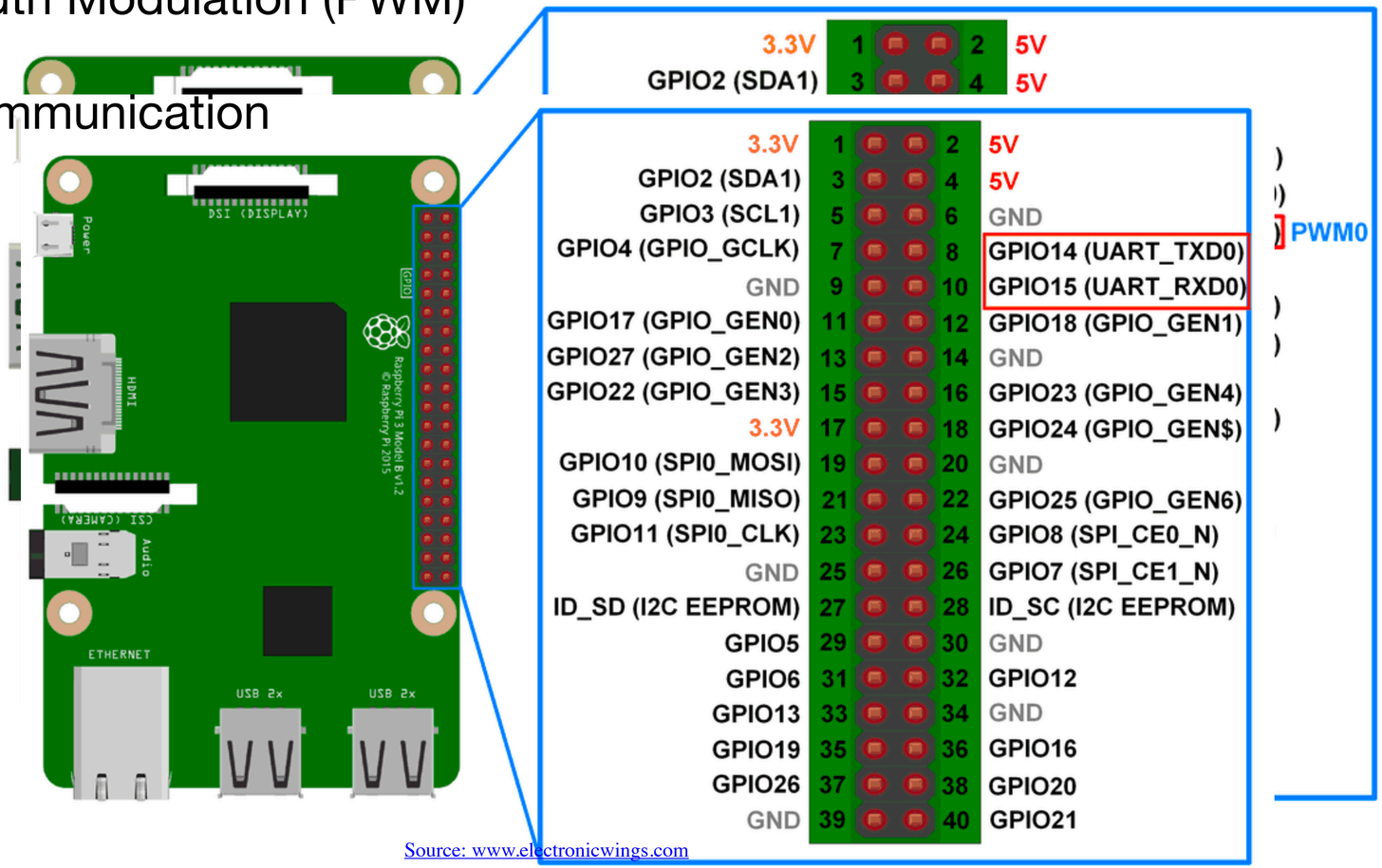
Lecture #4
Protocols & Interfaces

Android Things 2023

Peripheral I/O

- General Purpose Input/Output (GPIO)
- Pulse Width Modulation (PWM)
- Serial Communication

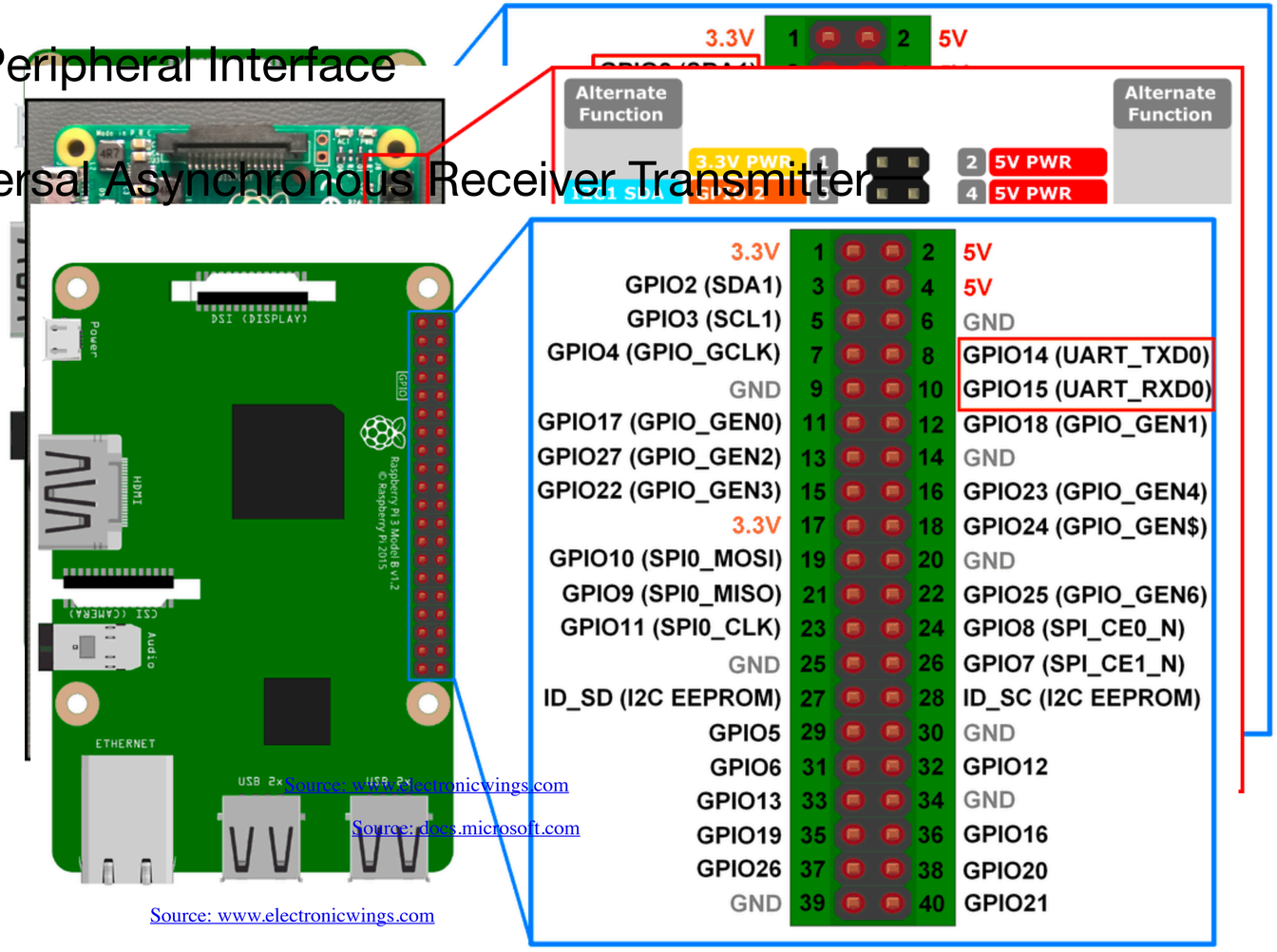


Source: www.electronicwings.com

Source: www.electronicwings.com

Serial Communication

- I2C - Inter-Integrated Circuit (IIC or I²C)
- SPI - Serial Peripheral Interface
- UART - Universal Asynchronous Receiver Transmitter

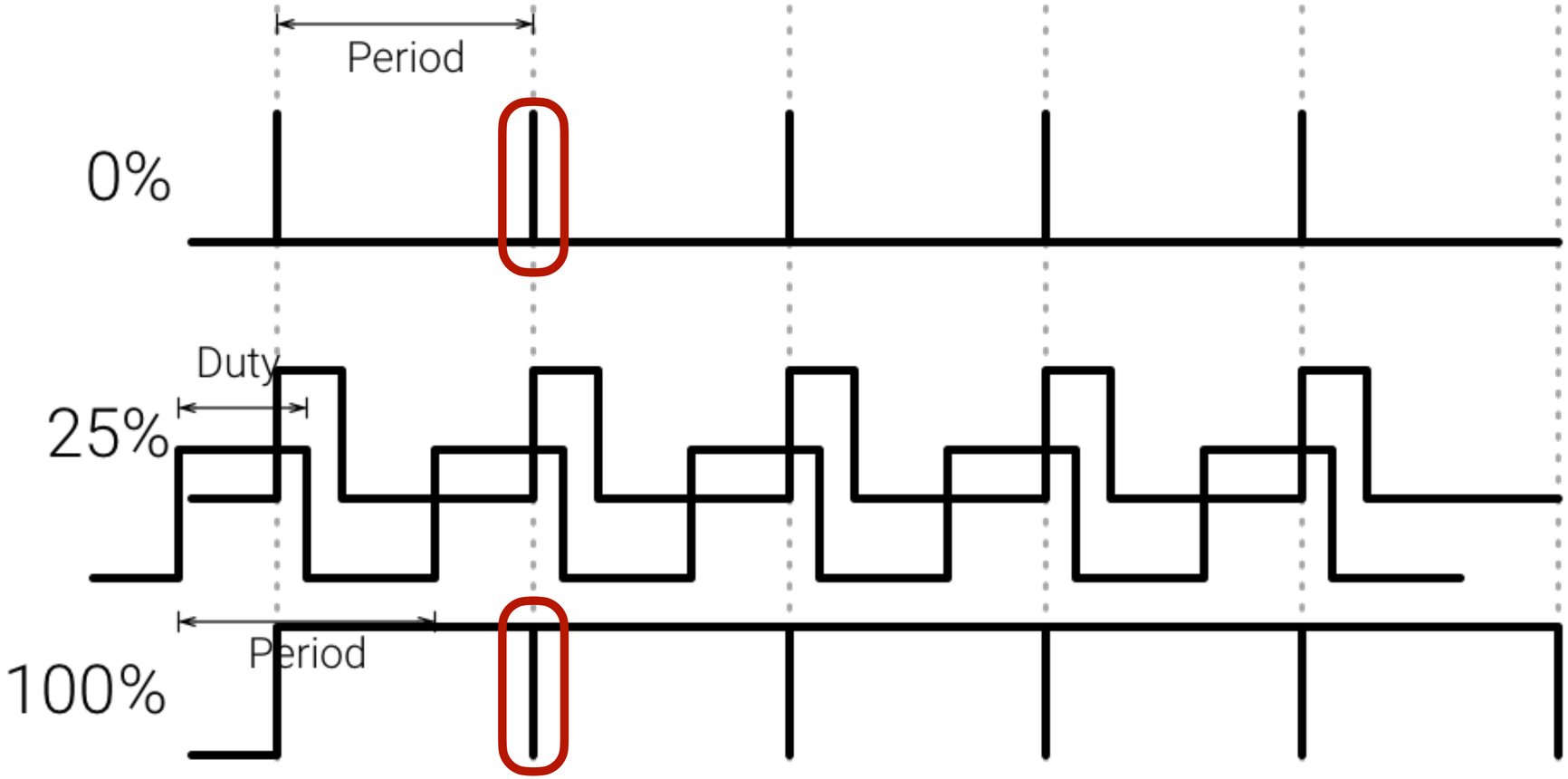


Source: www.electronicwings.com

Source: docs.microsoft.com

Source: www.electronicwings.com

Pulse Width Modulation



★ **Note:** Most PWM hardware has to toggle at least once per cycle, so even duty values of 0% and 100% will have a small transition at the beginning of each cycle.

