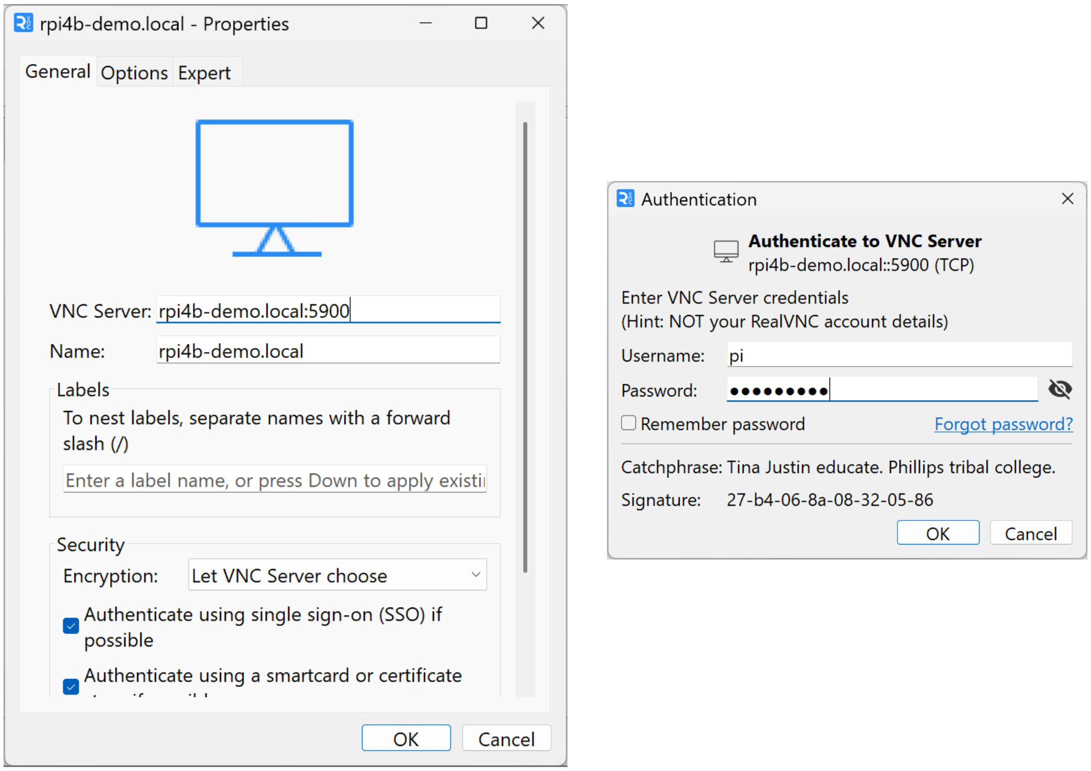


Use the raspi-config command to enable VNC server on the RPi

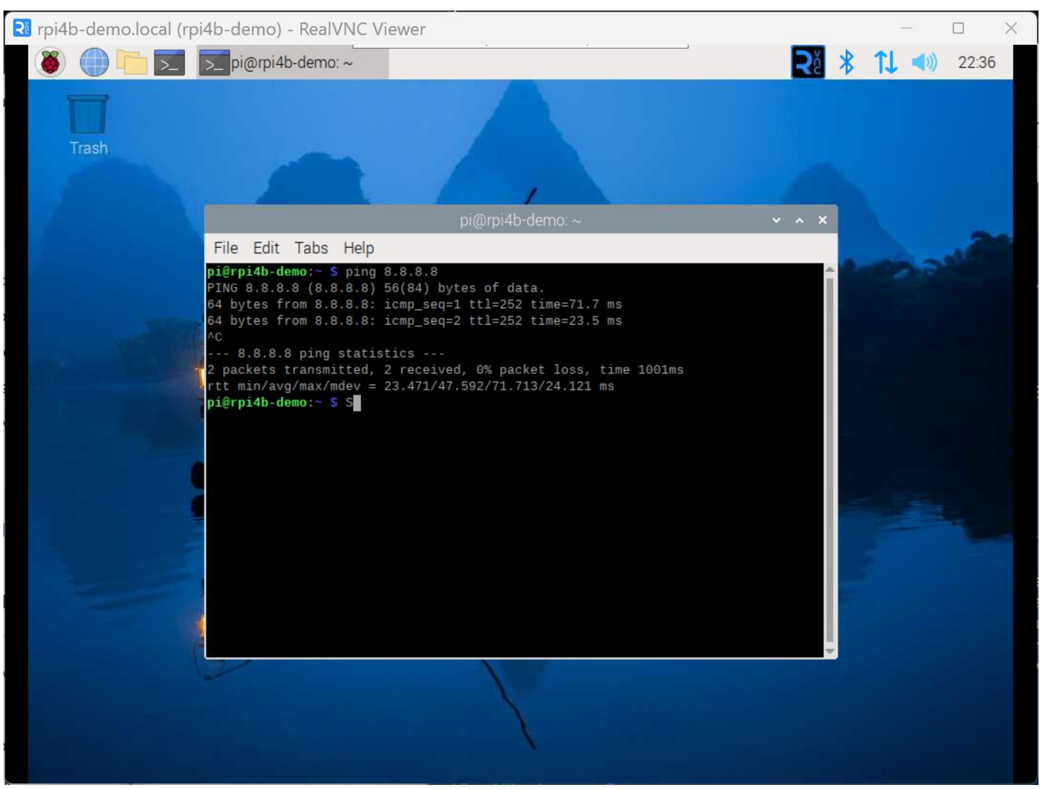




Open the RealVNC Viewer on Windows to connect a remote VNC Server



Connect the VNC Server running on the RPi (port 5900) and log in as pi.



Raspberry Pi Desktop accessed using the RealVNC Viewer on Windows.

Task 5: Connect to WSL2 Ubuntu from Raspberry Pi via SSH.

Assume that we have a **Raspberry Pi SBC** and a **Windows machine running WSL2 Ubuntu** on the same local network (e.g., 192.168.100.0/24). The following steps are required to allow the **RPi SBC** to connect to **WSL2 Ubuntu** via **SSH.** We need to forward a port (e.g., 2222) on **Windows** to the SSH default port (22) on **WSL2 Ubuntu.**

In this example, the following IPv4 addresses are used as example.

- Windows host: 192.168.100.38
- Raspberry Pi: 192.168.100.60 (pi : raspberry)
- WSL2 Ubuntu: 172.26.127.34 (ubuntu : ubuntu)

1. On **WSL2 Ubuntu**, make sure the SSH server is enabled.

```
$ sudo systemctl status ssh
```

If the **SSH2 server** is not installed or enabled, install or enable it first:

```
$ sudo apt install openssh-server
$ sudo systemctl enable ssh
```

You can get the IP address of the running **WSL2 Ubuntu** using the following command:

```
$ hostname -I
```

2. On **Windows host**, open the Windows PowerShell as Administrator and run the following commands.

```
# Get the IP address of the current WSL2 Ubuntu.
> wsl hostname -I

# Forward TCP port 2222 on the Windows host to port 22 on WSL2 Ubuntu.
> netsh interface portproxy add v4tov4 listenport=2222 `
  listenaddress=0.0.0.0 connectport=22 connectaddress=172.26.127.34

# Check port-forwarding configuration on Windows.
> netsh interface portproxy show all

# Check whether port 2222 is listening on Windows.
> netstat -an | findstr :2222

# Get the IPv4 address of the WiFi adapter on Windows.
> Get-NetIPAddress -InterfaceAlias "Wi-Fi" -AddressFamily IPv4 `
  | Select-Object IPAddress
```

3. On **Raspberry Pi**, run the following command.

```
# Try to connect from RPi to the WSL2 Ubuntu on the Windows host.
$ ssh ubuntu@192.168.100.38 -p 2222
```

Last update: 2025-07-06