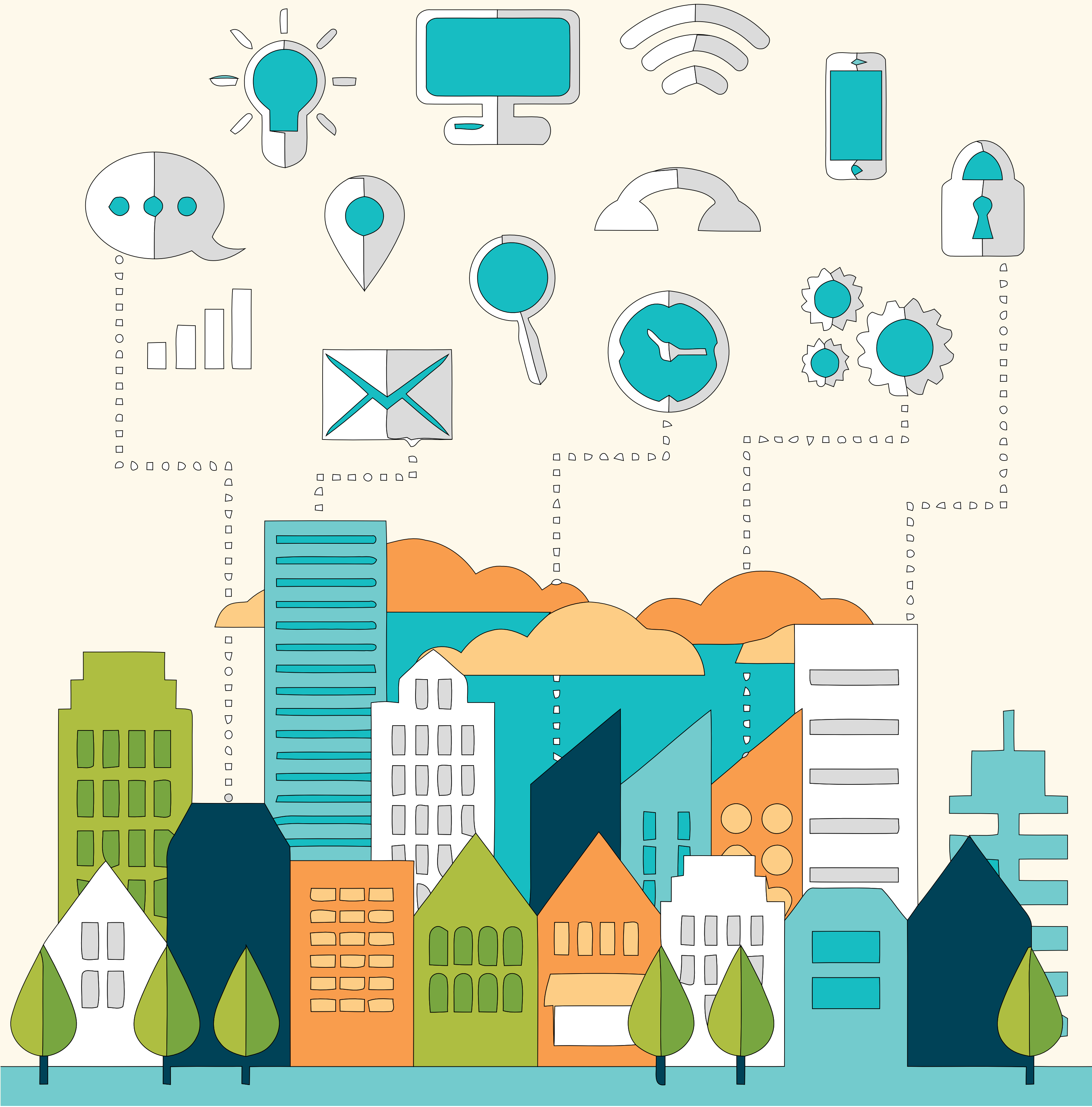
Charith Perera (Eds.)
PhD, MBA

Internet of Things

Systems Design

Lab Book

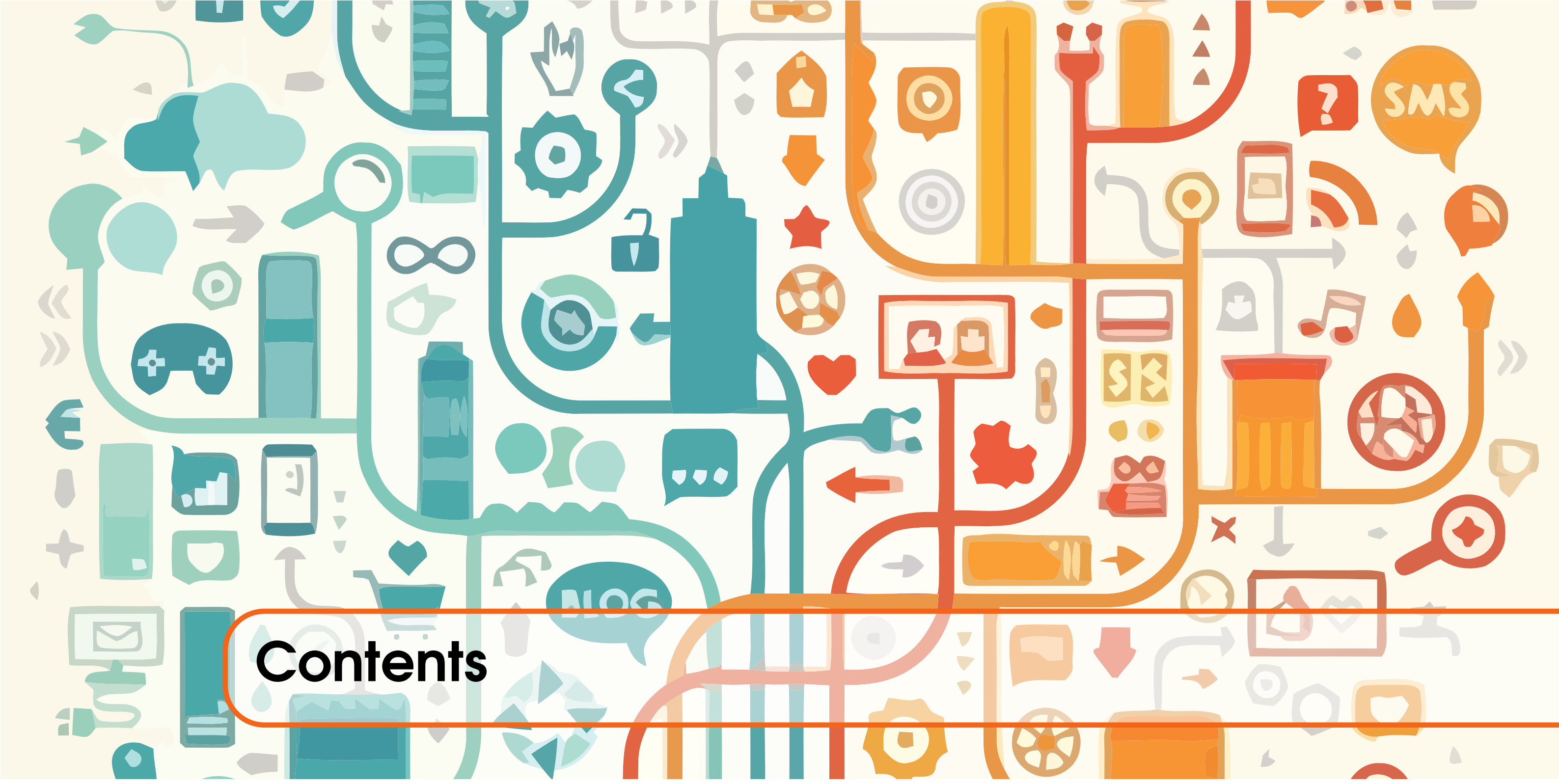


PUBLISHED BY IOT GARAGE

How to cite this book
Charith Perera (Eds.), Internet of Things: Systems Design Lab Book, IOT Garage, 2023

Contributing Authors (alphabetical order)
Hakan Kayan, Michal Malecki, Yasar Majib

Version 2.2 (Latest), July 2024
Version 2.1, March 2024
Version 2.0, February 2024
Version 1.9, January 2024
Version 1.8, November 2023
Version 1.7, September 2023
Version 1.6, March 2023
Version 1.5, September 2022
Version 1.4, March 2022
Version 1.3, February 2022
Version 1.2, January 2022
Version 1.1, January 2021
Version 1.0, January 2020



Contents

Preface 5
IoT Kit for BSc and MSc Labs and Coursework 6
Accessing the Code Repository 14
IoT Extension Pack for MSc Only Labs 15
Additional Components for BSc and MSc Coursework 18
Hardware Requirement Summary 20

1 Micro-Controller Programming ● 21
2 Single-board Computer Programming ● 33
3 Posting Data to an IoT Cloud Platform ● 49
4 Connecting an IoT Gateway to an IoT Cloud ● 61
5 Connecting a Sensor Node to IoT Gateway ● 71
6 End to End Full Stack IoT Development ● 83
7 Introduction to Wireshark on Raspberry Pi ● 91
8 Programming Arduino with Blockly ● 103
9 Programming Raspberry Pi with Python ● 115
10 Bluetooth Low Energy (BLE) Based Systems ● 127

11 RFID and NFC Based Tracking ● 139

12 Multimedia Communication ● 149

13 Microcontroller Programming Simulator ● 157

14 Advance Sensors, Actuators, Components ● 169

15 3D Objects Designing and Printing ● 181

16 Getting Started with Raspberry Pi Camera ● 191

17 Debugging the Raspberry Pi 203

Debugging the Raspberry Pi 203

Preface

This IOT LAB BOOK is primarily compiled to support the university courses on *'Internet of Things: Systems Design'* at both undergraduate and postgraduate levels. If you are taking either of these modules, please make sure you follow this lab book. It is compulsory to complete labs marked ● for both undergraduate and postgraduate level modules. The labs marked ● are compulsory for the postgraduate level.

This lab book guides you through a series of labs. Each lab has its objectives. It is expected that you should be able to complete each lab session within two hours (most of the time, much less). This lab book does assume that you have some amount of networking knowledge. Further, it is important to mention that IoT, by nature, is a broad subject. Therefore, in a few lab sessions, we cannot teach you all the topics in-depth. For example, Arduino programming use C/C++ programming languages. However, we do not expect you to be an expert on C/C++ to follow the lab session. However, if you have some background, you will find some known concepts in action and feel comfortable. If you have never seen C/C++ before, you will, of course, feel nervous and sometimes will feel lost.

Throughout the lab book, we provided explanations, external links, and references to reading material. Especially if you do not understand certain programming tasks such as C/C++ it may be worth reading those links. Further, these links will guide you to explore the universe of IoT on your own, beyond the labs we have provided here. Finally, we want to emphasise that this is not a programming course. Therefore, we do not try to teach you a particular programming language (though we try to provide as many links and references to develop your skills). It is up to you to develop the gaps in your knowledge by referring to the links we provided.

— Further information, links and references. We will use multiple dedicated structures throughout this lab book to offer different types of additional knowledge, including areas for deeper thought, useful hints, and practical tips. These are here to guide your thinking and support your learning, but you're welcome to skip them if you prefer.

IoT Kit for BSc and MSc Labs and Coursework

In order to follow all the labs in this lab book, you need the following components. The component requirement for each lab is presented at the beginning of each chapter. You can also see the lab numbers that we use for the items on figure captions.

Raspberry Pi The Raspberry Pi is a series of small single-board computers. We have tested the labs with both Raspberry Pi3B+ and Raspberry Pi 4.

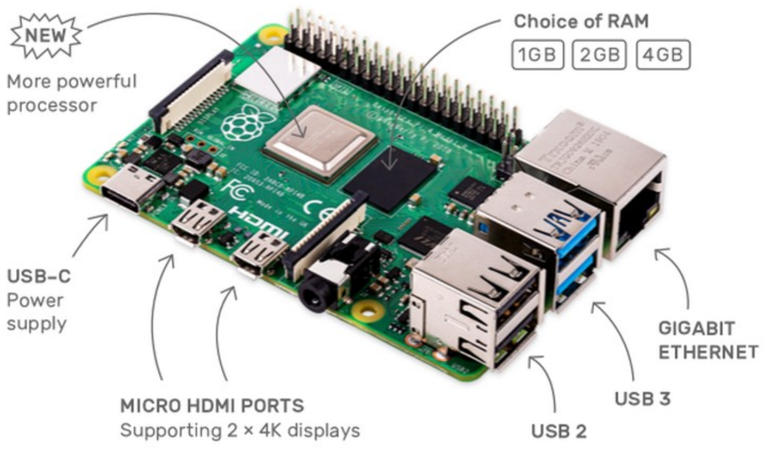


Figure 1: Raspberry Pi - Labs (2, 3, 4, 5, 6)

Notes We have provided the Raspberry Pi 4 version with 4GB RAM
- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- 1GB, 2GB or 4GB LPDDR4-3200 SDRAM (depending on model)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
- Gigabit Ethernet
- 2 USB 3.0 ports; 2 USB 2.0 ports.
- Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
- 2 x micro-HDMI ports (up to 4kp60 supported)
- 2-lane MIPI DSI display port
- 2-lane MIPI CSI camera port
- 4-pole stereo audio and composite video port
- H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
- OpenGL ES 3.0 graphics
- Micro-SD card slot for loading operating system and data storage
- 5V DC via USB-C connector (minimum 3A*)
- 5V DC via GPIO header (minimum 3A*)
- Power over Ethernet (PoE) enabled (requires separate PoE HAT)
- Operating temperature: 0 – 50 degrees C ambient

GrovePi Plus GrovePi+ is an add-on board with 15 Grove 4-pin interfaces that brings Grove sensors to the Raspberry Pi. GrovePi+ is an easy-to-use and modular system for hardware hacking with the Raspberry Pi, no need for soldering or breadboards: plug in your Grove sensors and start programming directly. Grove is an easy-to-use collection of more than 100 inexpensive plug-and-play modules that sense and control the physical world. Connecting Grove Sensors to Raspberry Pi empowers your Pi in the physical world. With hundreds of sensors to choose from, Grove families, the possibilities for interaction are endless.

— Grove Ecosystem. Most of the Grove components can be connected to Raspberry Pi through GrovePi+ and can be connected Arduino micro-controller through Base Shield for Arduino. Most components are interchangeable between two platforms (i.e.,g Raspberry Pi and Arduino).

- **Sensors:** `http://wiki.seeedstudio.com/Sensor/`
- **Actuators:** `http://wiki.seeedstudio.com/Actuator/`
- **Displays:** `http://wiki.seeedstudio.com/Display/`
- **Communications:** `http://wiki.seeedstudio.com/Communication/`

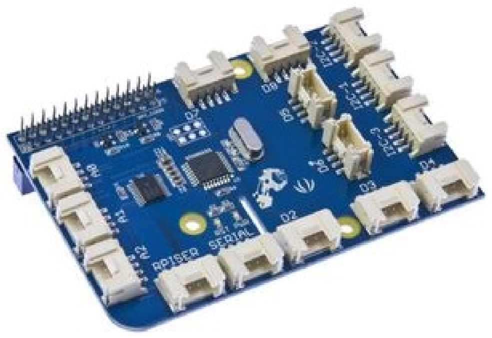


Figure 2: GrovePi+ - Labs (2, 3, 4, 6)

Arduino Expansion Shield for Raspberry Pi B+ (V2.0) With the Arduino Adapter For Raspberry Pi, there's a way for the Raspberry Pi GPIO interface to adapt to Arduino pinouts, it is now possible to use the Pi together with vast Arduino shields and hardware/software resources: (i) Raspberry Pi GPIO interface to adapt to Arduino pinout, (ii) Compatible with Arduino UNO, Leonardo, easy to connect with various Arduino shields, (iii) XBee connector for connecting various XBee modules, and (iv) Sensor interface for connecting various sensors.

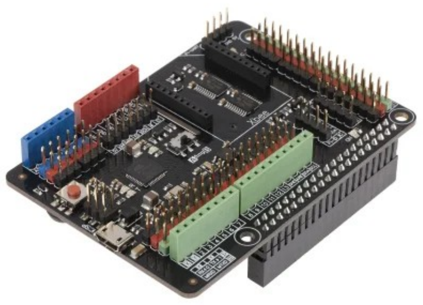


Figure 3: Arduino - Labs (1, 5, 6)

Base Shield for Arduino Arduino Uno is the most popular Arduino board so far; however, it is sometimes frustrating when your project requires many sensors or LEDs, and your jumper wires are in a mess. The Base Shield's purpose is to help you eliminate breadboard and jumper wires. With the rich grove connectors on the base board, you can conveniently add all the grove modules to the Arduino Uno! The pinout of Base Shield V2 is the same as Arduino Uno R3.

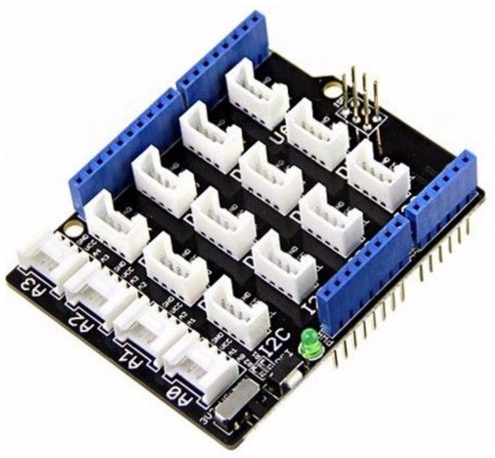


Figure 4: Base Shield - Labs (1, 5, 6)

Loudness (Sound) Sensor Sound Sensors can detect the sound intensity of the environment. The main component of the module is a simple microphone based on the LM386 amplifier and an electret microphone. This module's output is analogue and can be easily sampled and tested by a Seeeduino.