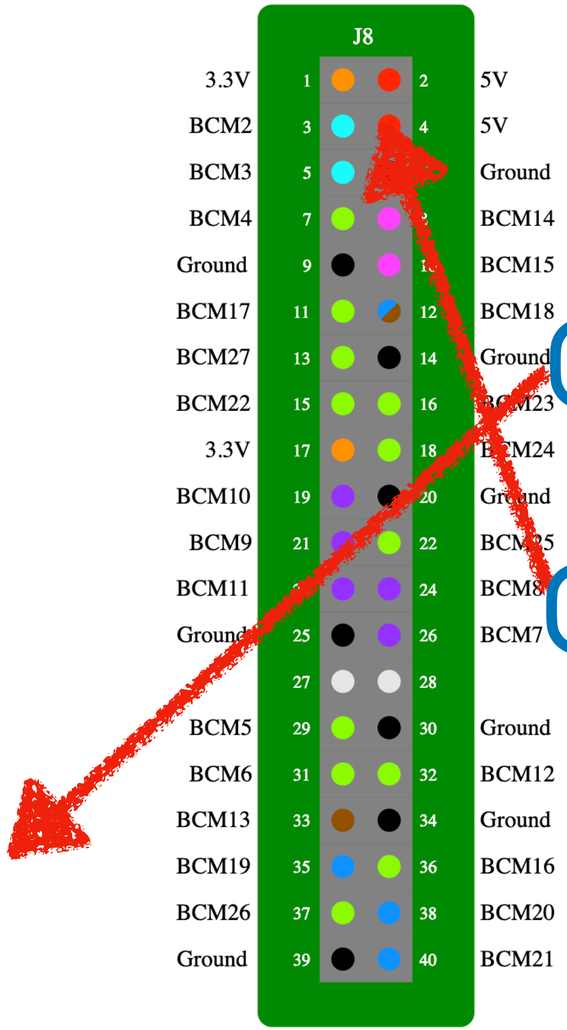
Permissions

```
<uses-permission
     android:name="com.google.android.things.permission.USE_PERIPHERAL_IO" />
```

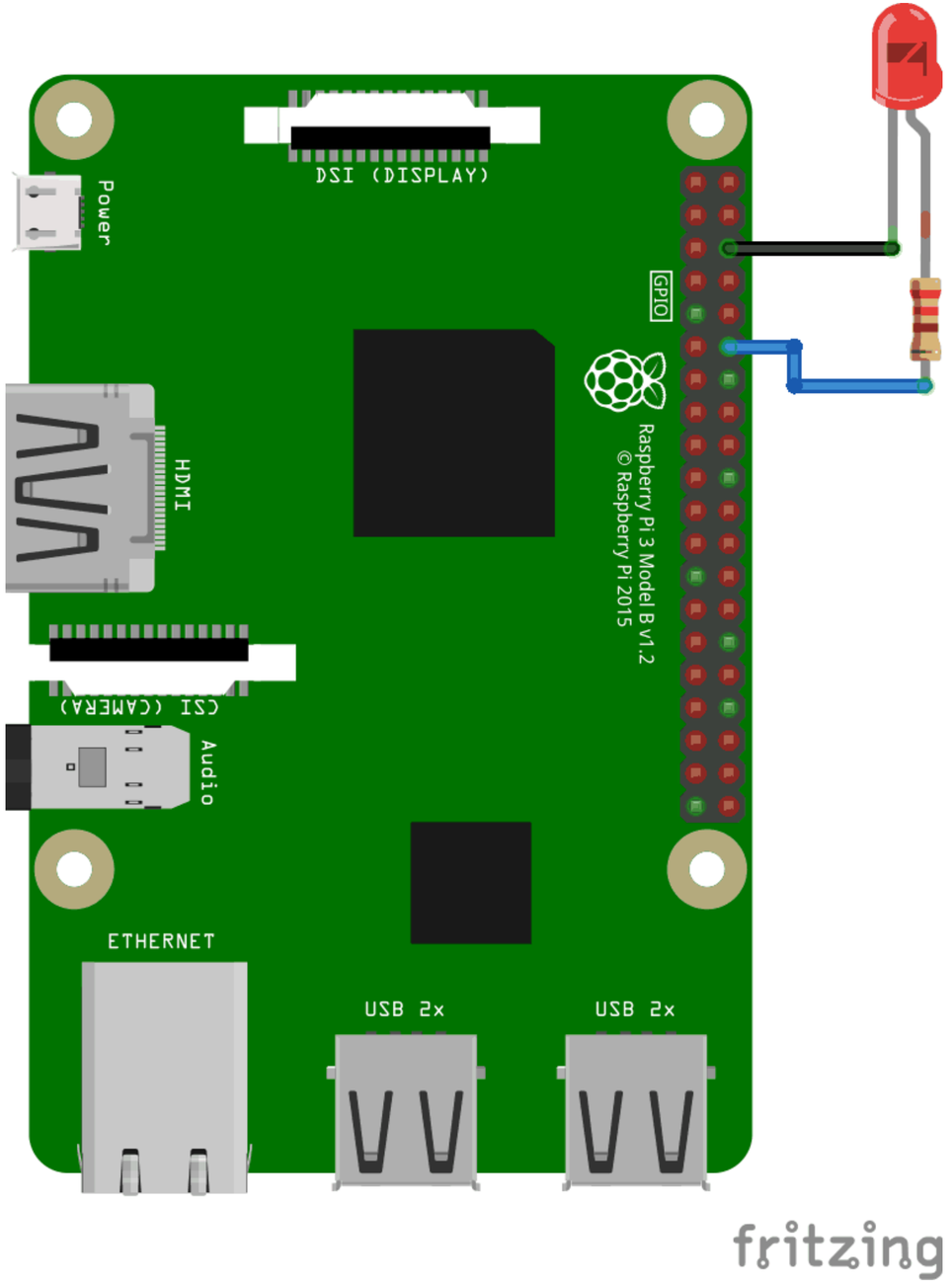
Managing the connection

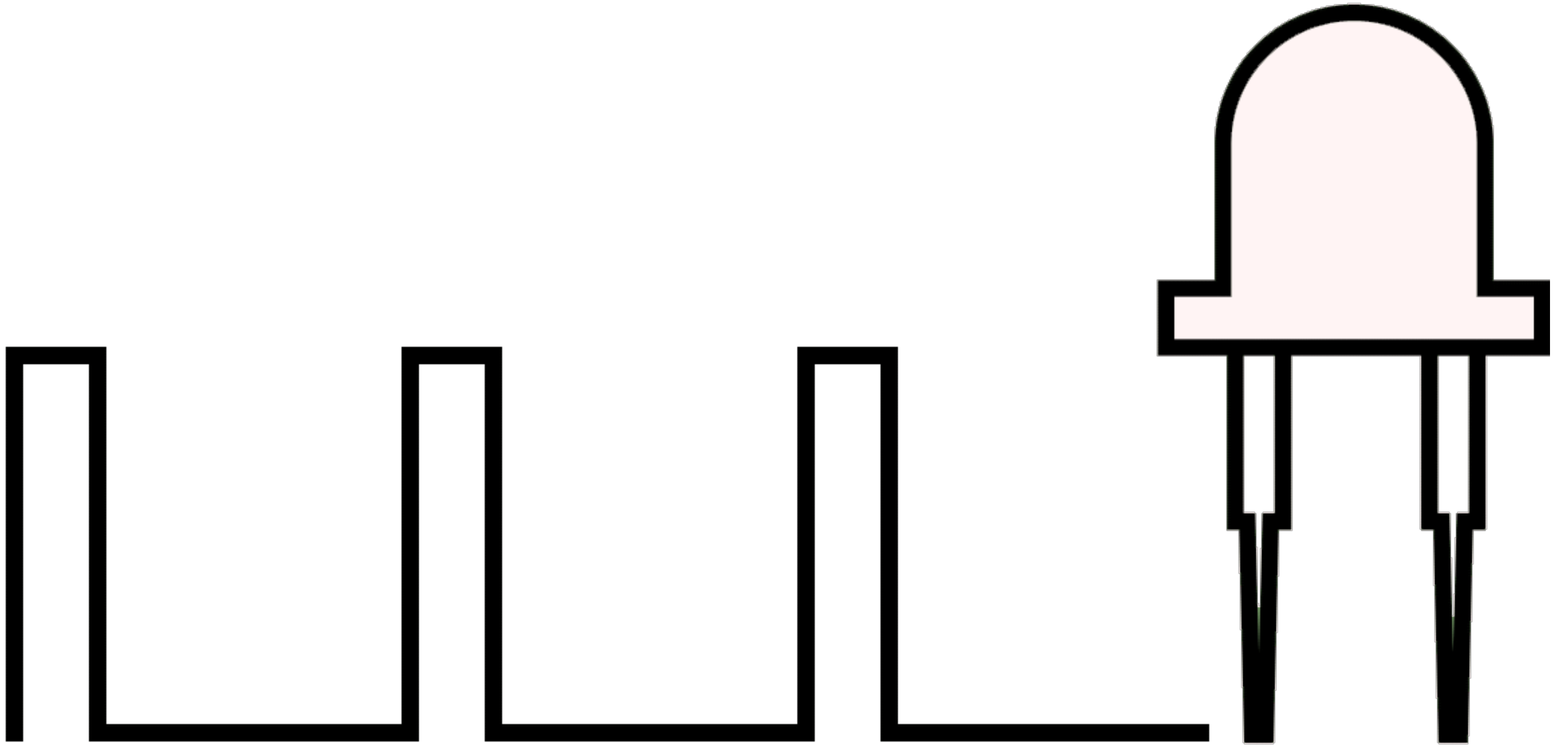
```
val manager = PeripheralManager.getInstance()
val portList: List<String> = manager.pwmList
if (portList.isEmpty()) {
    Log.i(TAG, "No PWM port available on this device.")
} else {
    Log.i(TAG, "List of available ports: $portList")
}
```

Pinout



GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	


DSI (DISPLAY)
Power
GPIO
HDMI
CSI (CAMERA)
Audio
Raspberry Pi 3 Model B v1.2
© Raspberry Pi 2015
ETHERNET
USB 2x
USB 2x
fritzing



Access the PWM port

```
// PWM Name
private const val PWM_NAME = ...
class HomeActivity : Activity() {
  private var pwm: Pwm? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the PWM port
    pwm = try {
      PeripheralManager.getInstance()
        .openPwm(PWM_NAME)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access PWM", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
    try {
      pwm?.close()
      pwm = null
    } catch (e: IOException) {
      Log.w(TAG, "Unable to close PWM", e)
```

★ **Note:** A pin configured for PWM continues to output its signal even after the close() method is called. Call setEnabled(false) to stop the signal.

Controlling the PWM signal

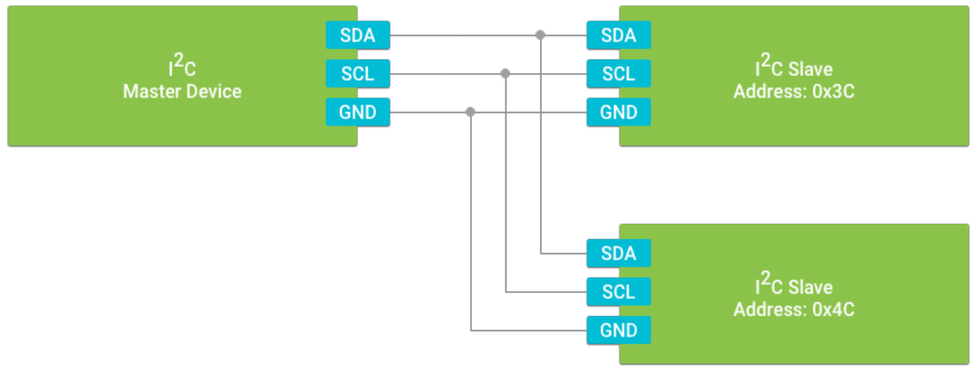
```
@Throws(IOException::class)
fun initializePwm(pwm: Pwm) {
  pwm.apply {
    setPwmFrequencyHz(120.0)
    setPwmDutyCycle(25.0)

    // Enable the PWM signal
    setEnabled(true)
  }
}
```

```
import RPi.GPIO as GPIO
from time import sleep

ledpin = 12        # PWM pin connected to LED
GPIO.setwarnings(False)      #disable warnings
GPIO.setmode(GPIO.BOARD)    #set pin numbering system
GPIO.setup(ledpin,GPIO.OUT)
pi_pwm = GPIO.PWM(ledpin,1000)    #create PWM instance with frequency
pi_pwm.start(0)        #start PWM of required Duty Cycle
while True:
    for duty in range(0,101,1):
        pi_pwm.ChangeDutyCycle(duty) #provide duty cycle in the range 0-100
        sleep(0.01)
    sleep(0.5)

    for duty in range(100,-1,-1):
        pi_pwm.ChangeDutyCycle(duty)
        sleep(0.01)
    sleep(0.5)
```

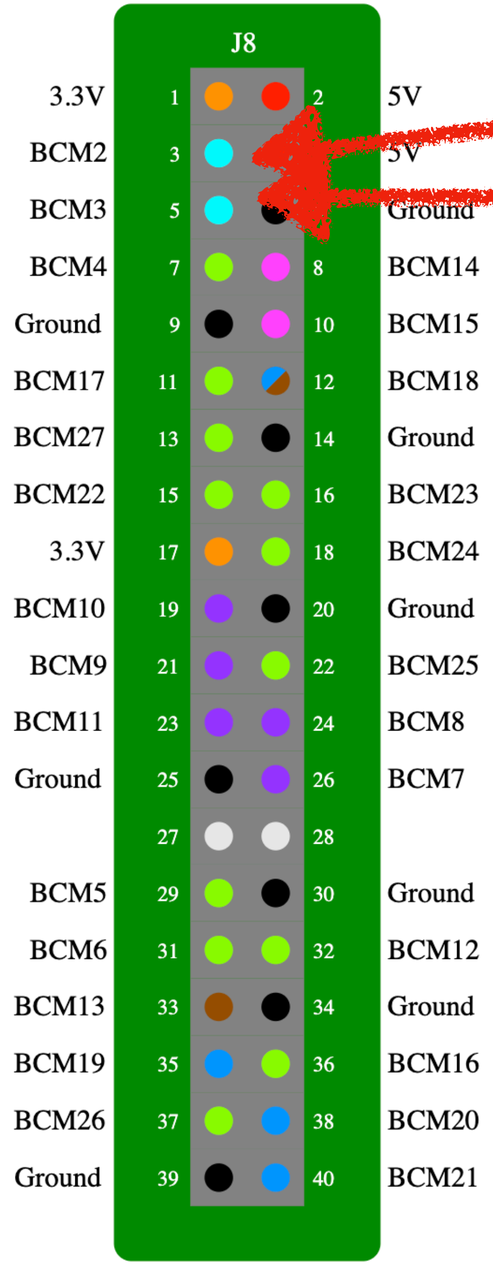
```rust
use rppal::pwm::{Channel, Polarity, Pwm};
use std::thread::sleep;
use std::time::Duration;

fn main() {
    // Initialize the PWM channel with a frequency of 100 Hz and a duty cycle of 50%
    let mut pwm = Pwm::new(Channel::Pwm0).unwrap();
    pwm.set_frequency(100.0).unwrap();
    pwm.set_polarity(Polarity::Normal).unwrap();
    pwm.set_duty_cycle(50.0).unwrap();
    pwm.enable().unwrap();

    // Fade the LED up and down
    loop {
        for duty_cycle in (0..=100).step_by(1) {
            pwm.set_duty_cycle(duty_cycle as f64).unwrap();
            sleep(Duration::from_millis(10));
        }
        for duty_cycle in (0..=100).rev().step_by(1) {
            pwm.set_duty_cycle(duty_cycle as f64).unwrap();
            sleep(Duration::from_millis(10));
        }
    }
}
```

Inter-Integrated Circuit
I2C

- I2C is a synchronous serial interface.
  - Relies on a shared clock signal to synchronize data transfer between devices.
- The device in control of triggering the clock signal is known as the **master**.
  - Shared clock signal (SCL)
  - Shared data line (SDA)
  - Common ground reference (GND)
- All other connected peripherals are known as **slaves**.
- Each device is connected to the same set of data signals to form a **bus**.



= 5V = 1.8V = GPIO = I2C = SPI
= 3.3V = Ground = PWM = I2S = UART



GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	

Adding the required permissions

```
<uses-permission
     android:name="com.google.android.things.permission.USE_PERIPHERAL_IO" />
```

Managing the slave device connection

```
val manager = PeripheralManager.getInstance()
val deviceList: List<String> = manager.i2cBusList
if (deviceList.isEmpty()) {
  Log.i(TAG, "No I2C bus available on this device.")
} else {
  Log.i(TAG, "List of available devices: $deviceList")
}
```

> **Note:** The device name represents the I²C bus, and the address represents the individual slave on that bus. Therefore, an I2cDevice is a connection to a specific slave device on the corresponding I²C bus.

Access the I2C device

```
// I2C Device Name
private const val I2C_DEVICE_NAME: String = ...
// I2C Slave Address
private const val I2C_ADDRESS: Int = ...
class HomeActivity : Activity() {
  private var mDevice: I2cDevice? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the I2C device
    mDevice = try {
      PeripheralManager.getInstance()
        .openI2cDevice(I2C_DEVICE_NAME, I2C_ADDRESS)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access I2C device", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
      try {
        mDevice?.close()
        mDevice = null
      } catch (e: IOException) {
        Log.w(TAG, "Unable to close I2C device", e)
```

Determine available addresses

```
fun PeripheralManager.scanI2cAvailableAddresses(i2cName: String): List<Int> {
  return (0..127).filter { address ->
    with(openI2cDevice(i2cName, address)) {
      try {
        write(ByteArray(1), 1)
        true
      } catch (e: IOException) {
        false
      } finally {
        close()
      }
    }
  }
}

Log.i(TAG, "Scanning I2C devices")
manager.scanI2cAvailableAddresses(I2C_BUS_NAME)
    .map { String.format(Locale.US, "0x%02X", it) }
    .forEach { address -> Log.i(TAG, "Found: $address") }
```

```
Scanning I2C devices
Found: 0x3C
Found: 0x3F
Found: 0x42
```

Interacting with registers


```
// Modify the contents of a single register
@Throws(IOException::class)
fun setRegisterFlag(device: I2cDevice, address: Int) {
    // Read one register from slave
    var value = device.readRegByte(address)
    // Set bit 6
    value = value or 0x40
    // Write the updated value back to slave
    device.writeRegByte(address, value)
}
```

```
// Read a register block
@Throws(IOException::class)
fun readCalibration(device: I2cDevice, startAddress: Int): ByteArray {
    // Read three consecutive register values
    return ByteArray(3).also { data ->
        device.readRegBuffer(startAddress, data, data.size)
    }
}
```

- Byte Data: **readRegByte**() and **writeRegByte**() Read or write a single 8-bit register value.
- Word Data: **readRegWord**() and **writeRegWord**() Read or write two consecutive register values as a 16-bit little-endian word. The first register address corresponds to the least significant byte (LSB) in the word, followed by the most significant byte (MSB).
- Block Data: **readRegBuffer**() and **writeRegBuffer**() Read or write up to 32 consecutive register values as an array.

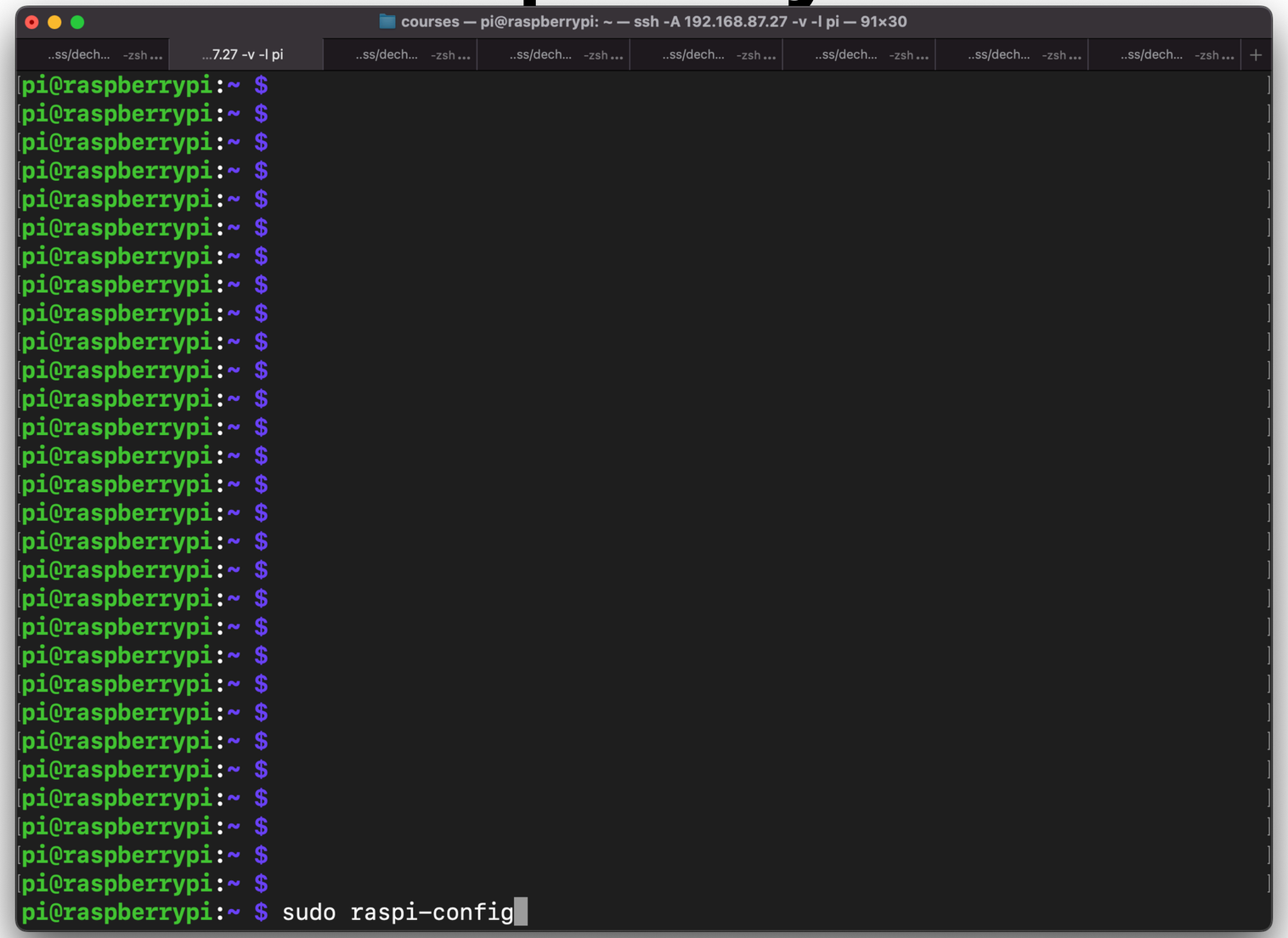
Transferring raw data


```
@Throws(IOException::class)
fun writeBuffer(device: I2cDevice, buffer: ByteArray) {
    device.write(buffer, buffer.size).also { count ->
        Log.d(TAG, "Wrote $count bytes over I2C.")
    }
}
```