Figure 5: Loudness Sensor - Labs (6)

Light Sensor The light sensor integrates a photo-resistor (light-dependent resistor) to detect light intensity. The resistance of the photo-resistor decreases when the intensity of light increases. A dual OpAmp chip LM358 on board produces a voltage corresponding to light intensity (i.e. based on resistance value). The output signal is an analogue value. The brighter the light is, the larger the value. This module can be used to build a light-controlled switch, i.e. switch off lights during daytime and switch on lights during nighttime.

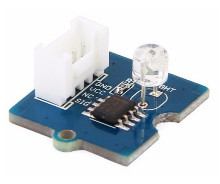


Figure 6: Light Sensor - Labs (1, 6)

PIR Motion Sensor This sensor allows you to sense motion, usually human movement in its range.

Simply connect it to Grove-Base shield and program it. When anyone moves in its detecting range, the sensor will output HIGH on its SIG (signal) pin.

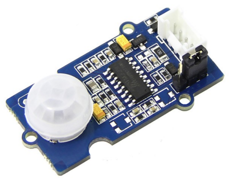
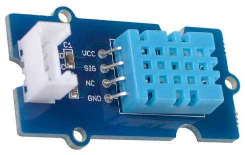
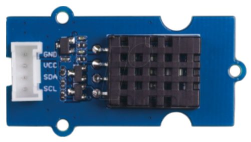


Figure 7: PIR - Labs (1, 5, 6)

Temperature and Humidity This is a powerful sister version of our Grove - Temperature and Humidity Sensor Pro in figure 8a. It has a complete and more accurate performance than the basic version. The detecting range of this sensor is 5% RH - 99% RH, and -40C-80C. And its accuracy reaches up to 2% RH and 0.5C. A professional choice for applications that have relatively strict requirements.
The new Grove - Temperature & Humidity Sensor in figure 8b is based on the DHT20 sensor. The DHT20 is an upgraded version of the DHT11, compared with the previous version, the temperature and humidity measurement accuracy are higher, and the measurement range is larger. It features I2C output which means it is easier to use.



(a) DHT11



(b) DHT20

Figure 8: Temperature & Humidity - Labs (2, 3)

Servo Motor Servo is a DC motor with a gearing and feedback system. It is used in the driving mechanism of robots. The module is a bonus product for Grove lovers. We regulated the three-wire servo into a Grove standard connector. You can plug and play it as a typical Grove module now, without jumper wires clutter.

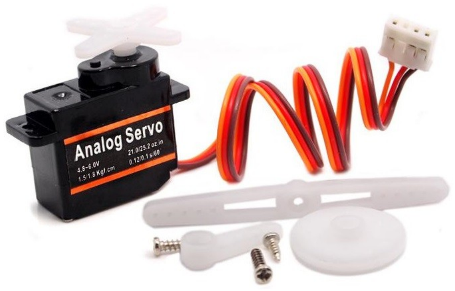


Figure 9: Servo Motor - Labs (1, 6)

LED Button LED Button is composed of Grove-Yellow Button, Grove-Blue LED Button and Grove-Red LED Button. This button is stable and reliable with a 100 000 times long life. With the built-in LED, you can apply it to many interesting projects. It is really useful to use the LED to show the status of the button. We use a high-quality N-Channel MOSFET to control the LED, ensuring high switching speed and low consumption.

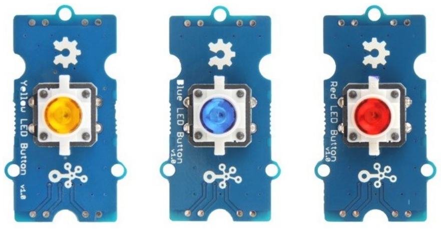


Figure 10: LED Button - Labs (2)

Ultrasonic Ranger The Ultrasonic Distance Sensor is an ultrasonic transducer that utilizes ultrasonic waves to measure distance. It can measure from 3cm to 350cm with an accuracy of up to 2mm. It is a perfect ultrasonic module for distance measurement, proximity sensors, and ultrasonic detectors. This module has an ultrasonic transmitter and an ultrasonic receiver, so you can consider it an ultrasonic transceiver. Familiar with sonar, when the 40KHz ultrasonic wave generated by the transmitter encounters the object, the sound wave will be emitted back, and the receiver can receive the reflected ultrasonic wave. It is only necessary to calculate the time from the transmission to the reception and then multiply the speed of the sound in the air (340 m/s) to calculate the distance from the sensor to the object.

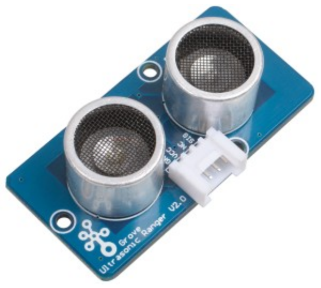


Figure 11: Ultrasonic Ranger - Labs (2, 6)

Buzzer The buzzer module has a piezo buzzer as the main component. The piezo can be connected to digital outputs and emit a tone when the output is HIGH. Alternatively, it can be connected to an analogue pulse-width modulation output to generate various tones and effects.

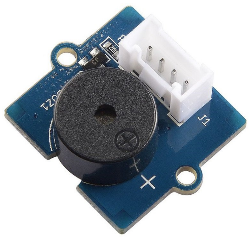
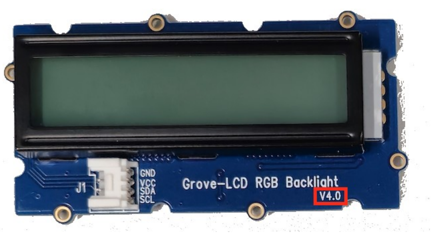
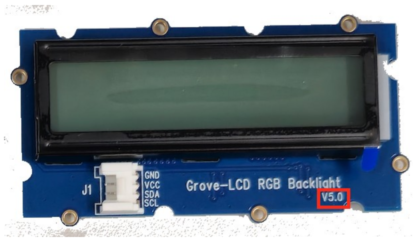


Figure 12: Buzzer - Labs (2, 3, 4, 6)

LCD RGB Backlight Display This Grove enables you to set the colour to whatever you like via the simple and concise Grove interface. It takes I2C as a communication method with your microcontroller. So the number of pins required for data exchange and backlight control shrinks from 10 to 2, relieving IOs for other challenging tasks. Besides, Grove - LCD RGB Backlight supports user-defined characters. Want to get a love heart or some other foreign characters? Just take advantage of this feature and design it.



(a) LCD version 4.0



(b) LCD verion 5.0

Figure 13: LCD - Labs (2, 6)

Serial Bluetooth v3.0 Serial Bluetooth is an easy-to-use module compatible with the existing Grove Base Shield and designed for a transparent wireless serial connection setup. The serial port Bluetooth module is fully qualified Bluetooth V2.0+EDR(Enhanced Data Rate) 2Mbps Modulation with a complete 2.4GHz radio transceiver and baseband. It uses a CSR Bluecore 04-External single chip Bluetooth system with CMOS technology and AFH(Adaptive Frequency Hopping Feature). It has the smallest footprint of 12.7mm x 27mm. Hope it will simplify your overall design/development cycle.

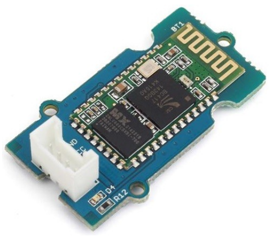


Figure 14: Serial Bluetooth - Labs (5)

Raspberry Pi Camera v2 The Raspberry Pi Camera Module 2 can take hide definition videos and photographs. It allows connecting attachments such as lenses. While we can attach different cameras to Raspberry Pi, if case is not very specific, this camera should be preferred as there are libraries already available. We can do live image processing on Raspberry Pi via this camera as well.

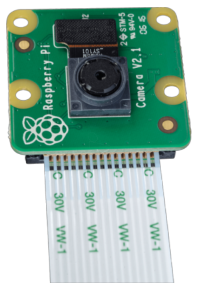


Figure 15: Camera - Labs (16)

SD Card Sandisk Class 10 Micro SD card reinstalled with the Raspberry Pi operating system.

Figure 16: SD Card - Labs (2, 3, 4, 5, 6)

Micro HDMI to HDMI Cable Connect a device that has a micro HDMI port to an HDMI compatible TV or monitor to share music, video or images up to 1080p resolution. Connect Monitor to Raspberry Pi.

Figure 17: Cable - Labs (2, 3, 4, 5, 6)

USB Keyboard and Mouse The keyboard has three in-built USB 2.0 type-A ports for powering other peripherals (such as the official Raspberry Pi mouse).

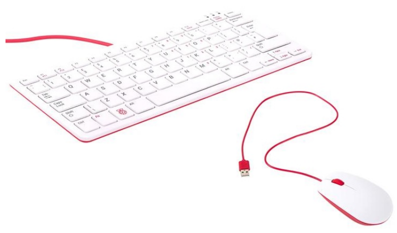
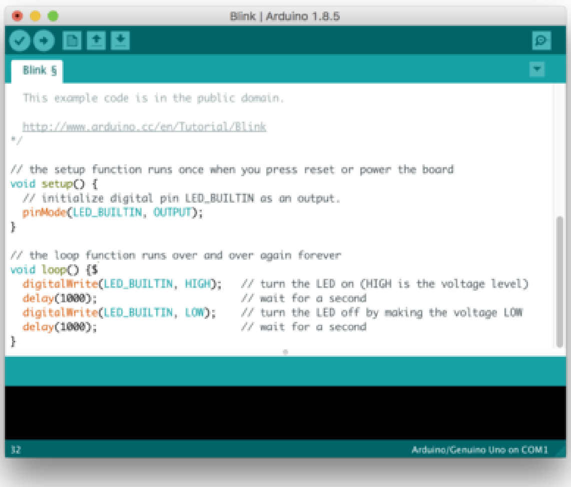


Figure 18: Keyboard - Labs (2, 3, 4, 5, 6)

Software Requirements

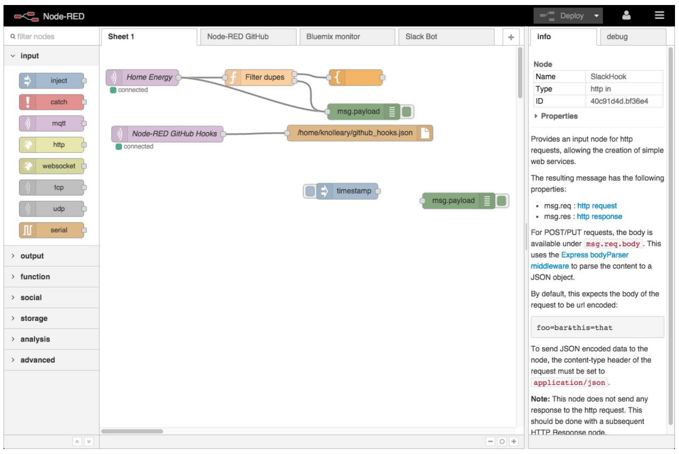
Arduino IDE The Arduino integrated development environment (IDE) is a cross-platform application (for Windows, macOS, and Linux) that is written in the programming language Java. It is used to write and upload programs to Arduino-compatible boards and, with the help of 3rd party cores, other vendor development boards (`https://www.arduino.cc/en/main/software`).



```
  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {$
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

Figure 19: Arduino - Labs (1, 5, 6)

Node-RED Node-RED is an open source flow-based development tool for visual programming developed originally by IBM for wiring together hardware devices, APIs and online services as part of the Internet of Things. Node-RED provides a web browser-based flow editor, which can be used to create JavaScript functions. Elements of applications can be saved or shared for re-use. The runtime is built on Node.js. The flows created in Node-RED are stored using JSON. Node-RED can be used both on edge on in the cloud. We will use Node-RED as the edge IoT platform in our labs. (`https://nodered.org/`).



Node

Name	SlackHook
Type	http in
ID	40c91d4d.bf36e4

▸ Properties

Provides an input node for http requests, allowing the creation of simple web services.

The resulting message has the following properties:

- msg.req : http request
- msg.res : http response

For POST/PUT requests, the body is available under `msg.req.body`. This uses the Express bodyParser middleware to parse the content to a JSON object.

By default, this expects the body of the request to be url encoded:

```
foo=bar&this=that
```

To send JSON encoded data to the node, the content-type header of the request must be set to `application/json`.

Note: This node does not send any response to the http request. This should be done with a subsequent HTTP Response node.

Figure 20: Node-RED - Labs (2, 3, 4, 5, 6)

Thingsboard ThingsBoard is an open-source IoT platform for device management, data collection, processing and visualization for your IoT projects. ThingsBoard can be used both on edge on in the cloud. In our labs, we will be using Thingsboard as the cloud IoT platform (`https://thingsboard.io/`).

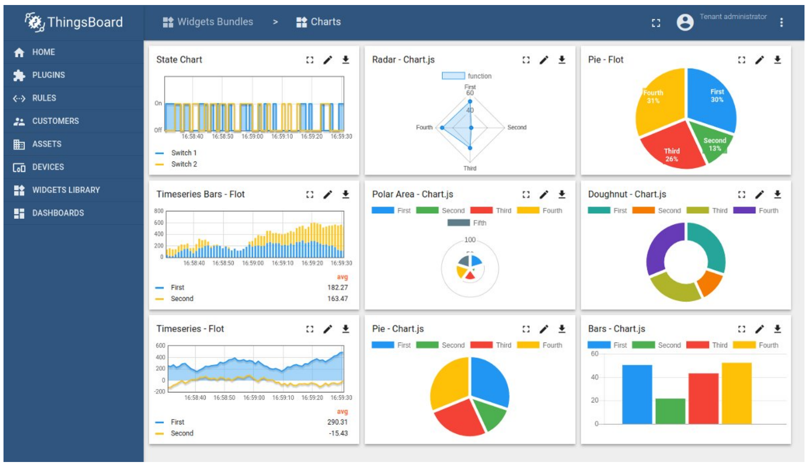


Figure 21: Thingsboard - Labs (3, 4, 6)

Monitor (To Work From Home) You will need an external monitor to connect your Raspberry Pi to complete the labs. If you are an experienced user, you can SSH into your Pi and do the labs. According to your monitor's input, you may need a micro-HDMI to HDMI, DVI, or VGA (for older monitors) cable.

Accessing the Code Repository

All the code segments required to complete the labs in this *IOT LAB BOOK* can be found in the following GitLab repository:

`https://gitlab.com/IOTGarage/iot-lab-book`

This repository contains scripts, sample code, and additional resources referenced throughout the labs. When working through any of the lab exercises, please refer to the respective chapter's folder or file within the repository to locate the matching code examples. Any updates, bug fixes, or enhancements will also be made available in this repository, so be sure to periodically check it for the latest version of the code. By visiting the repository, you can:

- **Clone or Download the Code:** Pull down all relevant examples, scripts, and configurations.
- **Review Commit History:** Explore how the code has evolved, examining different versions and branches that may contain experimental features.
- **Submit Issues:** If you encounter any bugs or have questions, open an issue and collaborate with the community for support or improvements.

IoT Extension Pack for MSc Only Labs

LED Bar LED Bar is a 10-segment LED gauge bar with a built-in LED controlling chip. We use LED bars when we want to demonstrate a level of something. For example, we can show the remaining battery capacity, temperature level, distance, sound level etc. The bar colours range from red to green, while red indicates the highest level.

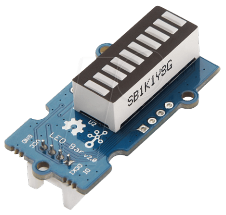


Figure 22: LED bar - Labs (8, 9, 14)

Rotary Angle Sensor Rotary Angle Sensor v1.2 is a potentiometer. We can set the resistance up to 10k Ohm. Thus, according to the input voltage potentiometer will generate an output. We can use potentiometers when we want to divide voltage to a certain degree. Thus we can use sensors that require 5V and 3.3V within the same setup.

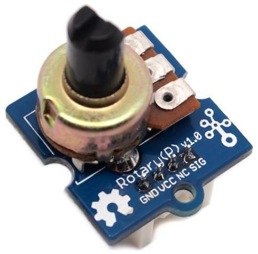


Figure 23: Rotary Angle - Labs (8)

EMG Sensor We use electromyography sensors (EMG) to measure signals generated by our muscles. They are useful for detecting any behavioural anomalies in our muscles. Also, they are used to model the muscle behaviour of professionals such as athletes and footballers.

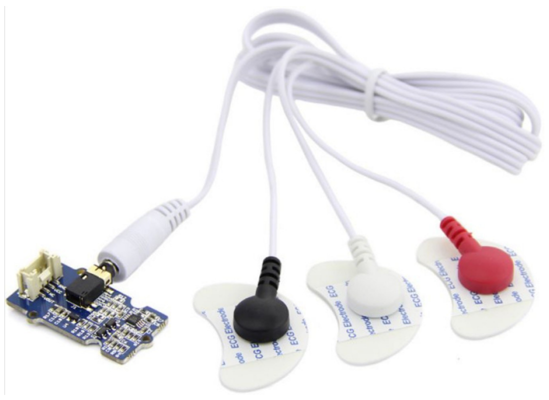


Figure 24: EMG - Labs (14)

Speaker Grove Speaker has a built-in potentiometer where we can set the sound level thus it offers power amplification. We can also set the frequency to generate a different tone. Speaker are useful when building alarm systems or distance indicators.

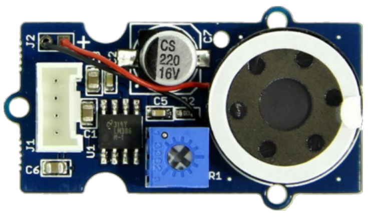


Figure 25: Speaker - Labs (12)

LED Socket Kit / LED Grove - LED Socket Kit allows us to control brightness of the LEDs via the built-in potentiometer. We might like to set the LED brightness in several cases such as reading a book, driving a car, or using a phone. Light-emitting diodes (**LED**) are one of the most efficient light sources. We currently have them in our phones, TVs, monitors, and simply anywhere we have a screen.

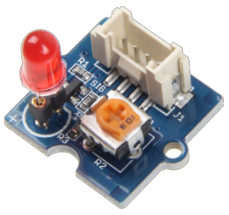


Figure 26: LED Socket Kit - Labs (10, 11)

RFID Reader RFID Reader is a module used to read uem4100 RFID card information with two output formats: Uart and Wiegand. It has a sensitivity with maximum 7cm sensing distance.

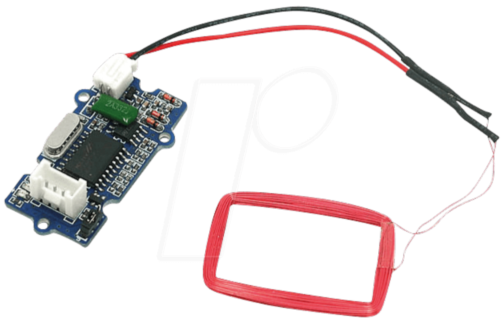


Figure 27: RFID Reader - Labs (11)

NFC Reader NFC Tag is a highly integrated Near Field Communication Tag module,this module is I2C interface,which base on M24LR64E-R,M24LR64E-R have a 64-bit unique identifier and 64 -Kbit EEPROM.Grove - NFC Tag attach an independent PCB antenna which can easily stretch out of any enclosure you use, leaving more room for you to design the exterior of your project.

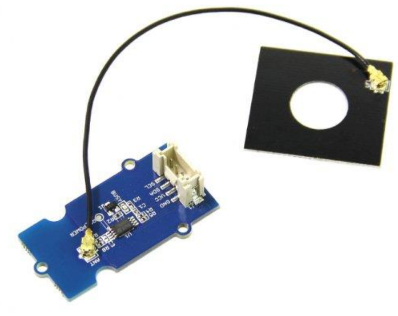


Figure 28: NFC Reader - Labs (11)

RDIF Tags Combo A set of 5 RFID tags, consisting of 3 key rings and 2 cards . Each chip has a unique 64-bit code. Identifiers work a short distance from the device. The carrier frequency is 125 kHz.

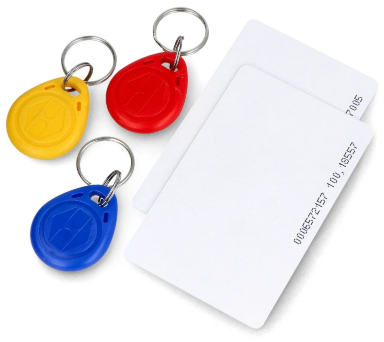


Figure 29: RDIF Tags Combo (11)

NFC Tags NFC Round Cards NFC 215 Card Tag Compatible with TagMo and Amiibo, 504 Bytes Memory Fully Programmable for NFC-Enabled Devices

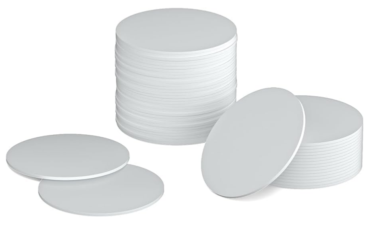


Figure 30: NFC Reader - Labs (11)

Additional Components for BSc and MSc Coursework

Thumb Joystick Joystick modules generate analogue signals that simulate the movements of the cartesian coordinate system. It also has a push button that we can use as input. Its output range is smaller than common joysticks. Thus, most of the time, we map the output to a certain range.

Figure 31: Thumb Joystick - Labs (12)

Gesture Sensor for Arduino Gesture is based on PAJ7620U2 that integrates gesture recognition function with general I2C interface into a single chip. It can recognize 9 gestures including moving up, moving down, moving left, moving right, etc with a simple swipe by your hand.

Figure 32: Gesture Sensor

UART Wifi UART WiFi is a serial transceiver module featuring the ubiquitous ESP8285 IoT SoC. With integrated TCP/IP protocol stack, this module lets your micro-controller interact with WiFi networks with only a few lines of code. Each ESP8285 module comes pre-programmed with an AT command set firmware, meaning you can send simple text commands to control the device. The SoC features integrated WEP, WPA/WPA2, TKIP, AES, and WAPI engines, can act as an access point with DHCP, can join existing WiFi networks and has configurable MAC and IP addresses.

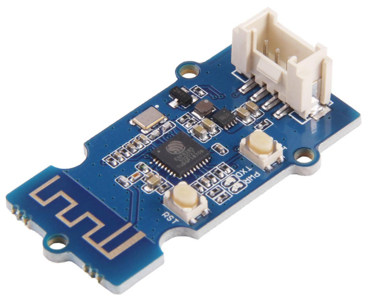


Figure 33: UART WiFi

Triple Colour E-Ink Display Triple Color E-Ink Display 2.13” is a screen that can still be displayed after power off, we call it E-Paper(electronic paper) or E-Ink. The display is a TFT active matrix electrophoretic display, with interface and a reference system design.The 2.13 inch active area contains 212x104 pixels, and has 1-bit white/black and 1-bit red full display capabilities.

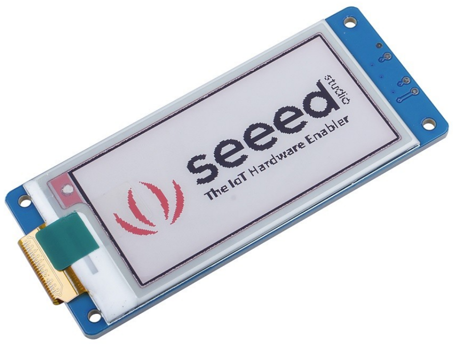


Figure 34: Triple Colour E-Ink Display

Blueseeed (BLE) Blueseeed is a cost-effective, low-power, true system-on-chip (SoC) for Bluetooth low energy applications. It enables robust BLE master or slave nodes to be built with very low total bill-of-material costs. It is based on TI CC2540 chip, which has AT command support.

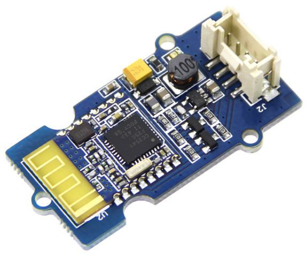
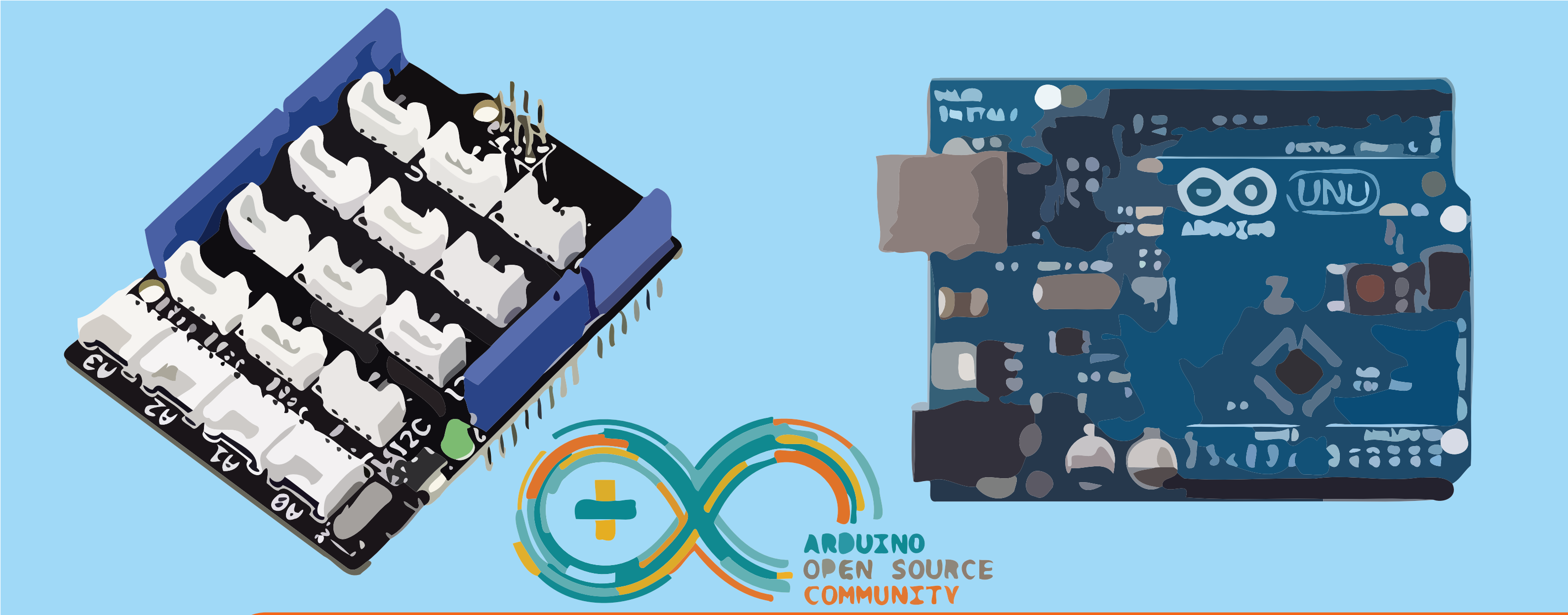


Figure 35: Blueseeed

Hardware Requirements Summary

Hardware Components	🟢 Lab 1	🟢 Lab 2	🟢 Lab 3	🟢 Lab 4	🟢 Lab 5	🟢 Lab 6	🟣 Lab 7	🟣 Lab 8	🟣 Lab 9	🟣 Lab 10	🟣 Lab 11	🟣 Lab 12	🟣 Lab 13	🟣 Lab 14	🟣 Lab 15	🟣 Lab 16
Raspberry Pi (3B+, 4)		✓	✓	✓	✓	✓	✓		✓							✓
GrovePi Plus		✓	✓	✓		✓			✓							
Arduino Expansion Shield for RPi	✓				✓	✓		✓		✓	✓	✓		✓		
Base Shield for Arduino	✓				✓	✓		✓		✓	✓	✓		✓		
Loudness (Sound) Sensor						✓			✓	✓						
Light Sensor	✓					✓		✓								
PIR Motion Sensor	✓				✓	✓										
Temperature/Humidity (V11,V20)		✓	✓	✓		✓										
Servo Motor	✓					✓										
LED Button (Yellow/Blue/Red)	✓					✓										
Ultrasonic Ranger		✓				✓		✓	✓	✓						
Buzzer		✓	✓	✓		✓			✓							
LCD Backlight Display (V4,V5)		✓				✓				✓	✓	✓				
Serial Bluetooth v3.0					✓	✓										
Raspberry Pi Camera v2																✓
SD Card		✓	✓	✓		✓	✓		✓							✓
Micro HDMI to HDMI Cable					✓		✓	✓	✓							✓
USB Keyboard and Mouse	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Monitor	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LED Bar								✓	✓					✓		
Rotary Angle Sensor								✓								
EMG Sensor														✓		
Speaker												✓				
LED Socket Kit / LED										✓	✓					
RFID Reader											✓					
NFC Reader											✓					
RFID Tags Combo											✓					
NFC Tags											✓					
Thumb Joystick												✓				
Blueseeed (BLE)										✓						
Gesture Sensor for Arduino																
UART WiFi																
Triple Colour E-Ink Display																

Table 1: Required Hardware for Each Lab Tutorial



1. Micro-Controller Programming

Objectives

- Learn how to program a microcontroller (e.g., Arduino) with the Arduino IDE
- Learn how to read data from multiple sensors (light sensor, PIR motion sensor)
- Learn how to handle basic event-driven programming
- Learn how to integrate and control an actuator (servo motor)

Lab Plan

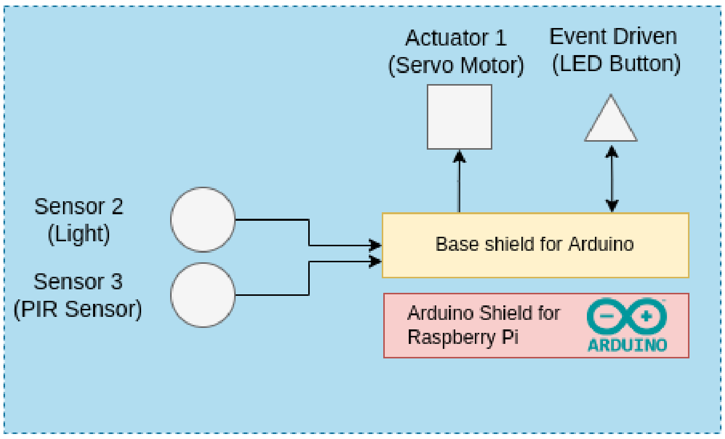


Figure 1.1: Conceptual Diagram of Lab 1 Setup

In this lab, you will:

1. Verify your Arduino environment and upload an empty (test) sketch.
2. Connect and read data from a **light sensor** (analog) and a **PIR motion sensor** (digital).
3. Use a **button** with an integrated LED to switch between sensor modes.
4. Attach and control a **servo motor** based on sensor conditions.

Required Hardware Components

- **Microcontroller board:** Arduino Leonardo or Arduino Uno
- **Grove Base Shield for Arduino**
- **Sensors:**
  - **Light Sensor (A3 port)** (analog output)
  - **PIR Motion Sensor (D2 port)** (digital output)
- **Actuators:**
  - **Servo Motor (D5 port)** (PWM required)
  - **LED Button (D3 port)** (digital input + LED indicator)
- **USB Cable** (Arduino to PC)

Important: Check the Base Shield Power: Make sure the small black switch on the shield is set to **5V** if your sensors require 5V. Confirm the green LED on the base shield is illuminated; if not, toggle the power switch on the shield's bottom-left corner. (Some sensors may work on 3.3V, but this lab assumes 5V.) ■

A. Setting Up the Arduino Environment

1. **Install or Open the Arduino IDE.**
   If you are using the lab machines, the Arduino IDE is preinstalled. Look for the Arduino icon on the desktop or in your applications menu.
   If you are working on your own computer, you can download and install the Arduino IDE from `https://www.arduino.cc/en/Main/Software`.

   — How to Install the Arduino Desktop IDE on your own computer. The Arduino Software (IDE) allows you to write programs (called "sketches") and upload them to your board. At the Arduino website: `https://www.arduino.cc/en/Main/Software`, you will find two main options:

   Online IDE If you have a reliable Internet connection, consider using the *Arduino Web Editor*. This option allows you to:
   - Save your sketches in the cloud, so you can access them from anywhere.
   - Always stay up to date with the latest IDE version and libraries.
   - Avoid manual installation or compatibility issues on your computer.

   Desktop IDE If you would rather work offline, or if your connection is limited, install the latest version of the desktop IDE. You can download it for Windows, macOS, or Linux.
   - **Windows users:** Run the installer, or unzip the portable version if you prefer no-install setups.
   - **macOS users:** Drag and drop the Arduino application into your *Applications* folder.
   - **Linux users:** Extract the downloaded package and run the installation script or use your distribution's package manager (if available).

   Once installed, launch the IDE to verify it opens without errors and recognizes your board when connected.

2. **Connect Arduino via USB and Select Board/Port.**
   Use a standard USB cable (type A to micro/mini/USB-C, depending on your board) to connect your Arduino to the computer. Give the system a few seconds to recognize the device.
   In the Arduino IDE:

- **Tools** → **Board** → *Arduino Leonardo* (or select the correct board model if using an *Arduino Uno*, *Arduino Mega*, etc.).
- **Tools** → **Port** → choose the COM port (Windows) or `/dev/ttyUSB_n`, `/dev/ttyACM_n`, or `/dev/cu.usbmodem_n` (macOS/Linux) that corresponds to your Arduino.

This step ensures the IDE knows exactly what type of board you have and which serial port to use when uploading sketches.

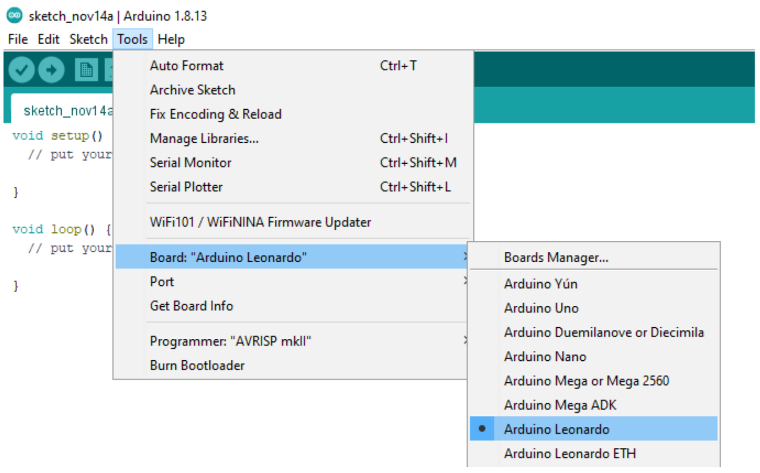

sketch_nov14a | Arduino 1.8.13
File Edit Sketch Tools Help
sketch_nov14a
void setup()
// put your
}
void loop() {
// put your
}
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino Leonardo"
Port
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Boards Manager...
Arduino Yún
Arduino Uno
Arduino Duemilanove or Diecimila
Arduino Nano
Arduino Mega or Mega 2560
Arduino Mega ADK
● Arduino Leonardo
Arduino Leonardo ETH

Figure 1.2: Configuring the Arduino IDE Board to Arduino Leonardo

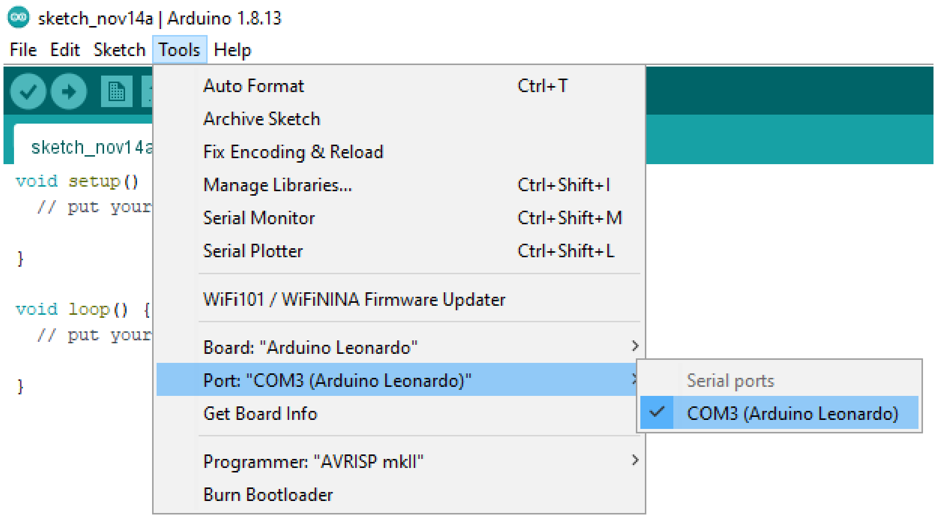

sketch_nov14a | Arduino 1.8.13
File Edit Sketch Tools Help
sketch_nov14a
void setup()
// put your
}
void loop() {
// put your
}
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino Leonardo"
Port: "COM3 (Arduino Leonardo)"
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Serial ports
✓ COM3 (Arduino Leonardo)

Figure 1.3: Selecting the Correct Port

3. **Upload an Empty Sketch for Testing.**
   Testing your setup with a simple, empty sketch confirms that the IDE can communicate with the Arduino.
   - **Create a new sketch** via `File` → `New`. This opens a blank template with two main functions: `setup()` and `loop()`.
   - **Click the Upload icon** ( ) to send this empty sketch to your Arduino.
   - If the process completes with "Done uploading" and no errors, your board is correctly set up and ready to run sketches.

— **Why do this?.** An empty sketch upload is a quick way to ensure the board and IDE can communicate. If you encounter errors here, it is easier to troubleshoot before adding any complex code.

4. **(Optional) Checking Serial Port Issues.**
   Occasionally, your computer may not detect the Arduino board or may assign an unexpected port name. If the IDE cannot detect your board/port:
   - **Windows Users:** Open *Device Manager* and check under *Ports (COM & LPT)*. Look for "Arduino" or "USB Serial Device" and note the COM number.
   - **macOS/Linux Users:** Open a terminal and type `ls /dev/tty.*` (macOS) or `ls /dev/ttyACM*` (Linux) to see all available serial devices. The Arduino port usually appears shortly after plugging the board in.

   If still undetected:
   - Try a different USB cable or USB port (some cables are charge-only, lacking data lines).
   - Restart the Arduino IDE after reconnecting the board.
   - Ensure you installed any drivers that certain Arduino clones might require.

Tips: If you still see no port, check that your board is receiving power (the onboard LED should be lit), and confirm your USB cable is not damaged. ■

B. Hardware Assembly

5. **Attach the Base Shield on the Arduino.**
   Carefully align the male pins from the underside of the Base Shield with the female pin headers on the Arduino. Make sure you press gently and evenly so that the shield sits flush on the board.

   — **Why do this?.** The Base Shield makes it simpler to connect Grove-compatible sensors and actuators without messy wiring. It has clearly labeled ports that correspond to digital, analog, or I^2C pins on the Arduino, minimizing wiring confusion.

6. **Connect Your Sensors and the Button:**
   Use Grove cables to link each component to the corresponding labeled port on the Base Shield.
   - **LED Button → D3 port**
     Align the four-pin connector on the LED button cable with the D3 port. Gently push it in until it clicks.
   - **PIR Motion Sensor → D2 port**
     The PIR sensor is sensitive to movement; placing it on a digital port (D2) makes it easier to detect HIGH/LOW motion signals.
   - **Light Sensor → A3 port**
     This sensor outputs an analog voltage that varies with light intensity, so connecting it to an analog input (A3) allows the Arduino to read precise light-level values.

Tips: Ensure that each Grove cable is fully inserted. If the sensor or button does not respond later in testing, double-check the connection or try re-seating the Grove cable. ■

7. **(Later) Connect the Servo:**
   After you confirm that your sensors and button work correctly, plug in the servo motor to **D5**, which is a PWM-capable pin (supports Pulse Width Modulation).

> **— Why do this?.** The servo is powered from the same supply as the Arduino; connecting it last helps you focus on sensor debugging first. Using a PWM pin gives you finer control over the servo's angle.

C. Reading Sensors and Handling Events

8. **Obtain and Type the Example Sketch.**
   In the GitLab repository, locate `light_motion_button.ino`. Rather than copying and pasting, **type the code manually** to practice each line. This helps you fully absorb how variables, functions, and libraries work together to read sensor data and handle user input.

   > **— Why do this?.** Typing the code by hand forces you to process each line and often reveals smaller details—like syntax, library calls, and variable names—that might be missed when you copy and paste. This builds a stronger foundation for troubleshooting and understanding code structure.

9. **Verify and Upload the Sketch.**

   - Click the checkmark icon (✓) to verify (compile) the code. Fix any typos or syntax errors if they appear in the console.
   - Then click the upload icon (→) to flash (upload) the compiled program to the board.
   - After upload, the LED on the button should blink by default. When you press the button, it switches from reading the *light* sensor to reading the *PIR* sensor.

   > **Tips:** If the LED on the button does not blink or the upload fails, re-check your board selection (**Tools** → **Board**) and COM port (**Tools** → **Port**). Also ensure the USB cable is firmly connected. ■

10. **Open the Serial Monitor.**
    Click the Serial Monitor icon (🔍) or go to **Tools** → **Serial Monitor** to view real-time sensor readings.

    > **— No Output in the serial monitor?.** If you are able to open "Serial Monitor" but there is no output, make sure the Base Shield is properly.
    > **Check the "Green" light** The LED on base shield must be illuminated in green colour like shown in figure 1.4
    > **Turn on "Base Shield"** If there is no illuminated green light on base shield then switch the black button found on the bottom left of base shield as shown in figure 1.4.

11. **Testing the Sensors.**

    - **Press/hold the button:** The code switches from reading the light sensor to reading the PIR motion sensor. Release the button to switch back.
    - **Light Sensor:** Cover or shine a light on the sensor to change the analog reading in the Serial Monitor. The closer you cover it, the lower the value; the brighter the environment, the higher the value.
    - **PIR Sensor:** Wave your hand in front of it or walk in front of the sensor. You should see a digital "HIGH" or "Motion Detected" printout when movement is sensed.

    > **Calibrating the Light Sensor** If your environment is extremely bright or dim, you may need to adjust thresholds in your code. For instance, you might change:

```
if (lightValue < 200) {
    // it's quite dark
}
```

Print out the `lightValue` using `Serial.println(lightValue)` to see actual numbers and set thresholds that make sense for your lab conditions.

12. **Final Setup Example.**
    Figure 1.4 shows a reference layout of the light sensor, PIR motion sensor, and LED button connected to the Base Shield.

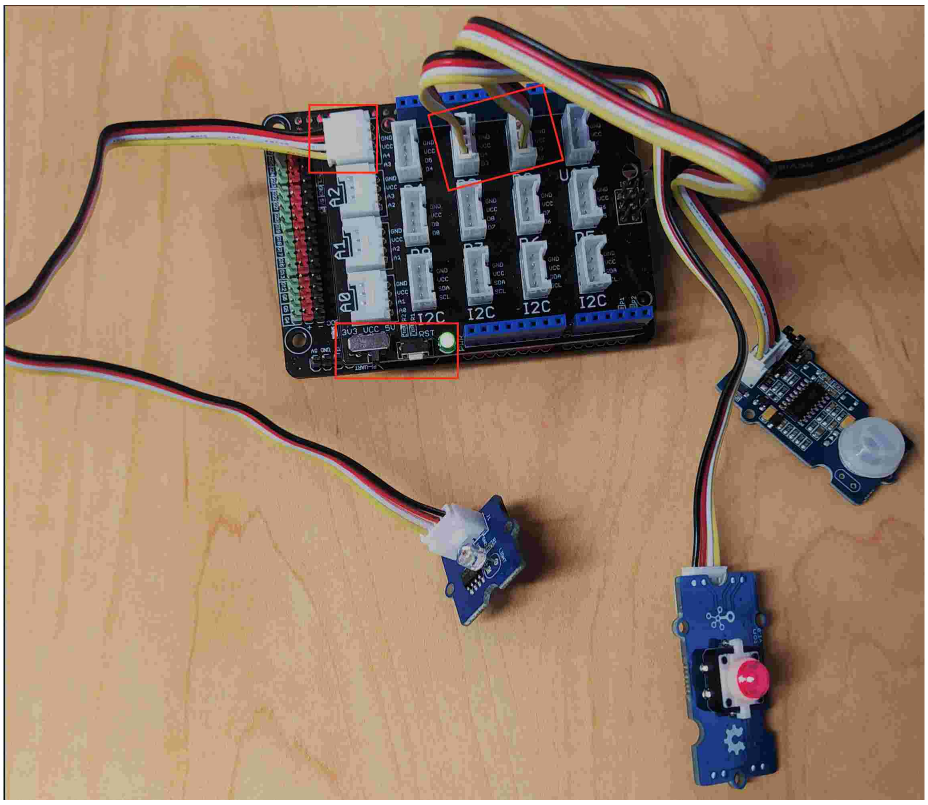
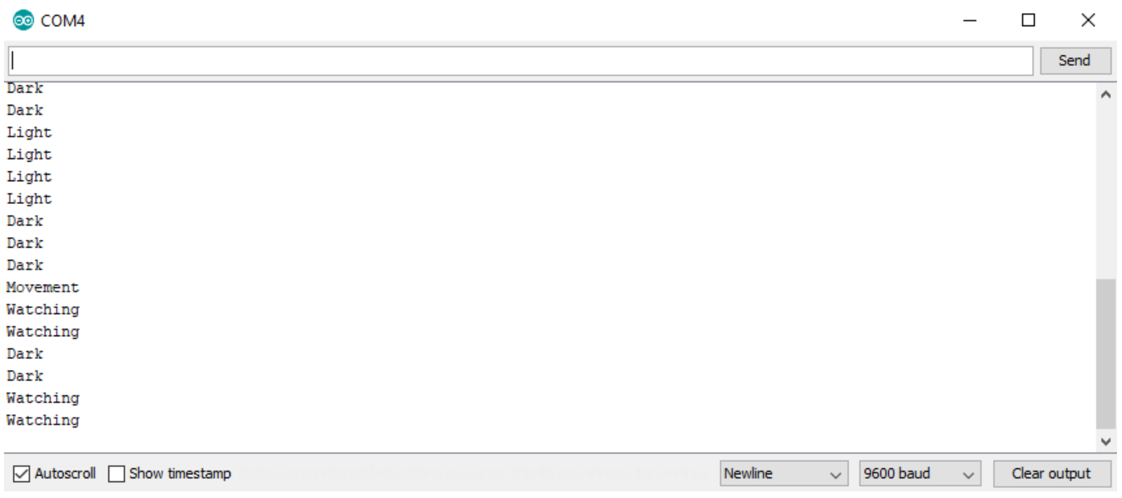


Figure 1.4: Final Setup: Light Sensor, PIR Sensor, and LED Button on Base Shield

13. **Sample Serial Monitor Output.**
    If everything is wired and programmed correctly, your Serial Monitor will display a continuous stream of sensor readings, as in Figure 1.5.