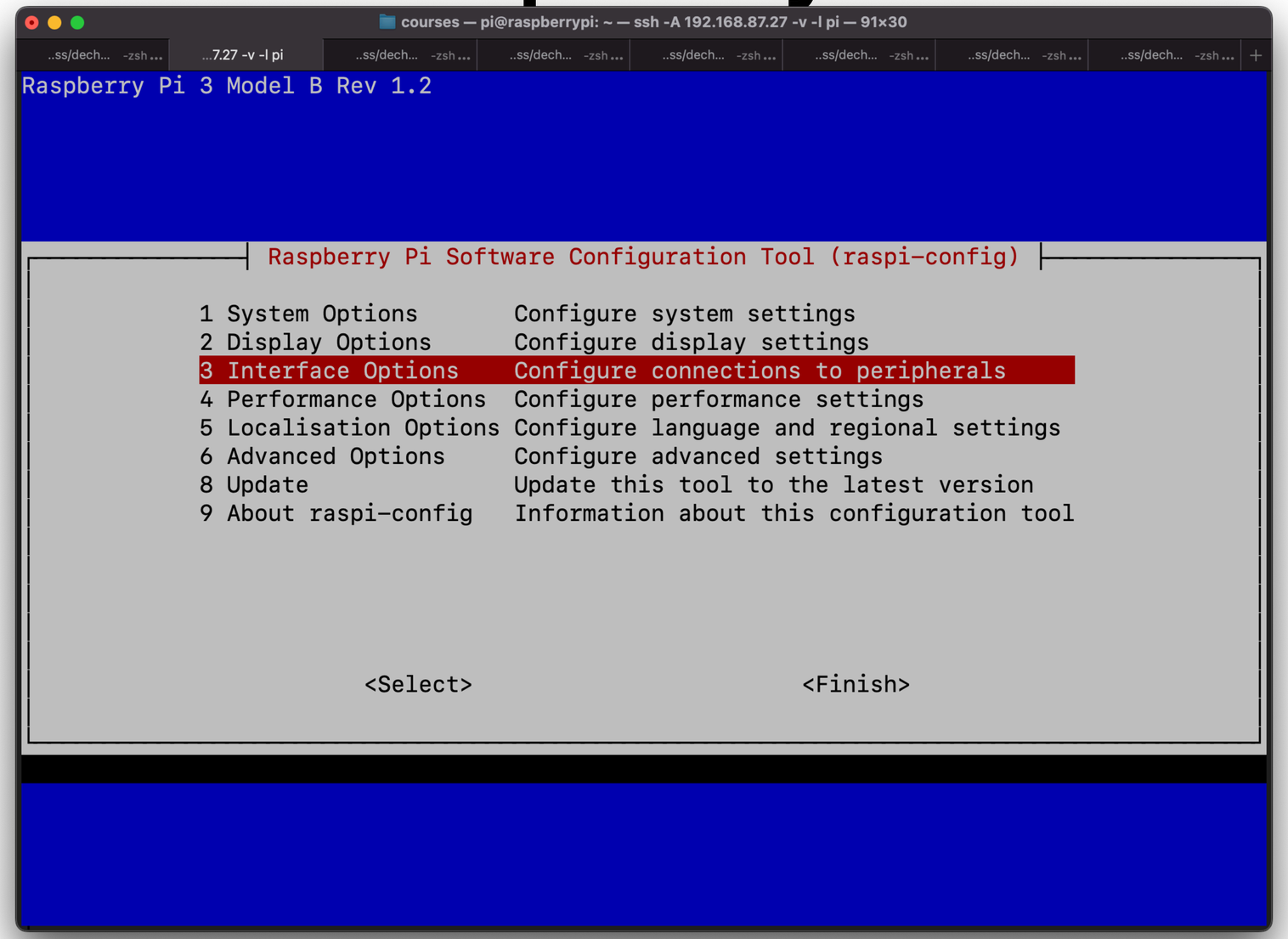
★ **Note:** There is no explicit maximum length that a raw transaction can handle, but the I^2C controller hardware on your device may have a limit on the number of bytes it can process. Consult your device hardware documentation if your peripheral requires large data transfers.

https://github.com/Nilhcem/i2cfun-androidthings/

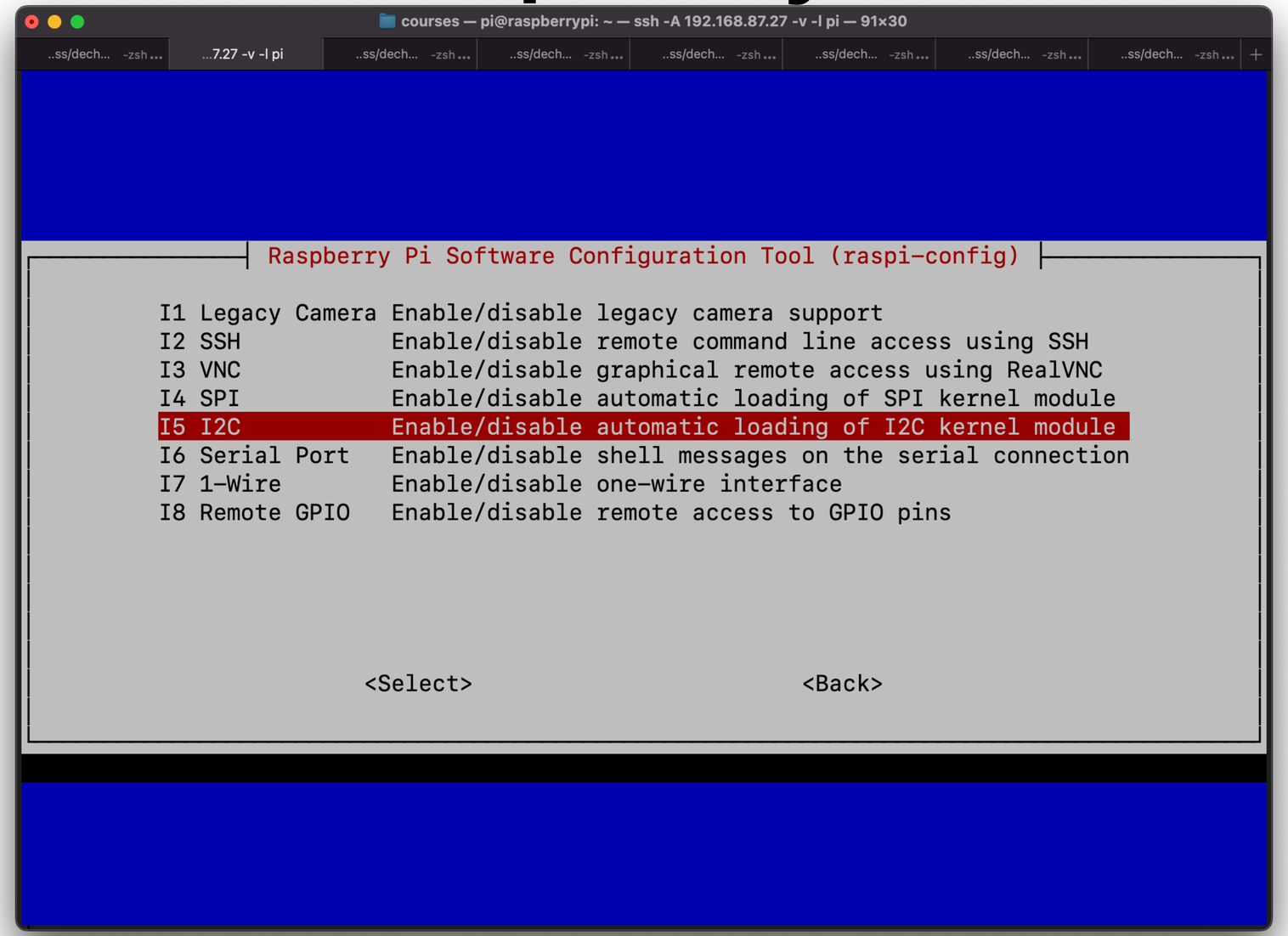
Enable I2C
on Raspberry Pi OS



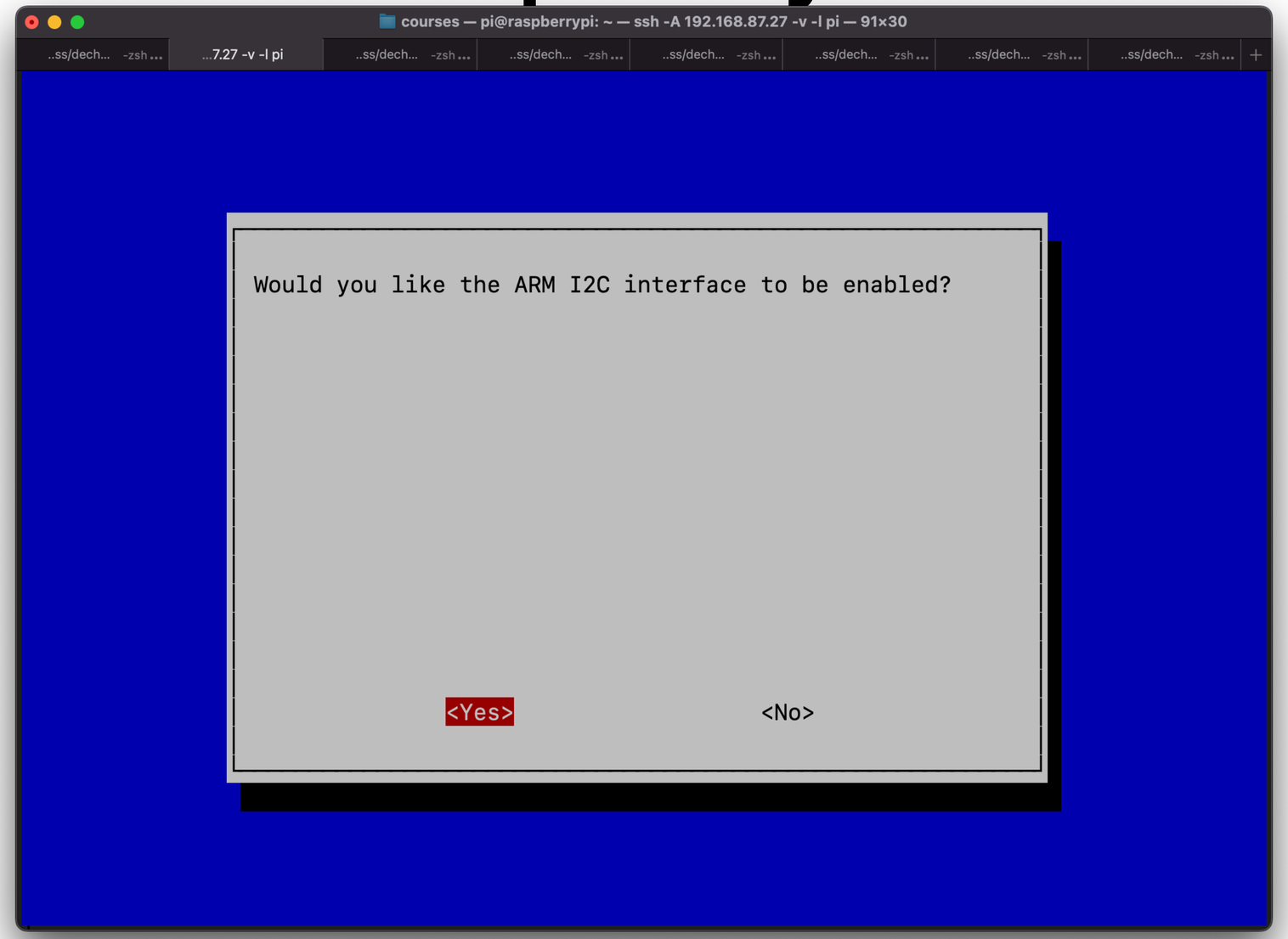
Enable I2C on Raspberry Pi OS



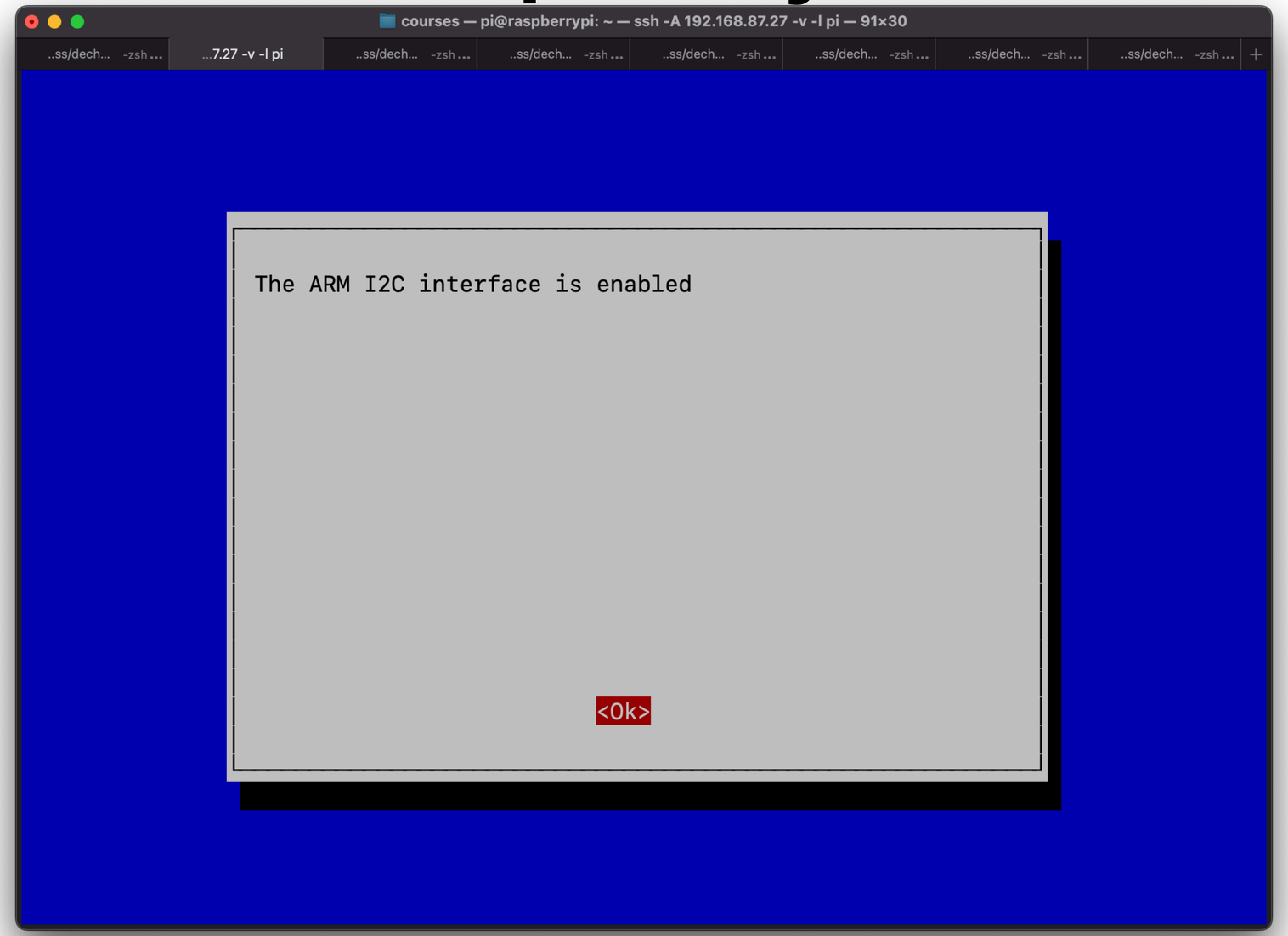
Enable I2C on Raspberry Pi OS



Enable I2C on Raspberry Pi OS



Enable I2C
on Raspberry Pi OS



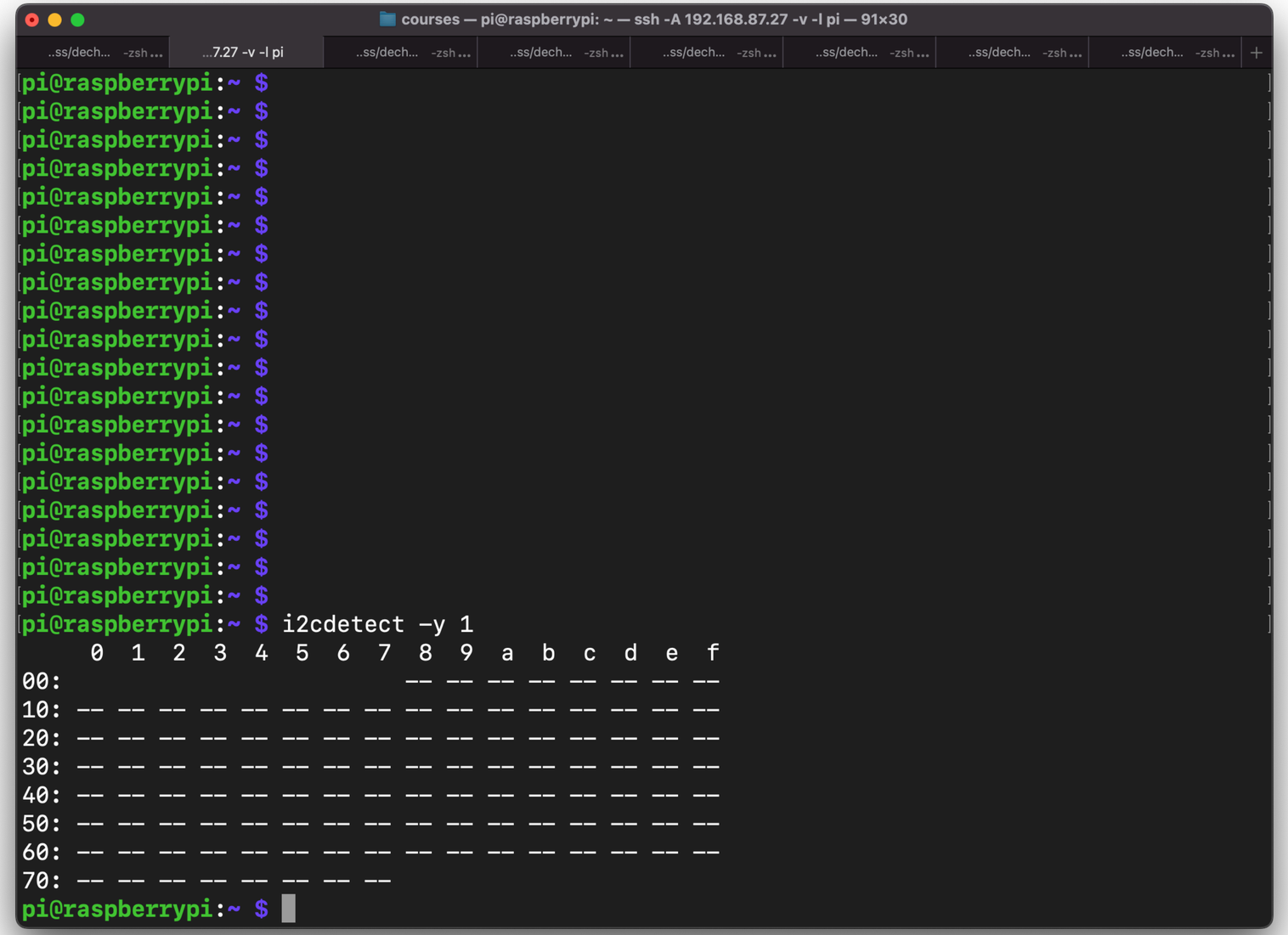
Alternative

/boot/config.txt

```
# Uncomment some or all of these to enable the optional hardware interfaces
dtparam=i2c_arm=on
```

Confirm setup

- i2cdetect -y 1



Determine available addresses

```
import smbus

# Define the I2C bus number (usually 1 on newer Raspberry Pis)
I2C_BUS = 1

# Scan for I2C devices on the bus
def scan_i2c():
    print("Scanning I2C bus", I2C_BUS, "for devices...")
    bus = smbus.SMBus(I2C_BUS)
    devices = []
    for address in range(0x03, 0x78):
        try:
            bus.read_byte(address)
            devices.append(hex(address))
            print("Device found at address", hex(address))
        except:
            pass
    if not devices:
        print("No devices found on the I2C bus.")
    return devices

# Call the scan_i2c function to scan for devices
devices = scan_i2c()
```

```rust
use i2cdev::linux::LinuxI2CError;
use linux_embedded_hal::{Delay, I2cdev};

const I2C_BUS: &str = "/dev/i2c-1";

fn scan_i2c() -> Result<Vec<u8>, LinuxI2CError> {
    println!("Scanning I2C bus {} for devices...", I2C_BUS);

    let i2c_bus = I2cdev::new(I2C_BUS)?;
    let mut devices = Vec::new();

    for address in 0x03..0x78 {
        if i2c_bus.smbus_read_byte(address).is_ok() {
            devices.push(address);
            println!("Device found at address 0x{:02X}", address);
        }
    }

    if devices.is_empty() {
        println!("No devices found on the I2C bus.");
    }

    Ok(devices)
}
```

```rust
const I2C_BUS: &str = "/dev/i2c-1";

fn scan_i2c() -> Result<Vec<u8>, LinuxI2CError> {
    println!("Scanning I2C bus {} for devices...", I2C_BUS);

    let i2c_bus = I2cdev::new(I2C_BUS)?;
    let mut devices = Vec::new();

    for address in 0x03..0x78 {
        if i2c_bus.smbus_read_byte(address).is_ok() {
            devices.push(address);
            println!("Device found at address 0x{:02X}", address);
        }
    }

    if devices.is_empty() {
        println!("No devices found on the I2C bus.");
    }

    Ok(devices)
}

fn main() {
    let devices = scan_i2c().unwrap();
    println!("Found devices: {:?}", devices);
}
```

Sample Read&Write

```
import smbus
import time
bus = smbus.SMBus(1)
address = 0x70

def writeValue(value):
        bus.write_byte_data(address, 0, value)
        return -1

def readValue():
        value = bus.read_byte_data(address, 1)
        return value

while True:
        writeValue(0x51)
        time.sleep(0.7)
        value = readValue()
        print value
```

Sample Temperature Reader

```
import smbus2
import time

# Initialize I2C bus (default is 1 on Raspberry Pi)
bus = smbus2.SMBus(1)

# Define I2C address of the device you want to communicate with
DEVICE_ADDRESS = 0x40

# Define register addresses of the device you want to read from/write to
REGISTER_ADDR1 = 0x01
REGISTER_ADDR2 = 0x02

# Read data from the device
def read_data():
    # Read 2 bytes of data from the device at register address 0x01
    data = bus.read_word_data(DEVICE_ADDRESS, REGISTER_ADDR1)

    # Convert data to temperature in degrees Celsius
```

```
import smbus2
import time

# Initialize I2C bus (default is 1 on Raspberry Pi)
bus = smbus2.SMBus(1)

# Define I2C address of the device you want to communicate with
DEVICE_ADDRESS = 0x40

# Define register addresses of the device you want to read from/write to
REGISTER_ADDR1 = 0x01
REGISTER_ADDR2 = 0x02

# Read data from the device
def read_data():
    # Read 2 bytes of data from the device at register address 0x01
    data = bus.read_word_data(DEVICE_ADDRESS, REGISTER_ADDR1)

    # Convert data to temperature in degrees Celsius
    temperature = (data / 65535.0) * 165.0 - 40.0

    # Print temperature
    print("Temperature: {:.2f} °C".format(temperature))

# Write data to the device
def write_data():
    # Write 0x1234 to register address 0x02 of the device
```

```
    # Read 2 bytes of data from the device at register address 0x01
    data = bus.read_word_data(DEVICE_ADDRESS, REGISTER_ADDR1)

    # Convert data to temperature in degrees Celsius
    temperature = (data / 65535.0) * 165.0 - 40.0

    # Print temperature
    print("Temperature: {:.2f} °C".format(temperature))

# Write data to the device
def write_data():
    # Write 0x1234 to register address 0x02 of the device
    bus.write_word_data(DEVICE_ADDRESS, REGISTER_ADDR2, 0x1234)

    # Wait for 1 second to allow the device to process the data
    time.sleep(1)

    # Read the data back from the device
    data = bus.read_word_data(DEVICE_ADDRESS, REGISTER_ADDR2)

    # Print the data
    print("Data written to the device: 0x{:04X}".format(0x1234))
    print("Data read back from the device: 0x{:04X}".format(data))

# Call read_data and write_data functions
read_data()
write_data()
```