```
Dark
Dark
Light
Light
Light
Light
Dark
Dark
Dark
Movement
Watching
Watching
Dark
Dark
Watching
Watching
```

Newline | 9600 baud

Figure 1.5: Serial Monitor readings for Light/PIR

D. Incorporating the Servo Motor

13. **Connect the Servo (D5 port).**
Plug the servo's three-wire cable (signal, VCC, ground) into the **D5** port on the Base Shield. If your servo cable does not have a Grove connector, ensure the pin order matches the shield's pin labeling (*SIG* to servo signal, *VCC* to servo power, *GND* to ground).
Next, download or open `servo_example.ino` from the same GitLab repository. This example sketch typically initializes the servo and sweeps it through a range of angles. Upload it to verify that the servo can move freely.

Tips: If the servo does not move or makes jittering noises:
- Double-check that the *SIG*, *VCC*, and *GND* wires are correctly aligned.
- Some servos may require slightly higher current. Ensure your Arduino is powered via USB *and* (if needed) an external power supply on the barrel jack.
- If you see random movements, try adding a short `delay(100)` after each angle command.

14. **Observe the Servo.**
With the `servo_example.ino` running, look for any movement as the code updates the servo angle. You can also integrate it with the LED button or existing sensors:
    - **LED Button Integration**: Pressing the button might trigger a specific angle, e.g., `180°` for "lock" or `0°` for "unlock."
    - **Sensor-Driven Behavior**: If the code is monitoring the PIR or light sensor, it can instruct the servo to rotate to a certain angle whenever motion is detected or when the light reading crosses a threshold.

— Why do this?. Linking sensors to a servo actuator demonstrates how real IoT devices respond to environmental changes. For instance, a locked door can open automatically when light levels are adequate, or a motion sensor triggers the servo for a security function.

15. **Refine Your Code.**
As you combine the servo with other components, consider:
    - **Create helper functions**: For instance, `void moveServoTo(int angle){...}` can help keep the `loop()` clean and readable.

- **Use** `Serial.println()`: Print out the servo angle or relevant sensor values. This helps with debugging and confirms that your logic (e.g., "Move to 180° if motion is detected") is being executed.
- **Manage Timing**: If the servo should only move once every few seconds, consider using `delay()` or track elapsed time with `millis()`.

Tips: If your code becomes crowded, separate it into sections: one for sensor handling, one for servo actions, and one for user interface (LED button). This modular approach eases troubleshooting.

16. **Optional Use-Case: "Door Control"**
    Imagine the servo shaft is attached to a mini "door" or latch:
    - If it's dark **OR** the PIR sensor detects motion, rotate the servo to `180°` (fully closed).
    - **Delay or Timer:** Add `delay(10000)` (10 seconds) to keep it closed, then move it back to `0°` if you want it to re-open automatically.
    - Combine with a **Buzzer or LED** for additional feedback, e.g., beep when the door closes.

    This scenario simulates a security or privacy feature and showcases how multiple sensors and actuators can work together.

Measuring Success

- **Blinking LED Button**: By default, the LED on the button blinks, and stops blinking while pressed.
- **Sensor Readings**: The Serial Monitor updates with readings from both the light sensor (analog) and the PIR motion sensor (digital).
- **Servo Motor Control**: The servo correctly rotates (0°–180°) based on sensor events or button press logic.

— Further Reading

- **Grove Sensors Overview**:
  `http://wiki.seeedstudio.com/Sensor/`
- **Arduino Programming Tutorials**:
  `https://www.arduino.cc/en/Tutorial/HomePage`
- **Deeper Sensor Theory**:
  `https://learn.sparkfun.com/tutorials/sensors`
- **Advanced PWM and Servos**:
  `https://www.arduino.cc/en/Reference/Servo`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the key theoretical concepts related to microcontroller-based projects. By understanding these fundamentals, you will gain deeper insights into how sensors, actuators, and event-driven logic come together in embedded systems and the Internet of Things (IoT).

A. Microcontroller Basics

A **microcontroller** (e.g., the ATmega32U4 or ATmega328P used on Arduino boards) is a compact integrated circuit designed to execute simple, dedicated tasks:

- **CPU Core:** Executes program instructions stored in internal Flash memory.
- **RAM:** Used for temporary data while running the program.
- **GPIO (General-Purpose Input/Output) Pins:** Allow connection to sensors, actuators, LEDs, and more.

A.1 Key Differences From Single-Board Computers

Unlike a Raspberry Pi (which runs a full operating system), a microcontroller typically:

- **Runs "bare-metal" code**: No multi-user OS.
- **Limited clock speed and memory**: For example, 16 MHz clock, 2 kB RAM.
- **Boots almost instantly and focuses on real-time responsiveness**.

A.1.1 Why Use a Microcontroller?

- **Real-Time Control:** Quick, deterministic responses to sensor inputs and hardware events.
- **Low Power Consumption:** Great for battery-powered or low-energy applications.
- **Simplicity:** Less complex than running a full OS; quick to set up, straightforward to deploy.

B. Sensor Interfacing and Signal Types

In this lab, you worked with a **light sensor** (analog) and a **PIR motion sensor** (digital). Microcontrollers can handle both analog and digital signals:

B.1 Analog Signals: Light Sensor

An **analog** signal can vary continuously over a range (e.g., 0–5 V). The Arduino includes an **Analog-to-Digital Converter (ADC)** on certain pins (A0, A1, etc.) to convert the voltage to a digital value:

- **10-bit Resolution:** Arduino Uno/Leonardo map 0–5 V to an integer range 0–1023.
- **Light Sensor Output:** More light $\rightarrow$ higher voltage $\rightarrow$ higher ADC reading.

B.1.1 Calibration and Thresholds

- **Calibrate:** Observe raw ADC readings in different lighting conditions (very dark vs. brightly lit).
- **Set a Threshold:** Decide at which reading the environment is "dark" or "bright." For instance, `if (lightValue < 200)` might indicate darkness.

B.2 Digital Signals: PIR Motion Sensor

A **PIR (Passive Infrared)** sensor detects abrupt changes in infrared radiation (e.g., a person moving). It outputs either:

- **LOW (0 V):** No motion detected.
- **HIGH (5 V):** Motion detected.

B.2.1 How PIR Works

- **Pyroelectric Element:** Generates a small charge when the IR level changes.
- **Fresnel Lens:** Broadens the detection area, focusing IR onto the sensor.

- **Internal Circuitry:** Interprets the signal and outputs a digital HIGH upon motion detection.

C. Event-Driven Programming and Code Structure

Event-driven programming means the system reacts to specific "events" (button presses, sensor triggers) rather than running everything continuously in a single loop. While Arduino code typically uses a `loop()`, you can mimic event-driven behavior by checking conditions each cycle:

- **Button Press Event:** Switch from reading the light sensor to reading the PIR sensor.
- **Motion Detected:** If PIR is HIGH, take an action (e.g., move the servo).

C.1 Handling Multiple Sensors

Inside the `loop()`, you might see:

- **Check Light Value:** Store in a variable, compare to a threshold.
- **Check PIR Output:** If `digitalRead(PIR_pin) == HIGH`, set a flag or trigger an action.
- **Button State:** If pressed, switch your "mode" or logic branch.

C.1.1 Avoiding Delays

- `delay()` **can block** reading sensors in real time. Consider using short delays or a non-blocking approach (e.g., using `millis()`).
- **Responsive Systems** often rely on quickly cycling through `loop()` to catch events promptly.

D. Controlling Actuators: Servo Motors

A **servo motor** can rotate to a specified angle (e.g., 0°–180°). Arduinos use **PWM (Pulse Width Modulation)** to send control signals:

- **Standard Servos:** Typically interpret a 1–2 ms pulse within a 20 ms frame (50 Hz).
  - $\sim$1.0 ms pulse $\rightarrow$ 0°
  - $\sim$1.5 ms pulse $\rightarrow$ 90°
  - $\sim$2.0 ms pulse $\rightarrow$ 180°
- `Servo` **Library** in Arduino simplifies generating these precise pulse widths.

D.1 Example Use Cases

- **Automated Door:** Rotate servo to 180° if motion is detected or if it's dark.
- **Indicator Arm:** Point an attached arrow or dial to indicate sensor readings (e.g., "safe" vs. "danger").

D.1.1 Avoiding Jitter

- **Stable Power:** Servos draw current surges; ensure enough power supply or use an external supply if needed.
- **Smooth Movement:** Gradually move the servo by small increments with short delays if abrupt motion is undesirable.

E. Putting It All Together: Example Workflows

- **Initialize Pins and Libraries:** In `setup()`, configure pin modes for the light sensor, PIR sensor, button, and servo.
- **Loop Processing:** Continuously read sensor values, check button state, and conditionally set servo angles.
- **Debugging via Serial Monitor:** Print sensor readings and servo positions to `Serial` for real-time feedback.
- **Calibration and Refined Thresholds:** Adjust code as needed based on real sensor data (light levels, PIR responsiveness).

E.1 Testing Strategy

- **Step-by-Step:** Test each component individually (light sensor reading, PIR detection, servo sweep).
- **Integration:** Combine sensor logic and servo control once each element works independently.
- **Troubleshoot:** If servo twitches erratically, check power. If PIR triggers too often, ensure no direct drafts or vibrations.

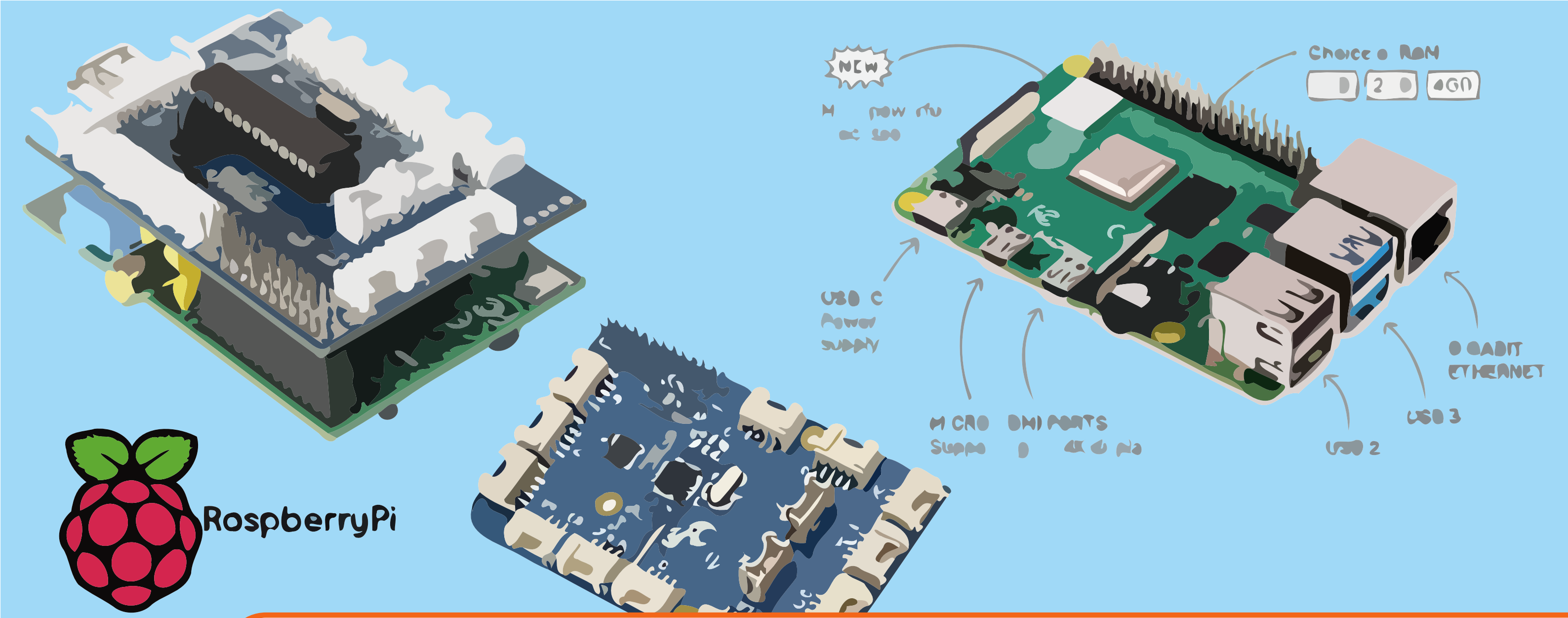
F. Further Extensions and Real-World Considerations

- **Buzzer or LED Alerts:** Add an audible or visual indicator when motion/light crosses thresholds.
- **Data Logging:** Store sensor values over time to detect patterns or run basic analytics (e.g., how often motion is detected at night).
- **Power Optimization:** If battery-powered, reduce sensor polling rate or use sleep modes to prolong battery life.
- **Shield Compatibility:** Different shields may have alternate pin mappings for sensors; always double-check documentation.

By exploring these topics, you enhance the functionality of your microcontroller projects, bridging the gap from simple demos to robust IoT solutions.

Final Note.

Understanding **microcontroller architecture**, **analog/digital signals**, **event-driven programming**, and **PWM control** is fundamental. By applying these concepts, you are well on your way to designing more advanced, creative, and reliable embedded systems. Experimentation and iterative testing will be your best guide as you refine and expand your projects.

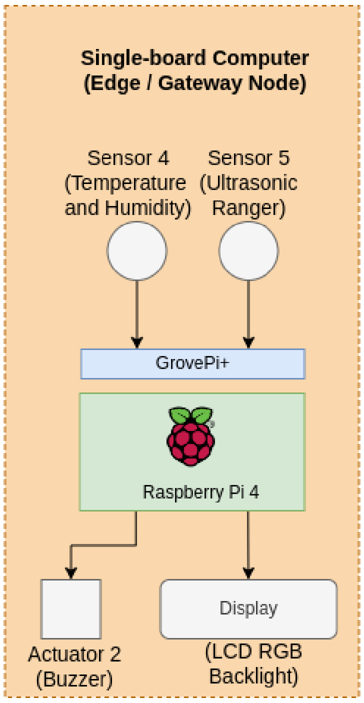


2. Single-board Computer Programming

Objective

- Learn how to program a Single-board Computer (Raspberry Pi model 4B with 4GB RAM)
- Learn how to read data from multiple sensors
- Learn how to program actuators (e.g., a buzzer)
- Learn how to program a display (LCD)

Lab Plan



In this lab, we will:

1. Set up a Raspberry Pi with Grove Pi+ and verify that all sensors are detected.
2. Use Node-RED to run Python scripts (via the `daemon` node).
3. Read data from a Temperature & Humidity sensor and an Ultrasonic Ranger.
4. Display sensor data on an LCD screen.
5. Trigger a buzzer when certain conditions are met.
6. Complete an optional use-case scenario ("Cat in the Kitchen") to tie everything together.

Required Hardware Components

- **Raspberry Pi**
- **Grove Pi+**
- **Grove sensors**: Ultrasonic Ranger, Temperature & Humidity sensor
- **Buzzer**
- **LCD Backlight Display**
- **SD Card with OS installed** (Raspberry Pi OS)
- **Display and HDMI cable**
- **Keyboard and mouse**
- **Power supply**

A. Setup of the Raspberry Pi

1. **Insert the SD card into the SD card slot.**
   This microSD card is preloaded with the Raspberry Pi OS (operating system). Gently push it into the card slot on the underside of the Pi until it clicks into place.

   > **— Why do this?.** The SD card contains the Pi's operating system and all required files to boot. Without it, the Pi cannot start up or run the lab environment.

2. **Connect Peripherals (but do not power yet):**

   - **Keyboard & Mouse:** Plug these into any available USB ports on the Pi.
   - **Monitor:** Use a micro HDMI to HDMI (or DVI) cable to attach your Pi to a display. This allows you to see the desktop and operate the Pi using its GUI.
   - **Important:** Do *not* apply power to the Pi yet. You will do this *after* confirming all other connections.

   > **Tips:** If you have a Raspberry Pi 4, you may have two micro HDMI ports. Usually, the left port (when the USB-C power is on your right) is the primary display port. If you see no display after powering on, try the other port. ■

3. **Attach Grove Pi+ to your Raspberry Pi.**
   Align the Grove Pi+ board's 40-pin connector with the Pi's GPIO header. Apply gentle pressure so it sits securely without bending pins.

   > **— Why do this?.** Grove Pi+ provides convenient Grove ports (D, A, I^{2}C) that map to the Pi's GPIO pins. This simplifies sensor and actuator connections, reducing wiring complexity.

4. **Connect Sensors to Grove Pi+** as shown below:
   - **LCD Backlight $\rightarrow$ I2C-2 port**
     This display uses I^{2}C communication, so it must be plugged into an I2C port for data and power.

- **Temperature & Humidity Sensor (DHT) → D4 port**
  This digital sensor reads environmental data and sends it on a single pin (D4).
- **Ultrasonic Ranger → D3 port**
  The ultrasonic module calculates distance by sending a ping and measuring its echo.
- **Buzzer → D8 port**
  A digital pin (D8) can drive the buzzer to create various tones or alarms.

> Tips: Confirm the cables click snugly into the Grove ports. If a sensor does not respond later, recheck you inserted it into the correct numbered port. ■

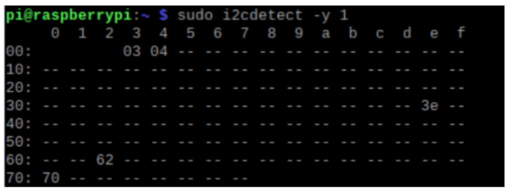
5. **Power on the Raspberry Pi.**
   Connect the Pi's power supply (USB-C or micro USB, depending on your model). You should see a **red LED** illuminate, indicating the Pi has power. The monitor should display a boot sequence with small "raspberry" icons.
6. **Connect to Wi-Fi.**
   After the desktop finishes loading:
   - Click the network icon (top-right corner of the desktop) to select and join a Wi-Fi network.
   - Alternatively, open a terminal and run `sudo raspi-config`, then navigate to **Network Options → Wi-Fi**.
7. **Check Date and Time.**
   If your Pi's clock is off, adjust it via the top menu bar or using the `raspi-config` tool (detailed in Chapter 17). Having an accurate clock is crucial for logging sensor data and authenticating with certain network services.
8. **Enable I2C (if not already).**
   Click the Raspberry icon , go to **Preferences → Raspberry Pi Configuration → Interfaces** and enable *I2C*. Reboot if prompted.

   > **— Why do this?.** The Pi's I2C interface is disabled by default for security and resource reasons. Many Grove modules (LCD displays, some sensors) communicate via I2C, so enabling it is essential to allow the Pi to detect them.

9. **Verify GrovePi+ via i2cdetect:**
   In a terminal window, type:

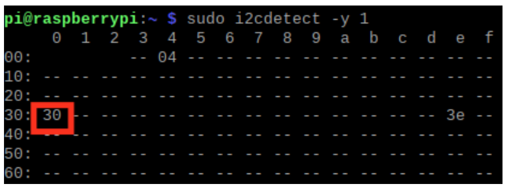
```
sudo i2cdetect -y 1

```

If you can see **04** in the output (similar to Figure 2.1), the Pi successfully detects the GrovePi+ board. Figures 2.1a and 2.1b show expected addresses for LCD versions 4.0 or 5.0.



```
pi@raspberrypi:~ $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          03 04 -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 3e --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- 62 -- -- -- -- -- -- -- -- -- -- -- -- --
70: 70 -- -- -- -- -- -- --
```

(a) LCD v4.0



```
pi@raspberrypi:~ $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- 04 -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: 30 -- -- -- -- -- -- -- -- -- -- -- -- -- 3e --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

(b) LCD v5.0

Figure 2.1: Verifying GrovePi+ Connection for different LCD versions

Tips: **04** typically indicates the GrovePi+ device address. If you have multiple devices on the I2C bus, you may see other addresses. For example, an I2C LCD might show up around **3e** or **3f**.

10. **If you DON'T see 04:**

    - Reseat the GrovePi+ board and run `i2cdetect -y 1` again.
    - If it still doesn't appear, update and reboot:

```
bash <(curl -sL https://raw.githubusercontent.com/node-red/linux-installers/master/deb/update-nodejs-and-nodered)
curl -kL dexterindustries.com/update_grovepi | bash
sudo reboot
```

— Why do this?. The above scripts update Node-RED and the GrovePi+ firmware, which can fix certain compatibility issues. If you still do not see address **04** after rebooting, there may be a hardware fault or a deeper configuration issue.

B. Programming with Node-RED

11. **Download the `dht.py` Script**
    Access the GitLab repository at this link. Save the file `dht.py` into your **/home/pi** directory on the Raspberry Pi.

    — Why do this?. This Python script reads temperature and humidity data from the DHT sensor. Saving it in **/home/pi** (the default home directory) makes it easy to reference in Node-RED or other scripts without navigating complex file paths.

12. **Make the Python script executable.**

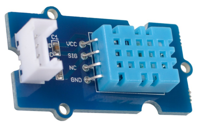
```
sudo chmod 755 dht.py
```

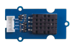
This command grants execution permissions, allowing Node-RED or any user to run the script directly (e.g., via `./dht.py` or `python3 dht.py`).

— Different Sensor? (DHT20). Plug-in I2C Port: If you have the black DHT20 sensor, it uses the I^{2}C protocol. Connect it to an I^{2}C port (e.g., I2C-1) on GrovePi+ instead of a digital port like D4.
Download Script: In the GitLab repository, look for `dht20.py` instead of `dht.py`.
Change File Permissions: Be sure to run:

```
sudo chmod 755 dht20.py
```

Adjust Code: If your Node-RED flow points to `dht.py`, rename references to `dht20.py`.


(a) DHT11 (Digital Signal)


(b) DHT20 (I^{2}C)

13. **Check npm Version (Optional).**

```
npm -v
```

If the version number is below 6, you may need to re-run the update steps from the previous lab section to upgrade Node.js and npm. Outdated versions can cause compatibility issues with certain Node-RED nodes or libraries.

14. **Starting Node-RED.**
    From the Raspberry Pi's desktop:
    - Click the Raspberry menu (top-left corner) → **Programming** → **Node-RED**, *or*
    - Open a terminal and type:

    ```
    node-red
    ```

    A new terminal window titled **Node-RED console** indicates the Node-RED server is launching.

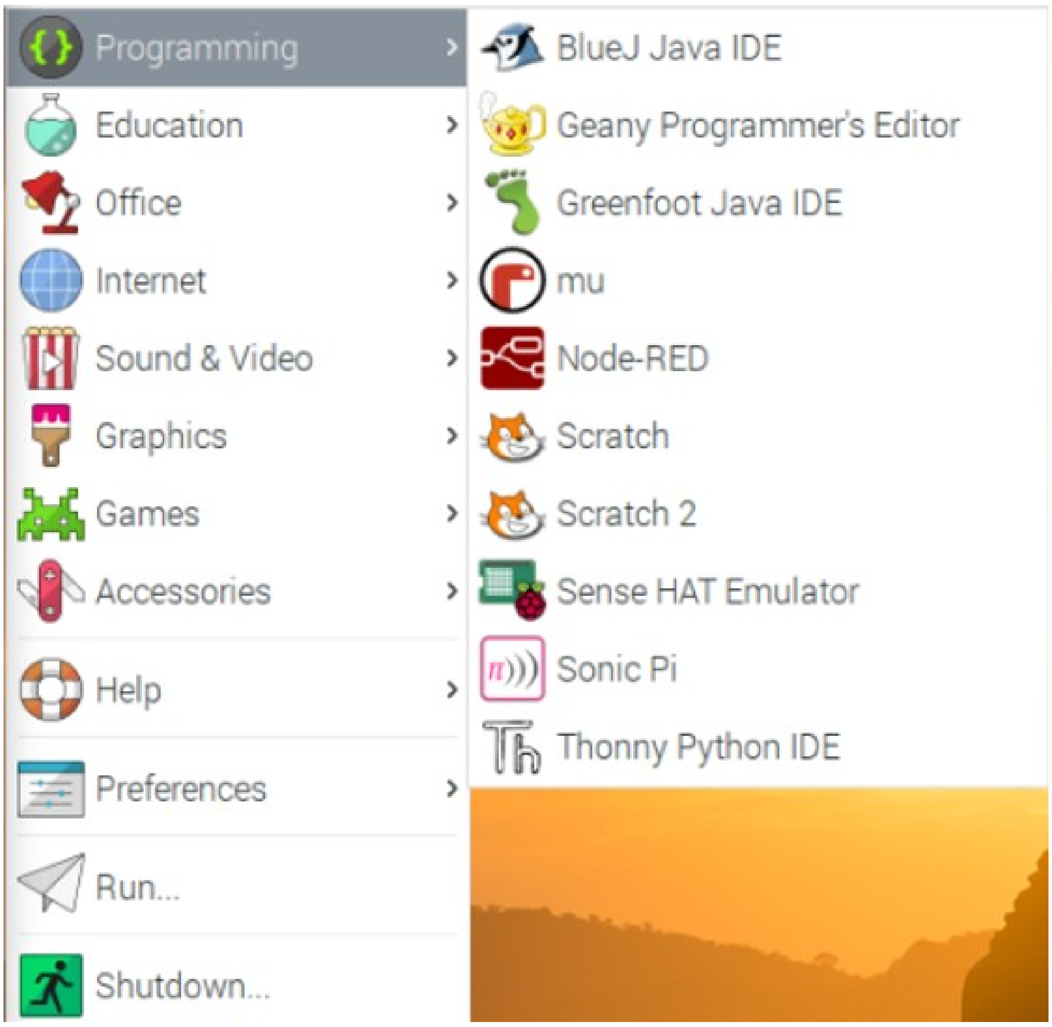


Figure 2.3: Starting the Node-RED service from the Pi Menu

> **— Why do this?.** Node-RED is a flow-based development tool, ideal for IoT projects. By running it locally on the Pi, you can visually wire together hardware inputs (sensors), logic nodes, and outputs (e.g., LCD or cloud services) without having to write lengthy code.