Rust Sample

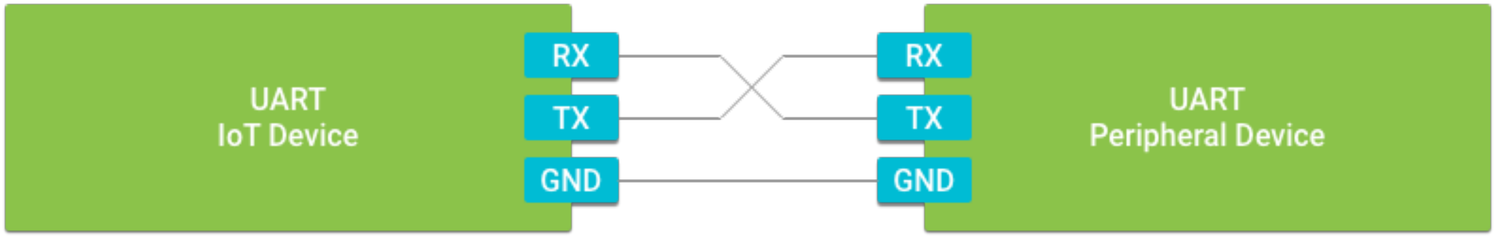
```rust
use linux_embedded_hal::I2cdev;
use nb::block;

// Define I2C address of the device you want to communicate with
const DEVICE_ADDRESS: u16 = 0x40;

// Define register addresses of the device you want to read from/write to
const REGISTER_ADDR1: u8 = 0x01;
const REGISTER_ADDR2: u8 = 0x02;

fn main() {
    // Initialize I2C bus (default is 1 on Raspberry Pi)
    let i2c_bus = I2cdev::new("/dev/i2c-1").unwrap();

    // Read data from the device
    read_data(&i2c_bus);

    // Write data to the device
    write_data(&i2c_bus);
}
```

```rust
use linux_embedded_hal::I2cdev;
use nb::block;

// Define I2C address of the device you want to communicate with
const DEVICE_ADDRESS: u16 = 0x40;

// Define register addresses of the device you want to read from/write to
const REGISTER_ADDR1: u8 = 0x01;
const REGISTER_ADDR2: u8 = 0x02;

fn main() {
    // Initialize I2C bus (default is 1 on Raspberry Pi)
    let i2c_bus = I2cdev::new("/dev/i2c-1").unwrap();

    // Read data from the device
    read_data(&i2c_bus);

    // Write data to the device
    write_data(&i2c_bus);
}

// Read data from the device
fn read_data(i2c_bus: &I2cdev) {
    // Read 2 bytes of data from the device at register address 0x01
    let data = block!(i2c_bus.read_word(DEVICE_ADDRESS, REGISTER_ADDR1)).unwrap();
```

```rust
// Read data from the device
fn read_data(i2c_bus: &I2cdev) {
    // Read 2 bytes of data from the device at register address 0x01
    let data = block!(i2c_bus.read_word(DEVICE_ADDRESS, REGISTER_ADDR1)).unwrap();

    // Convert data to temperature in degrees Celsius
    let temperature = (data as f32 / 65535.0) * 165.0 - 40.0;

    // Print temperature
    println!("Temperature: {:.2} °C", temperature);
}

// Write data to the device
fn write_data(i2c_bus: &I2cdev) {
    // Write 0x1234 to register address 0x02 of the device
    block!(i2c_bus.write_word(DEVICE_ADDRESS, REGISTER_ADDR2, 0x1234)).unwrap();

    // Wait for 1 second to allow the device to process the data
    std::thread::sleep(std::time::Duration::from_secs(1));

    // Read the data back from the device
    let data = block!(i2c_bus.read_word(DEVICE_ADDRESS, REGISTER_ADDR2)).unwrap();

    // Print the data
    println!("Data written to the device: 0x{:04X}", 0x1234);
    println!("Data read back from the device: 0x{:04X}", data);
}
```

```
[dependencies]
linux_embedded_hal = "0.5.1"
nb = "1.0.0"
```

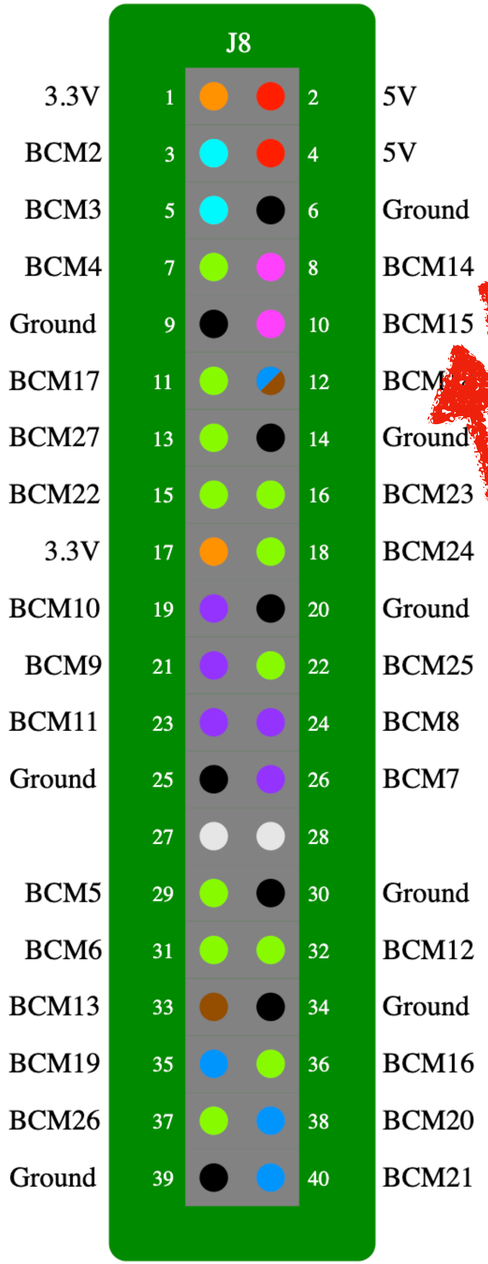
Universal Asynchronous Receiver Transmitter - UART



- GPS modules.
- LCD displays.

- 3-Wire ports include data receive (RX), data transmit (TX), and ground reference (GND) signals.
- 5-Wire ports add request to send (RTS) and clear to send (CTS) signals used for hardware flow control.

= 5V = 1.8V = GPIO = I2C = SPI
= 3.3V = Ground = PWM = I2S = UART



GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	

Managing the connection

```
<uses-permission android:name="com.google.android.things.permission.USE_PERIPHERAL_IO" />
```

```
val manager = PeripheralManager.getInstance()
val deviceList: List<String> = manager.uartDeviceList
if (deviceList.isEmpty()) {
  Log.i(TAG, "No UART port available on this device.")
} else {
  Log.i(TAG, "List of available devices: $deviceList")
}
```

Access UART Device

```
// UART Device Name
private val UART_DEVICE_NAME: String = ...
class HomeActivity : Activity() {
  private var mDevice: UartDevice? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the UART device
    mDevice = try {
      PeripheralManager.getInstance()
        .openUartDevice(UART_DEVICE_NAME)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access UART device", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
    try {
      mDevice?.close()
      mDevice = null
    } catch (e: IOException) {
      Log.w(TAG, "Unable to close UART device", e)
    }
  }
}
```

Configuring port parameters

Note: The default configuration for most UART devices is 8 data bits, no parity, and 1 stop bit (8N1).

```
@Throws(IOException::class)
fun configureUartFrame(uart: UartDevice) {
  uart.apply {
    // Configure the UART port
    setBaudrate(115200)
    setDataSize(8)
    setParity(UartDevice.PARITY_NONE)
    setStopBits(1)
  }
}
```

https://developer.android.com/things/sdk/pio/uart

Transmitting outgoing data

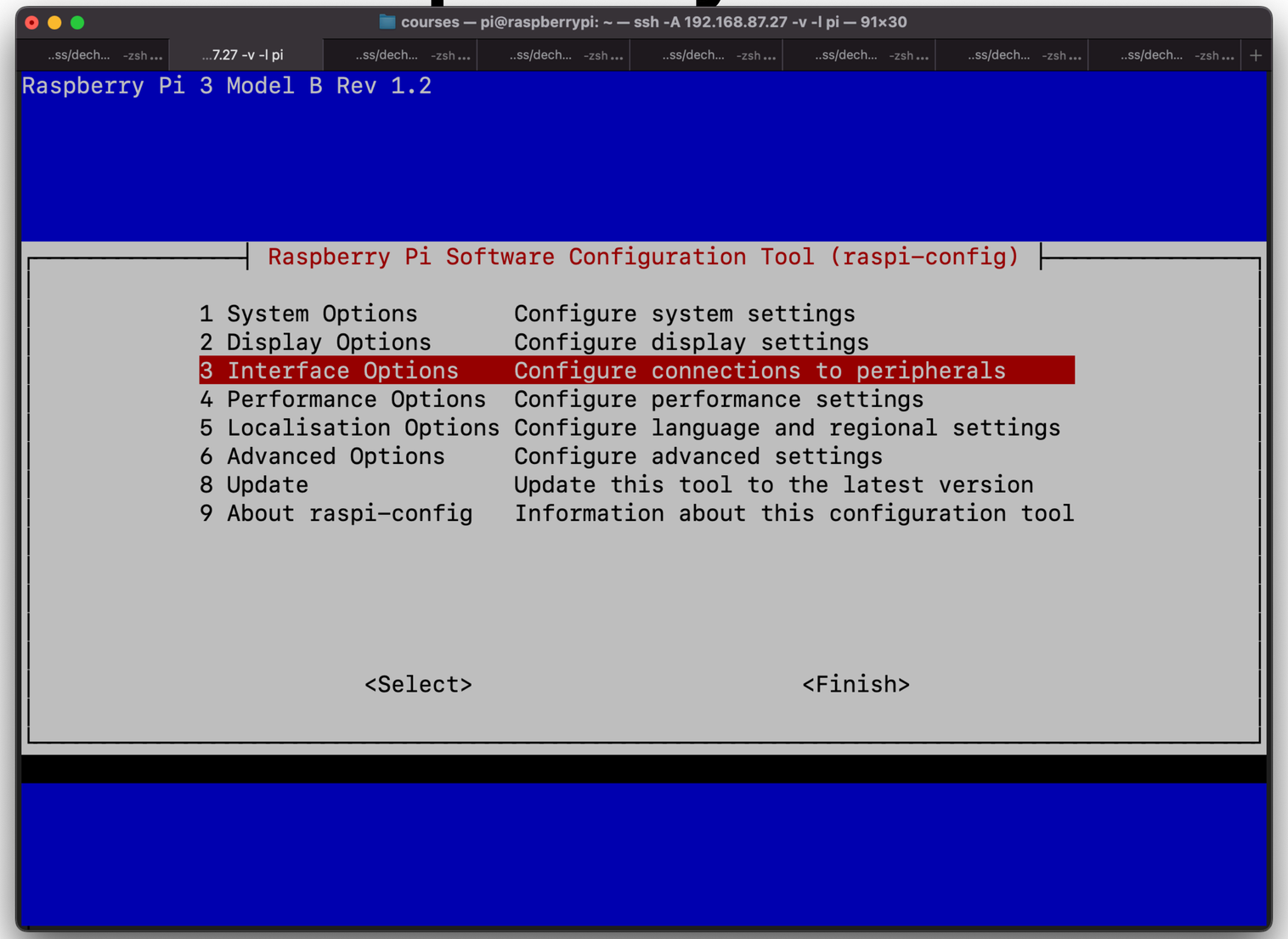
```
@Throws(IOException::class)
fun writeUartData(uart: UartDevice) {
  val count = uart.run {
    ByteArray(...).let { buffer ->
      write(buffer, buffer.size)
    }
  }
  Log.d(TAG, "Wrote $count bytes to peripheral")
}
```

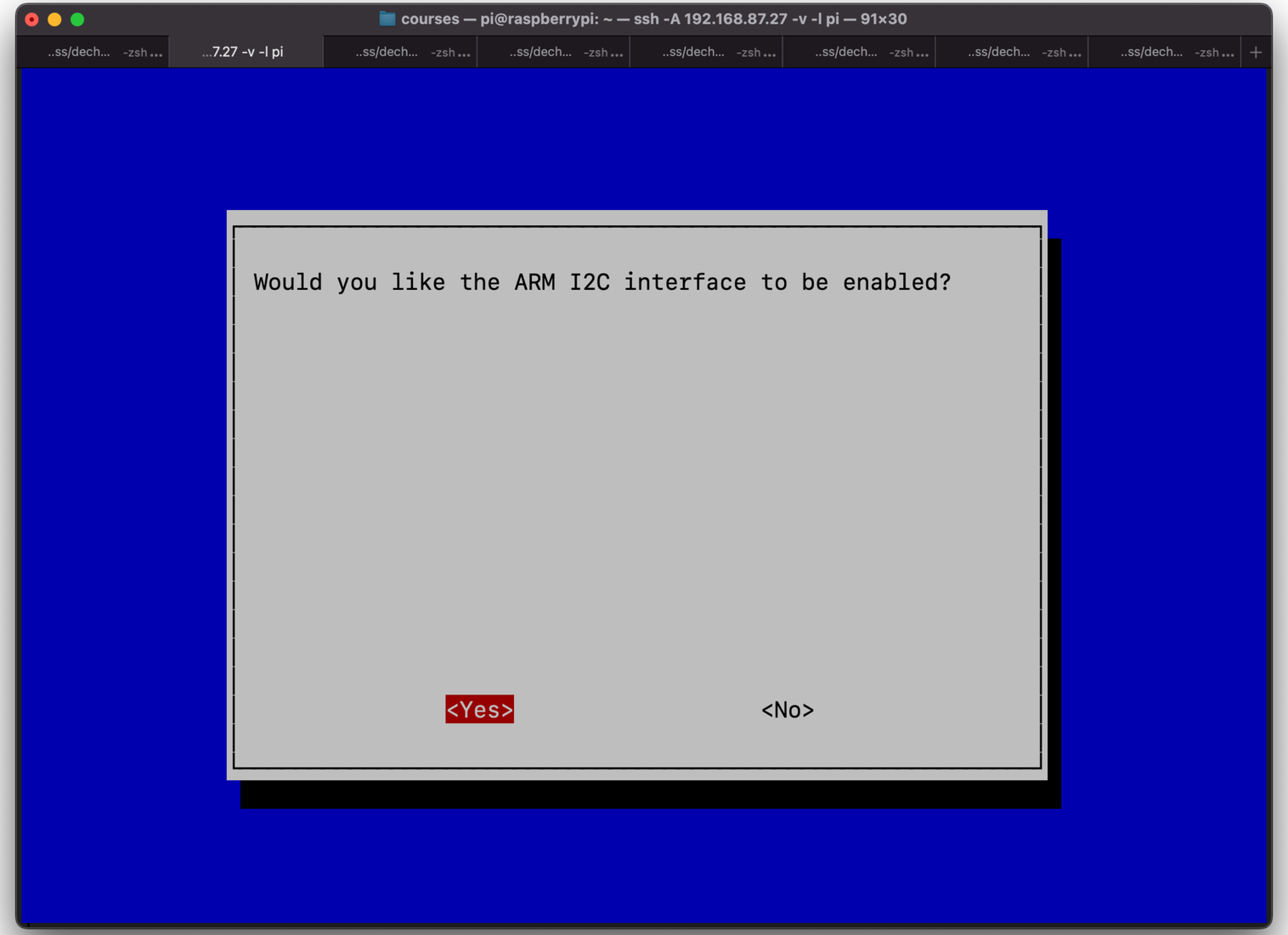
Listening for incoming data

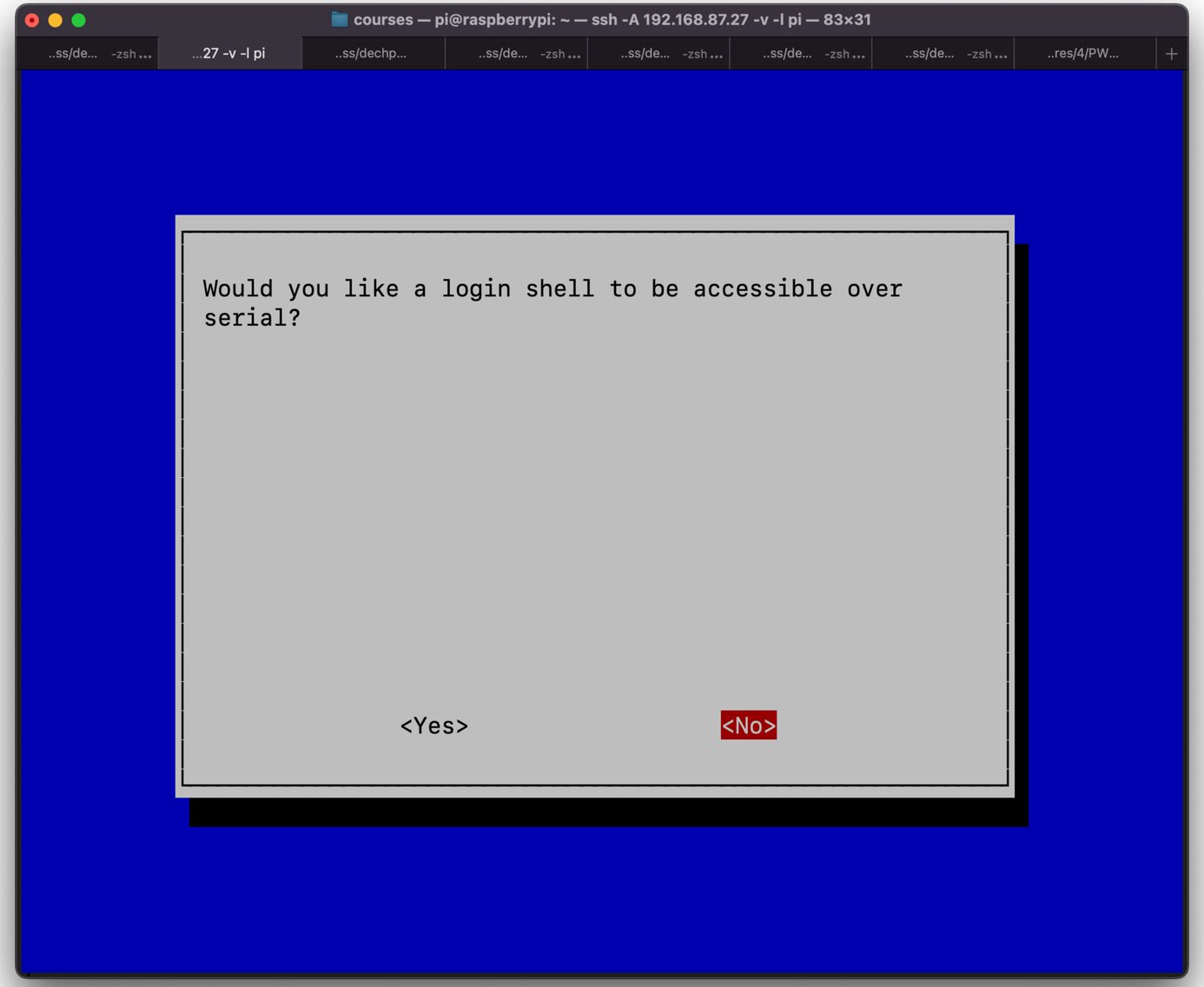
```
class HomeActivity : Activity() {
  private var mDevice: UartDevice? = null
  ...
  override fun onStart() {
    super.onStart()
    // Begin listening for interrupt events
    mDevice?.registerUartDeviceCallback(uartCallback)
  }
  override fun onStop() {
    super.onStop()
    // Interrupt events no longer necessary
    mDevice?.unregisterUartDeviceCallback(uartCallback)
  }
  private val uartCallback = object : UartDeviceCallback {
    override fun onUartDeviceDataAvailable(uart: UartDevice): Boolean {
      // Read available data from the UART device
      try {
        readUartBuffer(uart)
      } catch (e: IOException) {
        Log.w(TAG, "Unable to access UART device", e)
      }
      // Continue listening for more interrupts
      return true
    }
    override fun onUartDeviceError(uart: UartDevice?, error: Int) {
      Log.w(TAG, "$uart: Error event $error")
    }
  }
}
```

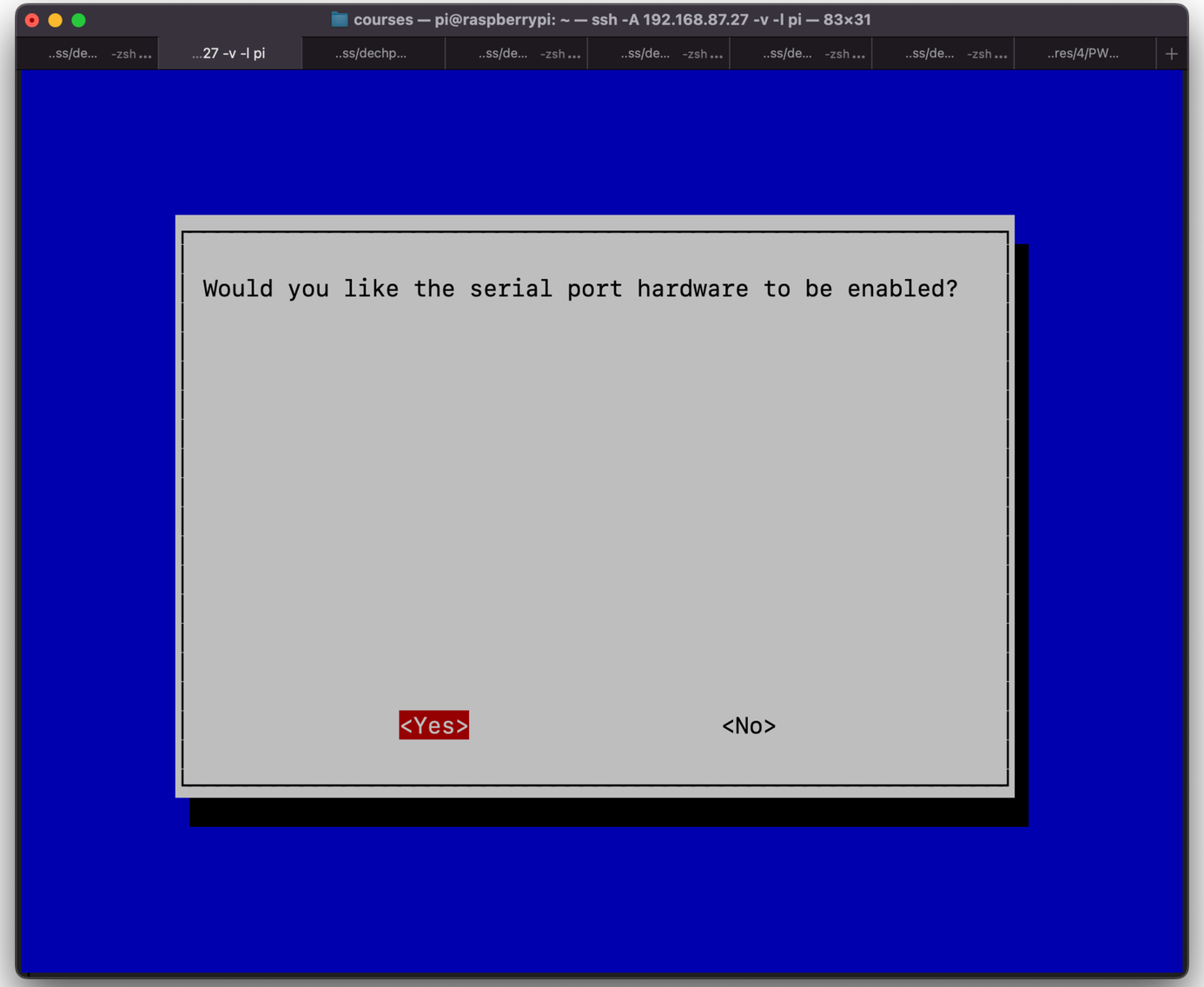
```
@Throws(IOException::class)
fun readUartBuffer(uart: UartDevice) {
    // Maximum amount of data to read at one time
    val maxCount = ...
    uart.apply {
        ByteArray(maxCount).also { buffer ->
            var count: Int = read(buffer, buffer.size)
            while (count > 0) {
                Log.d(TAG, "Read $count bytes from peripheral")
                count = read(buffer, buffer.size)
            }
        }
    }
}
```