15. **Node-RED Console and Local Server.**
    When Node-RED finishes starting, you'll see status messages (like "Server now running at

http://127.0.0.1:1880") in the console. **Open a web browser** on the Pi (e.g., Chromium) and navigate to:

```
http://localhost:1880
```

This is the Node-RED flow editor interface. You can ignore non-critical warnings in the console, such as missing optional libraries.

16. **Clean the Workspace and Deploy Once.**
The editor may have sample nodes or flows from previous sessions. Remove any unwanted flows or nodes:
    - Select nodes, press `Delete`, or click the menu icon to remove entire flows.
    - Click the red **Deploy** button (top-right). This saves your changes and initializes a fresh workspace for your project.
17. **Install `node-red-node-daemon` if Not Present.**
In the flow editor, click the menu icon ☰ (top-right corner) and select **Manage Palette** as shown in Figure 2.4. Then:
    - **Install tab:** Search for `node-red-node-daemon`.
    - If it's missing, click **Install**.
    - Wait for the installation to complete before closing the window or redeploying.

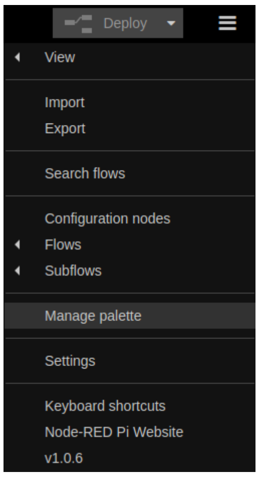
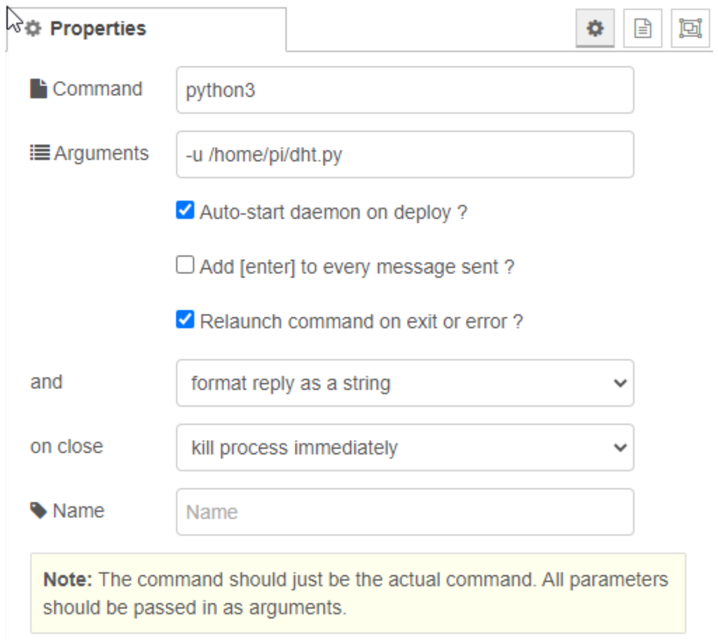


Figure 2.4: Manage Palette to Install Additional Nodes

18. **Add and Configure the Daemon Node.**

    - Locate the **daemon** node in the left panel (it may appear under "function" or "advanced" categories).
    - Drag it onto the workspace.
    - Double-click the daemon node to open its properties (Figure 2.5):
        - **Command**: `python3`
        - **Arguments**: `-u /home/pi/dht.py`
(`-u` for unbuffered, ensuring real-time output in Node-RED.)
        - **Auto-start**: Check "Start daemon on deploy" if you want the script to run each time you click **Deploy**.
19. **Connect a Debug Node.**

    - Drag a **debug** node from the left palette onto the workspace.


Properties

Command: python3

Arguments: -u /home/pi/dht.py

- [✓] Auto-start daemon on deploy ?
- [ ] Add [enter] to every message sent ?
- [✓] Relaunch command on exit or error ?

and: format reply as a string

on close: kill process immediately

Name: Name

Note: The command should just be the actual command. All parameters should be passed in as arguments.

Figure 2.5: Configuring the daemon Node with `python3 -u /home/pi/dht.py`

- Connect its input (the left side of the debug node) to the daemon node's output (the right side of the daemon node), forming a single wire between them.
- This debug node will display the temperature and humidity data that `dht.py` prints, helping you verify sensor readings.

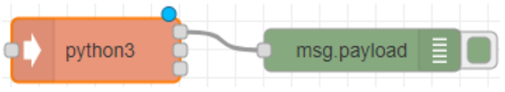
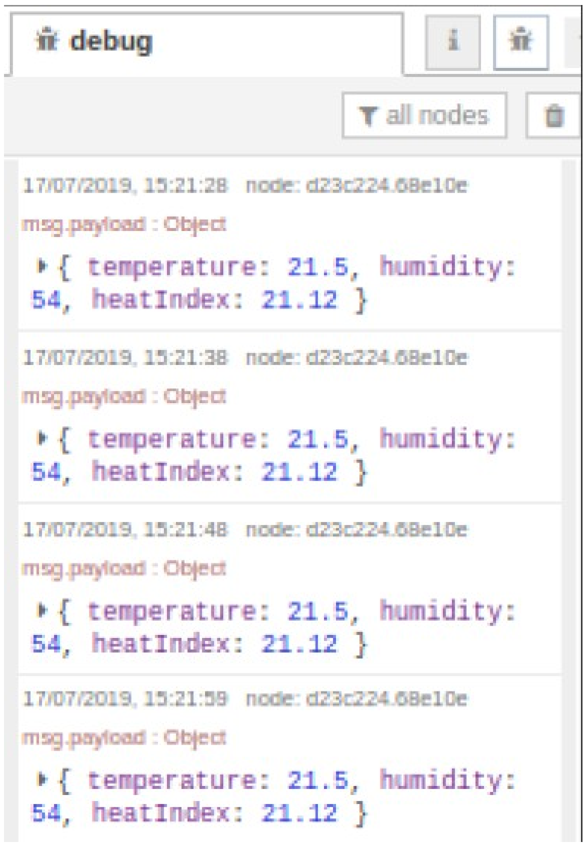


Figure 2.6: Simple Flow with a Daemon Node Feeding a Debug Node

20. **Deploy and Observe.**
Click **Deploy** again. In the right-hand panel of the Node-RED editor, click the **Debug** tab to watch real-time output.
If everything is correct, `dht.py` will stream temperature/humidity data, as shown in Figure 2.7. Any issues or error messages (e.g., "Command not found" or "No such file") may indicate a missing script or incorrect path.

Tips: If no data appears:
- Verify `/home/pi/dht.py` actually exists and is executable (`chmod 755`).
- Confirm the DHT sensor is on the correct port (D4 by default in the script) and well-seated on GrovePi+.
- Try running `python3 /home/pi/dht.py` in a terminal to check if it prints readings outside Node-RED.

21. **Remove Connection, Re-Deploy (Optional).**
If you want to stop the console from filling with data but keep the daemon running:
- Click the wire between the daemon node and the debug node, then press `Delete`.
- Click **Deploy** again. The script will still run, but its output no longer routes to the Debug panel.



```
17/07/2019, 15:21:28 node: d23c224.68e10e
msg.payload : Object
▸{ temperature: 21.5, humidity:
54, heatIndex: 21.12 }
17/07/2019, 15:21:38 node: d23c224.68e10e
msg.payload : Object
▸{ temperature: 21.5, humidity:
54, heatIndex: 21.12 }
17/07/2019, 15:21:48 node: d23c224.68e10e
msg.payload : Object
▸{ temperature: 21.5, humidity:
54, heatIndex: 21.12 }
17/07/2019, 15:21:59 node: d23c224.68e10e
msg.payload : Object
▸{ temperature: 21.5, humidity:
54, heatIndex: 21.12 }
```

Figure 2.7: Debug Window Showing Temperature/Humidity Values

— Why do this?. Large volumes of data can clutter the debug panel, making it hard to read other messages. Temporarily removing or disabling the debug connection is a convenient way to silence continuous output.

C. Creating a Thermometer with Display

20. **Download** `lcd.py`
    From the GitLab repository, locate and download `lcd.py`, placing it in your `/home/pi` directory. This script displays temperature data on the Grove-LCD RGB Backlight.

    — Why do this?. Unlike the `dht.py` script, which only outputs readings to the console, `lcd.py` showcases these readings on a physical LCD display. It provides a visual thermometer-like experience and is helpful for quick, real-world feedback without needing to open a serial monitor or debug window.

21. **Make** `lcd.py` **Executable.**

```
sudo chmod 755 lcd.py

```

    This step grants run permissions so that Node-RED (or any user) can execute `lcd.py` directly.

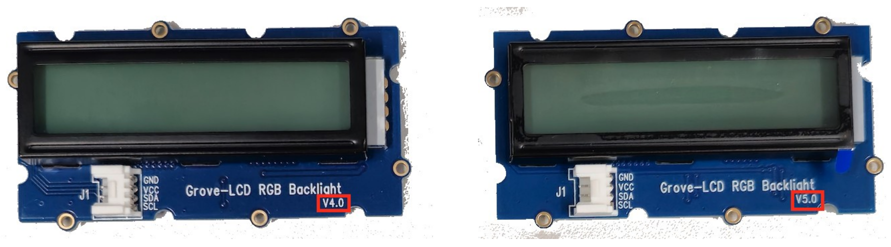
    — No output or error reading temperature?. Are you using a different screen or sensor? For instance, Grove-LCD RGB Backlight V5.0 instead of V4.0, or a DHT20 sensor rather than DHT11?
    What Happened? The `lcd.py` script is tailored for a DHT11 sensor on an LCD v4.0.

Using different hardware might result in missing or incorrect data.

Download Look for `dht#_lcd#.py` in the GitLab repository that matches your exact hardware combination (e.g., DHT20 sensor and LCD v5.0).

Change Access Control After saving the correct file name, apply `chmod 755` so it becomes executable.

Example For LCD v5.0 with a DHT20 sensor, you might need `dht20_lcd5.py`.



(a) LCD version 4.0 (b) LCD version 5.0

Figure 2.8: Different versions of the LCD

22. **Configure the Daemon Node.**
    In Node-RED, open your existing daemon node's properties and replace:
    - **Command:** `python3`
    - **Arguments:** `-u /home/pi/dht.py`

    with:
    - **Command:** `python3`
    - **Arguments:** `-u /home/pi/lcd.py`

    Then click **Deploy** once more to apply these changes.

Tips: If you want to switch between `dht.py` (console-only) and `lcd.py` (LCD output), consider creating two daemon nodes in Node-RED. Simply enable the one you need by connecting it to a debug node or relevant logic. ■

23. **Check the LCD.**
    If everything is wired correctly:
    - The LCD's backlight should illuminate.
    - Temperature (and possibly humidity) values should scroll or update on the display.
    - If you see no changes, verify that the LCD is plugged into **I2C-2** (or an I^{2}C port) and confirm `lcd.py` references the correct pins or addresses.

— Why do this?. Displaying data on an LCD provides a tangible, user-friendly readout—ideal for standalone devices. You can keep an eye on temperature/humidity levels without needing a laptop or additional serial terminal.

D. Programming Buzzer and Ranger

24. **Download** `buzzerRanger.py`**.**
    Navigate to the GitLab repository and save `buzzerRanger.py` to your `/home/pi` directory.

— Why do this?. This Python script demonstrates how to read distance values from the ultrasonic ranger and control the buzzer accordingly. Storing the file in `/home/pi` keeps it consistent with other lab scripts (e.g., `dht.py`, `lcd.py`) and makes it easy for Node-RED to locate.

25. **Make it Executable.**

```
sudo chmod 755 buzzerRanger.py

```

This command ensures the script has the necessary permissions to run when called by Node-RED or via a terminal.

26. **Change Daemon Node to** `buzzerRanger.py`**.**
In your Node-RED workspace:
    - Double-click the **daemon** node you used for `lcd.py`.
    - **Command**: `python3`
    - **Arguments**: `-u /home/pi/buzzerRanger.py`
    - Click **Done**, then **Deploy**.

Tips: If you want to preserve your thermometer code, you can duplicate the daemon node:
- One daemon runs `lcd.py` (for temperature display).
- Another daemon runs `buzzerRanger.py`.

Simply link or unlink each daemon to a debug node as needed.

27. **Observe Node-RED Debug.**
Open the **Debug** panel in Node-RED:
    - The console should show ultrasonic distance values in centimeters.
    - If the distance reading drops below 10 cm, the buzzer will start beeping.
    - Wave an object (e.g., your hand) closer or farther to test the buzzer's activation threshold.

— Why do this?. Visualizing distance in Node-RED while simultaneously hearing buzzer feedback offers a clear demonstration of how sensor data can trigger immediate real-world actions. This pattern—reading a sensor, applying logic, and activating an actuator—is fundamental to most IoT applications.

E. Use Case Scenario (Optional): "Cat in the Kitchen"

28. **Scenario Overview:**
You're cooking dinner, and your curious cat keeps approaching the oven. To prevent accidents, you want an alert that escalates as the cat gets closer:
    - **Discrete Buzzer Tone**: If the cat is within a moderate range (e.g., 20–30 cm), the buzzer beeps briefly every 2 seconds.
    - **Continuous or Rapid Beep**: If the distance is below 10 cm, the buzzer stays on or beeps rapidly.
    - **LCD Display**: Show both *temperature* and *distance* values in real time.

— Why do this?. This scenario ties together sensors (ultrasonic ranger for distance and DHT for temperature) and actuators (buzzer, LCD). It simulates a safety or alert mechanism often found in real-world IoT systems—providing visual and audible feedback based on sensor thresholds.

29. **Modify** `buzzerRanger.py`:
    - Integrate new logic to read the ultrasonic distance. Use conditional statements to:
        - **Beep Interval**: If distance is, say, 20–30 cm, play a short beep every 2 seconds using `time.sleep(2)`.
        - **Rapid Beep/Continuous Tone**: If distance < 10 cm, run the buzzer in a loop without delay or with a very short pause (e.g., 0.1 s).

- Print/log the distance so Node-RED's debug tab displays how close the cat (or your hand) is.
- **Optional**: Incorporate the temperature reading (from `dht.py` or a combined script) so that your script can display and react to temperature as well.

Tips:

- If you need both temperature and distance simultaneously, merge `buzzerRanger.py` and `lcd.py` into one script. Read both sensors, then decide on buzzer behavior and LCD output within the same Python file.
- Use non-blocking techniques (e.g., `time.time()` checks or threading) if you need the buzzer to beep repeatedly while still reading new sensor data.

30. **(Optional) Additional Features:**
    Feel free to extend this scenario with even more IoT functionality:
    - **Notifications**: In Node-RED, add an email or SMS node to send you a warning if the distance remains too close for a prolonged period (e.g., 30 seconds).
    - **Complex Triggers**: Combine temperature readings with distance thresholds. For instance, if the oven temperature is above a certain level *and* the cat is under 10 cm away, trigger a louder alarm or flash an LED.
    - **Data Logging**: Send distance and temperature data to a database or cloud platform so you can analyze patterns over time.

Measuring Success

- **Sensor Data in Node-RED**: You can see output from your Python scripts (e.g., `dht.py`, `buzzerRanger.py`) in the Node-RED debug tab.
- **LCD Updates**: The `lcd.py` (or your hardware-specific script) displays temperature (and humidity, if configured) on the LCD screen.
- **Buzzer Behavior**: The buzzer rings when the ultrasonic reading is below 10 cm. Adjust thresholds in your script to see different beep patterns.
- **Optional Use-Case Logic**: In the "Cat in the Kitchen" scenario, your code should produce a discrete or continuous beep based on the distance, and display the distance on the LCD.

— Further Reading

- **Node-RED Docs**:
  `https://nodered.org/docs/`
- **Raspberry Pi**:
  `https://www.raspberrypi.org/`

Theory Deep Dive: Underlying Principles and Concepts

This section explores additional theoretical underpinnings of the hardware and software used in this lab. By understanding these fundamentals, you will build a stronger foundation in single-board computer (SBC) projects and gain insight into how operating systems, digital interfaces, sensors, and actuators come together in the Internet of Things (IoT).

A. Single-Board Computer (SBC) Basics

A **Single-Board Computer** (SBC), such as the Raspberry Pi, is a full computer built on a single circuit board. Unlike a simpler microcontroller (e.g., Arduino), an SBC typically offers:

- **More powerful CPU:** Often an ARM processor capable of running Linux or other full operating systems.
- **Larger RAM:** Raspberry Pi 4B can have 2 GB, 4 GB, or even 8 GB of RAM.
- **Storage via SD card or eMMC:** SBCs rely on microSD cards (or built-in eMMC) as their "hard drive."
- **Ports and Connectivity:** USB, HDMI, Ethernet, Wi-Fi, and Bluetooth for versatile peripherals and network access.

A.1 Comparing SBCs and Microcontrollers

It is useful to understand how SBCs differ from traditional microcontrollers:

- **Operating System:** SBCs typically run a full OS (e.g., Linux), while microcontrollers run "bare metal" or a lightweight RTOS.
- **Development Environment:** You can program an SBC using Python, Node.js, C++, etc., directly on the board. Microcontrollers often require cross-compilers and specialized toolchains.
- **Performance vs. Real-Time:** SBCs have higher processing power and memory, but microcontrollers sometimes offer more predictable timing for hard real-time tasks.

A.1.1 Boot Process and System Resources

When you power on a Raspberry Pi:

1. The GPU firmware (stored on the SD card) initializes basic hardware.
2. The `bootloader` loads the Linux kernel into memory.
3. Linux starts system services and user programs (e.g., desktop, Node-RED).

This multi-stage boot is more complex than a microcontroller, which typically jumps directly to user code after a simple reset routine. However, the OS on an SBC provides advanced features like multitasking, file systems, and networking that ease IoT development.

B. Linux OS and Node-RED Basics

A Raspberry Pi typically runs a Linux-based operating system (e.g., Raspberry Pi OS). Basic Linux skills are crucial for:

- **Package Management:** Installing or updating software using `apt-get` or `apt`.
- **File System Hierarchy:** Familiar directories like `/home/pi`, `/etc`, `/usr`, etc.
- **Processes and Permissions:** Linux manages multiple programs (processes) simultaneously; `sudo` grants administrative privileges.

B.1 Node-RED Overview

Node-RED is a flow-based tool for visually wiring together hardware devices, APIs, and online services:

- **Based on Node.js:** Written in JavaScript, installable via `npm`.
- **Browser-Based Editor:** Accessed at `http://localhost:1880` on the Pi.
- **Palette of Nodes:** Includes input nodes (e.g., `daemon`), output nodes (`debug`), and function nodes for custom logic.

B.1.1 Event-Driven vs. Polling Approaches

Many Node-RED flows are **event-driven**, meaning nodes react to incoming messages (events) rather than constantly polling hardware in a tight loop. This is especially useful when integrating multiple sensors and actuators:

- **Efficiency:** The flow remains idle until new data arrives.
- **Scalability:** Additional sensors can be added as new event-driven nodes without drastically changing the rest of the system.

C. Digital Communication and I2C

I2C (Inter-Integrated Circuit) is a two-wire interface commonly used in SBC and microcontroller projects:

- **SDA (Data) and SCL (Clock):** These two lines allow communication between a master (Raspberry Pi) and multiple slaves (sensors, displays).
- **Addresses:** Each I2C device has a unique 7-bit address (e.g., `0x04` for Grove Pi+).
- **Bus Speeds:** Standard mode (100 kHz), Fast mode (400 kHz), among others.

C.1 Sensor Interfacing: Temperature & Humidity, Ultrasonic

Temperature & Humidity Sensors (DHT11, DHT20):

- **DHT11:** Uses a single-wire protocol. Typically connected to a digital port (e.g., D4 on Grove Pi+).
- **DHT20:** Communicates via I2C. Plugged into an I2C port (I2C-1 or I2C-2).
- **Measurement Principle:** Often use a humidity-sensitive capacitor and an NTC thermistor, converting analog changes into a digital signal.

Ultrasonic Ranger:

- Sends out a 40 kHz pulse and measures the echo return time.
- **Distance** = (speed of sound) $\times \frac{\text{echo time}}{2}$.
- Environmental factors (temperature, humidity) can slightly affect the speed of sound.

C.1.1 Practical Considerations

- **Minimum Ranges:** Ultrasonic sensors often cannot detect objects closer than 2–3 cm accurately.
- **Max Range & Cone Angle:** The beam spreads out with distance, so small objects might not always reflect properly.
- **Noise or Interference:** Multiple ultrasonic sensors used simultaneously may interfere with each other's signals.

D. Actuators: Buzzer and LCD

Actuators convert electrical signals into physical outputs. In this lab, you used:

- **Buzzer:** A piezo element that emits a tone when driven with a digital or PWM signal.
- **LCD Display:** An I2C-driven screen (e.g., Grove-LCD) for showing sensor values or status messages.

D.1 Buzzer Basics

A simple **piezo buzzer** typically:

- Generates an audible beep when a voltage is applied.
- Allows for varying beep patterns by toggling `HIGH/LOW` with short delays or using PWM for different tones.

D.1.1 Why Use a Buzzer?

- **Alerts and Alarms:** Quickly notifies the user if a threshold is exceeded (e.g., distance $< 10\,\text{cm}$).
- **Immediate Feedback:** Helps catch attention even if the user isn't looking at the screen.

D.2 LCD Display Basics

An **LCD (Liquid Crystal Display)** with an I2C interface:

- Receives text commands (e.g., "Temp: 25 °C") via the I2C bus.
- Some models have RGB backlights allowing color changes (alert modes, neutral modes, etc.).
- Only requires two main wires (SDA, SCL) plus power pins, freeing up GPIO for other uses.

E. Automating Logic: Event-Driven Scripts in Python and Node-RED

While the Raspberry Pi can directly run Python scripts, combining Python with **Node-RED** offers a more robust event-driven system:

- **Daemon Node:** Launch Python scripts from Node-RED and capture output in real time (stdout).
- **Flow Logic:** Connect different nodes (e.g., `debug`, `switch`, `function`) to handle sensor data or implement complex conditions.
- **Integration:** Link your scripts to web services, mobile notifications, or local displays for a complete IoT pipeline.

E.1 Python Essentials for Sensor/Actuator Control

- **Libraries:** Many Pi projects use `RPi.GPIO` or custom libraries (like GrovePi) to read/write pins.
- **Time Module:** `time.sleep()` for delays in beeps or reading intervals.
- **Exception Handling:** Use `try/except` to gracefully handle sensor read errors or missing hardware.

E.1.1 Example Workflow

- **1) Daemon Node:** Calls `python3 dht.py` every time it's deployed or continuously runs in the background.
- **2) Debug Node:** Displays temperature/humidity in Node-RED's debug console.
- **3) Additional Logic:** A `switch` node triggers the buzzer if humidity or distance exceeds a set threshold.

F. Calibration and Thresholding

Sensors may need **calibration** to account for hardware tolerances or environmental variations:

- **Ultrasonic Ranger:** Verify the minimum reliable distance. Adjust your threshold if readings are inconsistent.
- **Temperature/Humidity:** Some DHT sensors might read slightly off; you can apply an offset (e.g., +1 °C) in code.

Thresholding Strategies:

- **Hysteresis:** Introduce a small gap between ON and OFF thresholds (e.g., beep on <10 cm, stop beeping if >12 cm) to avoid rapid toggles.
- **Averaging/Smoothing:** For noisy signals, implement a moving average to stabilize readings.

F.1 Debugging Tips

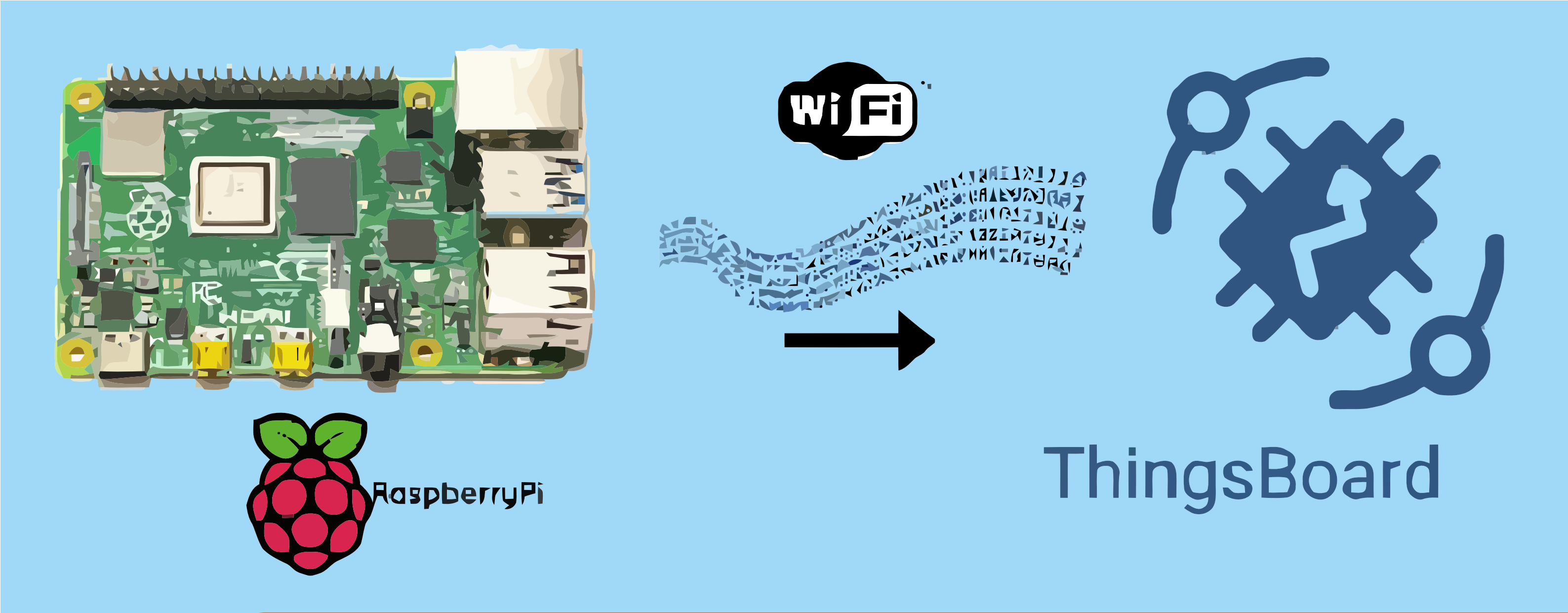
- **Serial Prints/Debug Logs:** Print raw sensor readings to Node-RED's debug window to spot anomalies.
- **Physical Inspection:** Loose wires, reversed connectors, or power supply issues can lead to erratic readings.

G. Further Extensions & Use Cases

- **Data Logging:** Store sensor readings in a file or database (SQLite, InfluxDB) for trend analysis over time.
- **Remote Monitoring:** Create a Node-RED dashboard to display temperature, distance, or beep states over Wi-Fi.
- **Automation Rules:** Combine multiple sensors (e.g., humidity + distance) to trigger multi-stage alerts or control a fan/buzzer automatically.
- **Edge AI:** The Pi's CPU can run advanced frameworks (TensorFlow Lite, OpenCV) for tasks like object detection or machine learning on sensor data.

By adding these features, you can transform a basic sensor-actuator setup into a robust, real-world IoT solution that reacts intelligently to its environment.

Final Note.
Understanding *SBC architecture, Linux fundamentals, Node-RED flow-based programming, I2C communication, sensor principles, and event-driven scripting* sets you on a path to building more sophisticated IoT projects. Experimentation, careful calibration, and iterative testing are key as you progress toward fully polished, real-world solutions.

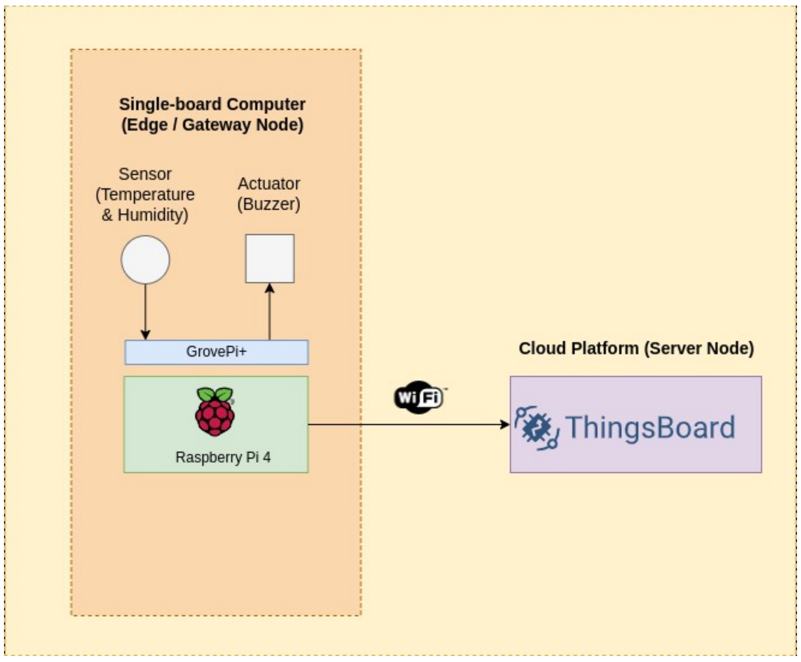


3. Posting Data to an IoT Cloud Platform

Objectives

- Learn how to post IoT data to a cloud IoT platform (ThingsBoard).
- Learn how to configure MQTT in Node-RED for reliable data communication.
- Learn how to create and customize widgets on ThingsBoard dashboards.

Lab Plan


Single-board Computer (Edge / Gateway Node)
Sensor (Temperature & Humidity)
Actuator (Buzzer)
GrovePi+
Raspberry Pi 4
WiFi
Cloud Platform (Server Node)
ThingsBoard

In this lab, we will:

1. Set up a **ThingsBoard** device and copy its access token.

2. Use **Node-RED** on a Raspberry Pi to publish sensor data via MQTT.
3. Visualize real-time readings on a custom **ThingsBoard dashboard**.
4. Optionally create an alarm condition or scenario (e.g., “meteor approaching!”).

Required Hardware Components

- **Raspberry Pi** (with an SD card that has Raspberry Pi OS)
- **Display and HDMI cable**
- **Keyboard and mouse**
- **Power supply**

Additionally, for sensor data:

- **Grove Pi+**
- **Temperature & Humidity Sensor (DHT)**
- **LCD Backlight Display** (optional display output)

A. Preparing the Raspberry Pi

1. **Prerequisite: Lab 2 Completed.**
   Before starting this lab, make sure you followed Lab 2 *up to step 20*, verifying that your Raspberry Pi is correctly reading data from the Temperature & Humidity sensor in Node-RED.

   > **— Why do this?.** Each lab builds on the previous one. Successfully completing Lab 2 ensures that Node-RED is installed and configured, and that you understand how to connect and read basic sensor data. If you skipped any steps or encountered errors, you may face issues in this lab.

2. **Check Node-RED Installation.**
   Confirm Node-RED is still installed and functional on your Raspberry Pi. If you need to reinstall or update:
   - Refer back to Lab 2’s detailed steps on installing Node-RED.
   - Optionally, in a terminal, run:

     ```
     node-red
     ```

     to see if it starts without errors.

   > **Tips:** If Node-RED fails to launch or shows “command not found,” re-run the Node-RED install/update script from Lab 2. Also, verify you have an internet connection before reinstalling, since the script downloads required packages. ■

3. **Verify Sensor Data in Node-RED.**
   Make sure your `dht.py` (or `dht20.py`, if using DHT20) script continues to send valid temperature/humidity readings to a **debug** node in Node-RED. Observe the debug panel:
   - If data appears regularly (e.g., every second or so), you’re good to proceed.
   - If you see no messages, re-check sensor connections or re-deploy the flow.

   > **— Why do this?.** The sensor data is the foundation for this lab. Without verified temperature/humidity readings, the subsequent tasks (e.g., sending data to a cloud platform) won’t function correctly. Confirming the baseline now saves troubleshooting time later.

B. ThingsBoard Device Setup

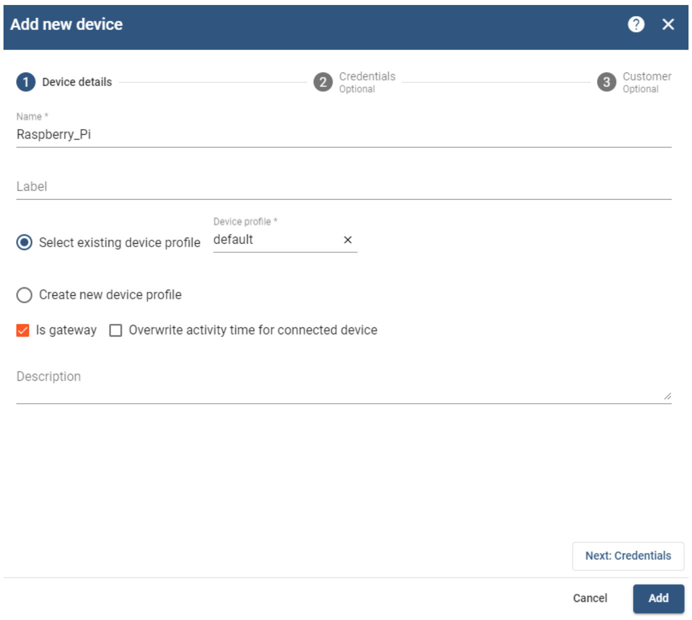
4. **Login to ThingsBoard.**
Open a web browser and navigate to `https://thingsboard.cs.cf.ac.uk/login`. **Log in** using the group credentials provided by your instructor (e.g., a shared username and password).

> **— Why do this?.** ThingsBoard is the IoT platform you'll use for device management and data visualization. Logging in with group credentials ensures everyone can view and manipulate the same devices and dashboards.

5. **Create a New Device.**
In the left-hand menu, click on **Devices** .
   - At the top-right corner, click the **+** icon labeled *Add new device*.
   - Provide a clear, descriptive name (e.g., `GroupX_GatewayDevice`) in the "Device Name" field.
   - Optionally, add a short description (e.g., "Raspberry Pi acting as IoT gateway").

> **Tips:**
> - Avoid using spaces or special characters in the device name to keep things simpler when referencing the device via APIs.
> - If you plan to create multiple devices for different lab tasks, add descriptive details to differentiate them easily.

6. **Configure as Gateway.**
In the device creation form, you'll see a toggle or dropdown for **Device type**. Select **Gateway**, as shown in Figure 3.1. This instructs ThingsBoard to treat your Raspberry Pi as a gateway that can manage and forward data from multiple sensors or other edge devices.