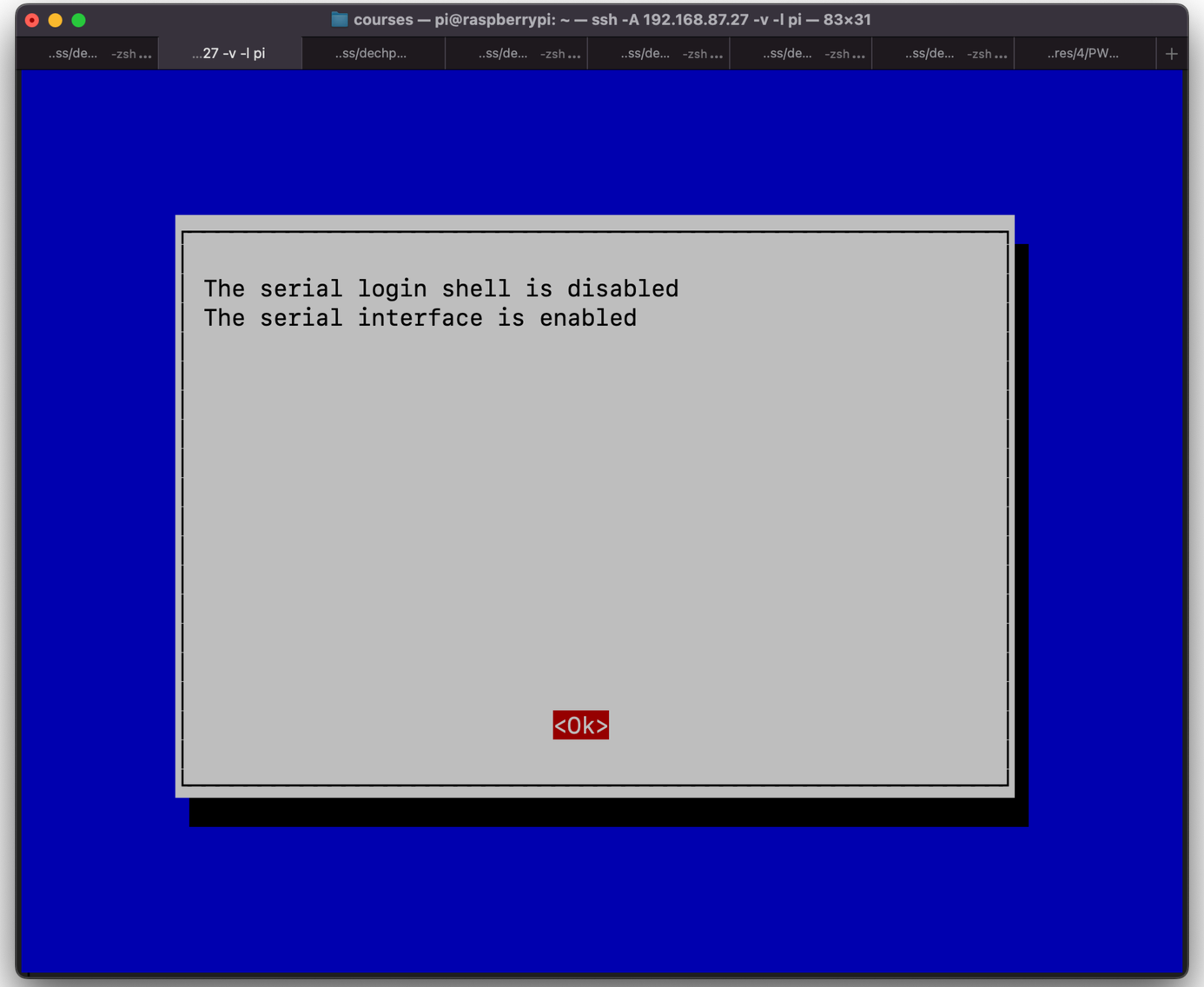
Enable UART on Raspberry Pi OS

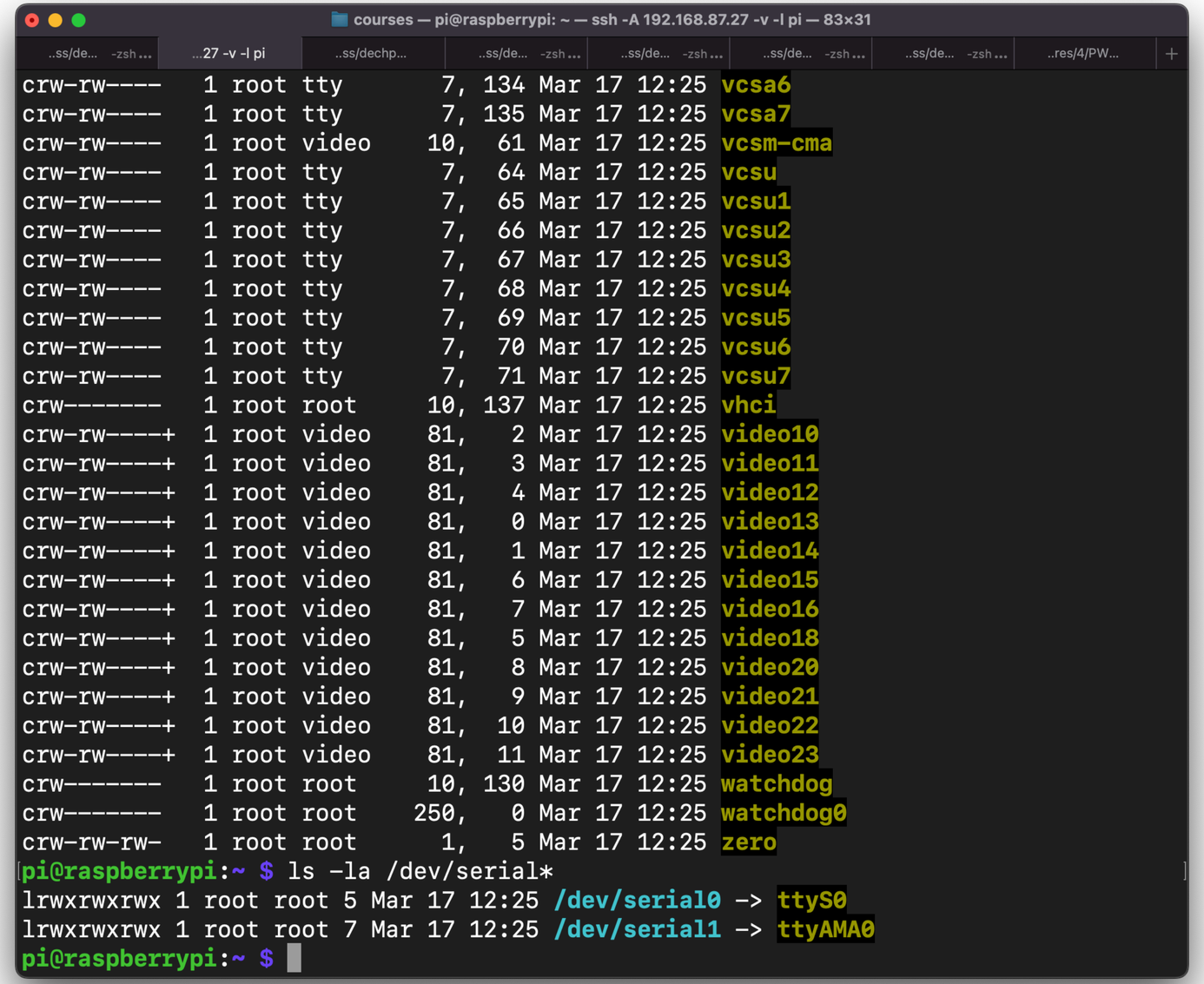



courses — pi@raspberrypi: ~ — ssh -A 192.168.87.27 -v -l pi — 91×30
..ss/dech... -zsh ...
...7.27 -v -l pi
..ss/dech... -zsh ...
..ss/dech... -zsh ...
..ss/dech... -zsh ...
..ss/dech... -zsh ...
..ss/dech... -zsh ...
..ss/dech... -zsh ...
Would you like the ARM I2C interface to be enabled?
<Yes>
<No>


courses — pi@raspberrypi: ~ — ssh -A 192.168.87.27 -v -l pi — 83×31
..ss/de... -zsh
...27 -v -l pi
..ss/dechp...
..ss/de... -zsh
..ss/de... -zsh
..ss/de... -zsh
..ss/de... -zsh
..res/4/PW...
Would you like a login shell to be accessible over serial?
<Yes>
<No>


courses — pi@raspberrypi: ~ — ssh -A 192.168.87.27 -v -l pi — 83×31
..ss/de... -zsh
...27 -v -l pi
..ss/dechp...
..ss/de... -zsh
..ss/de... -zsh
..ss/de... -zsh
..ss/de... -zsh
..res/4/PW...
Would you like the serial port hardware to be enabled?
<Yes>
<No>


courses — pi@raspberrypi: ~ — ssh -A 192.168.87.27 -v -l pi — 83×31
..ss/de... -zsh
...27 -v -l pi
..ss/dechp...
..ss/de... -zsh
..ss/de... -zsh
..ss/de... -zsh
..ss/de... -zsh
..res/4/PW...
The serial login shell is disabled
The serial interface is enabled
<Ok>


courses — pi@raspberrypi: ~ — ssh -A 192.168.87.27 -v -l pi — 83×31
crw-rw---- 1 root tty 7, 134 Mar 17 12:25 vcsa6
crw-rw---- 1 root tty 7, 135 Mar 17 12:25 vcsa7
crw-rw---- 1 root video 10, 61 Mar 17 12:25 vcsm-cma
crw-rw---- 1 root tty 7, 64 Mar 17 12:25 vcsu
crw-rw---- 1 root tty 7, 65 Mar 17 12:25 vcsu1
crw-rw---- 1 root tty 7, 66 Mar 17 12:25 vcsu2
crw-rw---- 1 root tty 7, 67 Mar 17 12:25 vcsu3
crw-rw---- 1 root tty 7, 68 Mar 17 12:25 vcsu4
crw-rw---- 1 root tty 7, 69 Mar 17 12:25 vcsu5
crw-rw---- 1 root tty 7, 70 Mar 17 12:25 vcsu6
crw-rw---- 1 root tty 7, 71 Mar 17 12:25 vcsu7
crw------- 1 root root 10, 137 Mar 17 12:25 vhci
crw-rw----+ 1 root video 81, 2 Mar 17 12:25 video10
crw-rw----+ 1 root video 81, 3 Mar 17 12:25 video11
crw-rw----+ 1 root video 81, 4 Mar 17 12:25 video12
crw-rw----+ 1 root video 81, 0 Mar 17 12:25 video13
crw-rw----+ 1 root video 81, 1 Mar 17 12:25 video14
crw-rw----+ 1 root video 81, 6 Mar 17 12:25 video15
crw-rw----+ 1 root video 81, 7 Mar 17 12:25 video16
crw-rw----+ 1 root video 81, 5 Mar 17 12:25 video18
crw-rw----+ 1 root video 81, 8 Mar 17 12:25 video20
crw-rw----+ 1 root video 81, 9 Mar 17 12:25 video21
crw-rw----+ 1 root video 81, 10 Mar 17 12:25 video22
crw-rw----+ 1 root video 81, 11 Mar 17 12:25 video23
crw------- 1 root root 10, 130 Mar 17 12:25 watchdog
crw------- 1 root root 250, 0 Mar 17 12:25 watchdog0
crw-rw-rw- 1 root root 1, 5 Mar 17 12:25 zero
pi@raspberrypi:~ $ ls -la /dev/serial*
lrwxrwxrwx 1 root root 5 Mar 17 12:25 /dev/serial0 -> ttyS0
lrwxrwxrwx 1 root root 7 Mar 17 12:25 /dev/serial1 -> ttyAMA0
pi@raspberrypi:~ $

Hello World

```
import serial

# Initialize the UART connection
ser = serial.Serial('/dev/ttyS0', baudrate=9600, timeout=1)

# Send a command to the device
ser.write(b'Hello, World!')

# Wait for the device to respond and print the response
response = ser.readline()
print(response.decode('utf-8'))

# Close the UART connection
ser.close()
```

Hello World

```
use std::io::{Read, Write};
use std::fs::OpenOptions;
use std::time::Duration;

fn main() {
    // Open the serial port
    let mut serial_port = OpenOptions::new()
        .read(true)
        .write(true)
        .open("/dev/ttyS0")
        .unwrap();

    // Configure the serial port settings
    serial_port.reconfigure(&|settings| {
        settings.set_baud_rate(serial::Baud9600)?;
        settings.set_char_size(serial::Bits8);
        settings.set_parity(serial::ParityNone);
        settings.set_stop_bits(serial::Stop1);
        Ok(())
    }).unwrap();
```

```rust
use std::io::{Read, Write};
use std::fs::OpenOptions;
use std::time::Duration;

fn main() {
    // Open the serial port
    let mut serial_port = OpenOptions::new()
        .read(true)
        .write(true)
        .open("/dev/ttyS0")
        .unwrap();

    // Configure the serial port settings
    serial_port.reconfigure(&|settings| {
        settings.set_baud_rate(serial::Baud9600)?;
        settings.set_char_size(serial::Bits8);
        settings.set_parity(serial::ParityNone);
        settings.set_stop_bits(serial::Stop1);
        Ok(())
    }).unwrap();

    // Send a command to the device
    let command = "Hello, World!\n";
    serial_port.write_all(command.as_bytes()).unwrap();

    // Wait for the device to respond and print the response
```

```
    let mut serial_port = OpenOptions::new()
        .read(true)
        .write(true)
        .open("/dev/ttyS0")
        .unwrap();

    // Configure the serial port settings
    serial_port.reconfigure(&|settings| {
        settings.set_baud_rate(serial::Baud9600)?;
        settings.set_char_size(serial::Bits8);
        settings.set_parity(serial::ParityNone);
        settings.set_stop_bits(serial::Stop1);
        Ok(())
    }).unwrap();

    // Send a command to the device
    let command = "Hello, World!\n";
    serial_port.write_all(command.as_bytes()).unwrap();

    // Wait for the device to respond and print the response
    let mut buffer = String::new();
    serial_port.read_to_string(&mut buffer).unwrap();
    println!("{}", buffer);

    // Close the serial port
    drop(serial_port);
}
```

Lecture outcomes

- Understand PWM and I2C.
- Transfer data using UART.