Add new device

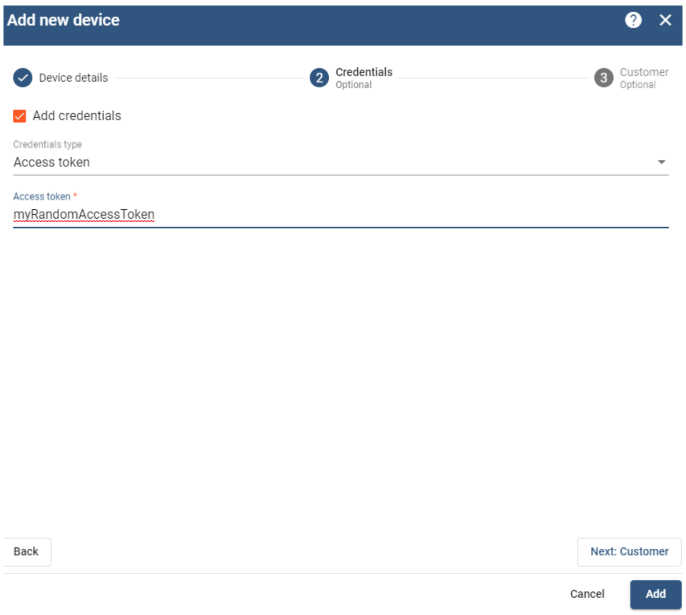
Name *: Raspberry_Pi
Label:
[✓ Select existing device profile] Device profile *: default
Create new device profile
[✓ Is gateway] Overwrite activity time for connected device
Description:

Figure 3.1: Adding New Device

> **— Why do this?.** A gateway device in ThingsBoard can handle data from many sensors, bundling or preprocessing it before sending to the cloud. Marking the device as a "Gateway" enables advanced features like telemetry forwarding and sub-device management.

7. **Next: Credentials.**
   After naming and configuring the device, click the **Next: Credentials** button to set up authentication. Provide an **Access Token**:
   - Either generate a **random** token by clicking a button, *or*
   - Manually enter a **custom** token (use a unique string with letters and numbers).

   Refer to Figure 3.2 for an example of this screen.



Credentials type: Access token
Access token *: myRandomAccessToken

Figure 3.2: Random Access Token

> **Tips:** Using a random token helps avoid accidental collisions. If you choose a custom token, store it somewhere secure. You'll need this value when configuring MQTT or HTTP clients in Node-RED.

8. **Copy the Access Token.**
   Once the device is created, you'll see it listed in your device table. Click its name to open the device details. Under *Credentials*, click **Show token** to reveal the *Access Token*, as shown in Figure 3.3. **Copy** this token.

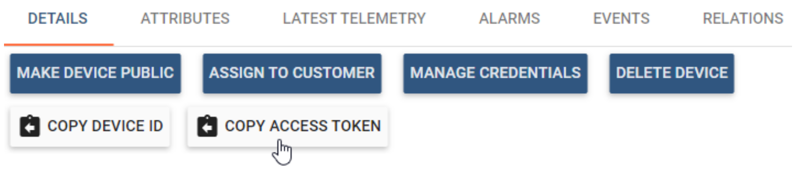


Figure 3.3: Copy Access Token

— Why do this?. The access token is how ThingsBoard authenticates data sent from your gateway. You'll configure Node-RED to include this token in MQTT or HTTP requests so your Pi can securely post telemetry.

C. Publishing Data via Node-RED (MQTT)

9. **Add MQTT Out Node.**
   In the Node-RED editor (left sidebar), locate and drag an **mqtt out** node onto your workspace (Figure 3.4). Connect its input to the **daemon** node (or any node) that outputs your temperature/humidity data, as illustrated in Figure 3.5.

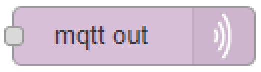


Figure 3.4: MQTT out node.

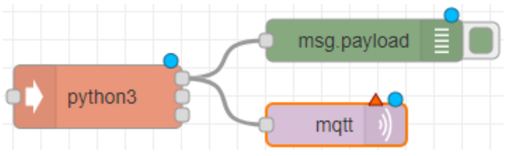


Figure 3.5: Node-RED setup: `daemon` node → `mqtt out`

— Why do this?. The `mqtt out` node publishes messages to an MQTT broker—in this case, ThingsBoard. By wiring it directly after your `daemon` node, you send the temperature/humidity data straight to the cloud in real time.

10. **Configure the MQTT Node.**

    - Double-click on the newly added **mqtt out** node.
    - Next to "Server," click the **pen icon** to add a new MQTT broker configuration.
    - When the modal appears (Figure 3.6), fill in:
      - **Server**: `thingsboard.cs.cf.ac.uk`
      - **Port**: `1883`

      These settings point Node-RED to the ThingsBoard MQTT broker on port `1883`.

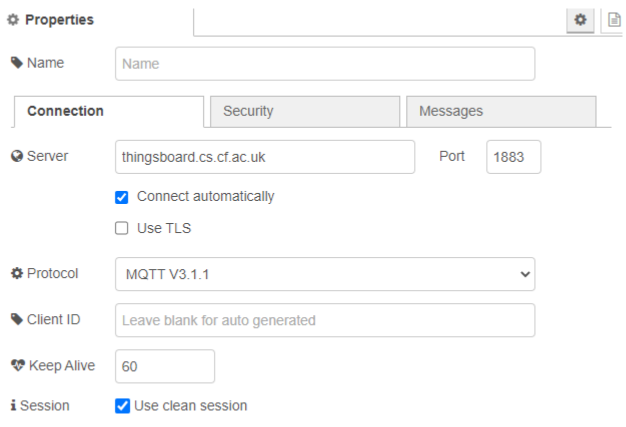
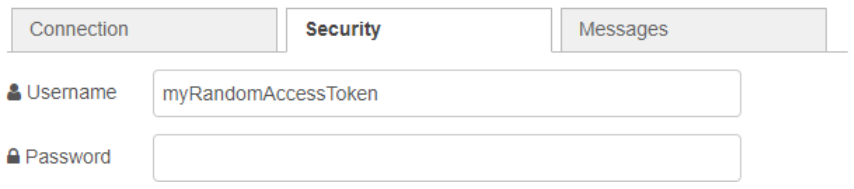

Properties
Name: Name
Connection | Security | Messages
Server: thingsboard.cs.cf.ac.uk Port: 1883
☑ Connect automatically
☐ Use TLS
Protocol: MQTT V3.1.1
Client ID: Leave blank for auto generated
Keep Alive: 60
Session: ☑ Use clean session

Figure 3.6: MQTT Configuration

Tips: If you encounter connection errors, ensure your Pi's internet is stable. Some networks block port 1883, so try alternative networks or check firewall settings if you cannot connect.

11. **Add Security Credentials (Access Token).**

    - Switch to the **Security** tab in the MQTT broker settings (Figure 3.7).
    - For "Username," paste the **access token** you copied from your ThingsBoard device. You do **not** need a password.
    - Click **Update** to save the credentials, then click **Done**.

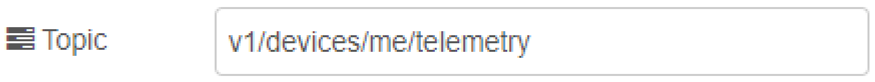


myRandomAccessToken

Figure 3.7: Providing the Access Token as Username

— **Why do this?.** ThingsBoard uses the access token as the MQTT username to authenticate your device. Without these credentials, the broker won't let your Pi publish telemetry. Think of it like a key that proves you're allowed to send data to your device's channel on ThingsBoard.

12. **Set the Topic.**
    In the main properties window of the MQTT out node:
    - Enter `v1/devices/me/telemetry` in the **Topic** field (Figure 3.8).
    - This topic tells ThingsBoard to treat published messages as telemetry data for the *current device* (`me`).
    - Click **Done**, then **Deploy** (top-right corner of Node-RED) to activate your changes.



v1/devices/me/telemetry

Figure 3.8: Specifying the MQTT Topic

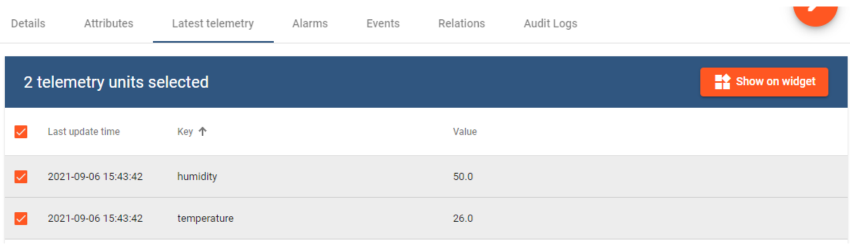
Tips: If you have multiple ThingsBoard devices, each will have its own access token. But the topic `v1/devices/me/telemetry` stays the same for each device—just be sure to use the correct token for the device you want to publish to.

D. Observing Live Data in ThingsBoard

14. **Create a Dashboard.**
    In the ThingsBoard left-hand menu, select **Dashboards** ⊞.
    - Click the **+** icon labeled *Create a new dashboard*.
    - Provide a descriptive name (e.g., *RaspPiDashboard*) and an optional description.

— Why do this?. A dashboard is where you can visually monitor and analyze your data. It can display charts, gauges, tables, or other widgets that update in real time with sensor readings or device status.

15. **Select Your Device & Data Fields.**
    Open the newly created dashboard to configure your widgets:
    - Choose your target device from the available list.
    - Click **Latest telemetry** to view all recent data points published by that device.
    - Check the boxes for **temperature** and **humidity** (Figure 3.9).



2 telemetry units selected

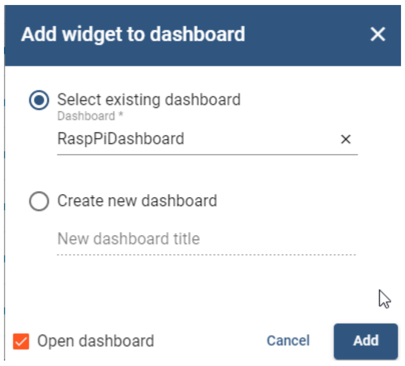
✓	Last update time	Key ↑	Value
✓	2021-09-06 15:43:42	humidity	50.0
✓	2021-09-06 15:43:42	temperature	26.0

Figure 3.9: Latest Telemetry

Tips: If you don't see `temperature` or `humidity` listed, verify that your Node-RED flow is sending telemetry in JSON format (e.g., `{"temperature":25.3,"humidity":60}`). Also confirm your device's access token and MQTT topic are correct. ■

16. **Show on Widget & Add to Dashboard.**

    - Click **Show on widget** next to each telemetry field you want to visualize.
    - By default, the widget type may be *Cards*, but you can pick from a variety of widget options (gauges, charts, etc.).
    - Click **Add to dashboard** and configure your widget settings as shown in Figure 3.10.



[✓ Select existing dashboard]
Dashboard *: RaspPiDashboard

Create new dashboard
New dashboard title

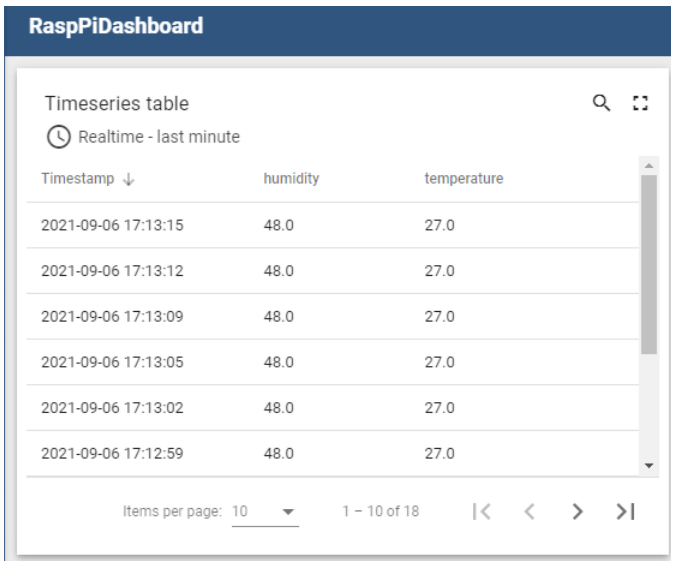
✓ Open dashboard

Figure 3.10: Dashboard Settings

— Why do this?. Widgets let you transform raw sensor data into intuitive, visual representations—graphs, numeric cards, or even maps. Exploring different widget styles helps you find the best fit for your monitoring or analysis needs.

17. **Observe Live Data.**

- After clicking **Add**, the widget appears on your dashboard.
- The dashboard (Figure 3.11) updates in real time with incoming sensor readings.
- Try different widget types (e.g., *Time series charts* or *Gauge*) to see how your data can be presented.



Timestamp ↓	humidity	temperature
2021-09-06 17:13:15	48.0	27.0
2021-09-06 17:13:12	48.0	27.0
2021-09-06 17:13:09	48.0	27.0
2021-09-06 17:13:05	48.0	27.0
2021-09-06 17:13:02	48.0	27.0
2021-09-06 17:12:59	48.0	27.0

Figure 3.11: Observing Live Data on ThingsBoard

Tips:
- **Customize Layout:** You can drag and resize widgets on the dashboard for a personalized layout.
- **Historical Data:** For time-series analysis, configure the widget to display *past* data over minutes/hours/days.
- **Multiple Devices:** If you have multiple devices sending data, you can add more widgets or create separate dashboards to monitor each device individually.

Measuring Success
- **Node-RED to Cloud**: You see temperature/humidity logs arrive in your ThingsBoard dashboard's "Latest Telemetry" or "Cards" widgets.
- **MQTT Config**: The MQTT out node is connected with your correct device token and topic (`v1/devices/me/telemetry`), enabling data flow.
- **Dashboard Display**: Your chosen widget(s) show real-time updates (e.g., 22°C, 55% humidity) that match Node-RED debug outputs.

F. Use-Case Scenario and Shutdown

18. **Optional Final Task (Meteor Approach).**
    Imagine a scenario where a meteor is approaching Earth, causing temperature rises. If it exceeds a certain threshold, a buzzer should ring. Meanwhile, you visualize the temperature changes on your ThingsBoard dashboard.
    - **Hint**: Warm the DHT sensor with your hand to simulate temperature spikes.

- **Add Logic**: Node-RED function node or Python script can trigger a buzzer if `temp > X`.

19. **Shutting Down Safely.**
    If you're done, remove all nodes from the workspace and click **Deploy** to stop sending data. In a terminal:

```
node-red-stop
sudo shutdown now
```

Wait 5 seconds, then unplug your Pi if you won't continue to the final task.

— Further Reading

- **ThingsBoard IoT Use Cases**:
  `https://thingsboard.io/iot-use-cases/`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the theoretical concepts behind sending IoT data to a cloud platform (ThingsBoard), using MQTT in Node-RED for real-time communication, and visualizing sensor data on dynamic dashboards. Understanding these principles will enable you to design robust, scalable, and interactive IoT solutions.

A. IoT Cloud Platforms

An **IoT cloud platform** like ThingsBoard serves as a centralized hub for:

- **Device Management:** Each device or gateway has an identity (access token) for secure data exchange.
- **Data Ingestion & Storage:** Large volumes of time-series sensor data are stored and can be queried or analyzed later.
- **Visualization & Dashboards:** Interactive widgets (charts, gauges, tables) display real-time readings or historical trends.
- **Rule Engine & Alarms:** Configurable logic that can trigger notifications, RPC commands, or external APIs if certain conditions are met.

A.1 Why Use a Cloud Platform?

- **Scalability:** You can manage tens, hundreds, or thousands of devices in one place.
- **Remote Access:** View device data and dashboards from any browser, enabling distributed collaboration.
- **Integrations:** Many IoT platforms include connectors for email, SMS, databases, or webhooks to facilitate complex workflows.

A.1.1 ThingsBoard as an Example

- **Open-Source Core:** ThingsBoard's community edition is freely available, while advanced features exist in commercial editions.
- **Multi-Protocol Support:** MQTT, HTTP, CoAP for data ingestion.
- **Widget Library:** Customizable widgets (charts, switches, maps) for a variety of use cases.

B. MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight publish/subscribe messaging protocol often used in IoT:

- **Broker-Based Architecture:** A central broker (in this case, ThingsBoard's MQTT service) handles all message routing between publishers and subscribers.
- **Topics:** Each message is published to a `topic` string (e.g., `v1/devices/me/telemetry`). Any client subscribed to that topic receives the message.
- **QoS (Quality of Service) Levels:** Control message delivery guarantees (e.g., at most once, at least once, exactly once).

B.1 MQTT in Node-RED

- **mqtt out Node:** Publishes messages to a specified broker at a chosen topic.
- **mqtt in Node:** Subscribes to a topic, receiving messages published by other clients or devices.
- **Authentication:** When connecting to a secure MQTT broker, a username and/or password (or token) is often required.

B.1.1 Common MQTT Use Cases

- **Telemetry Upload:** IoT devices (sensors) periodically publish their readings.
- **Command/Control:** The cloud can publish a command (e.g., "turn on LED") to which the device is subscribed.

C. Node-RED as Edge/Fog Middleware

Node-RED is a flow-based programming tool allowing rapid prototyping of IoT applications:

- **Visual Flows:** Nodes represent devices or functions; wires connect them to define data paths.
- **Integration Hub:** Easily integrate Python scripts, web APIs, and MQTT brokers in a single flow.
- **Debug Panel:** Real-time visibility into data payloads helps with quick troubleshooting.

C.1 Data Path in Node-RED for This Lab

- **Sensor Reads**: A Python script (`dht.py`) outputs temperature/humidity data to Node-RED (via `daemon` node).
- **MQTT Publish**: The `mqtt out` node takes those readings and publishes them to ThingsBoard.
- **Debug & Logging**: An optional `debug` node can print the sensor data in Node-RED for local verification.

C.1.1 Handling Errors or Connection Loss

- **MQTT Node Status**: Node-RED shows a status indicator (green or red) for the `mqtt out` node, reflecting connection success or failure.
- **Retries**: If ThingsBoard becomes temporarily unreachable, the MQTT client may keep retrying, buffering messages in Node-RED's queue.

D. ThingsBoard Device Model and Dashboarding

ThingsBoard organizes IoT infrastructure via **devices**. Each device has:

- **Access Token**: A secret token used to authenticate MQTT or HTTP requests.
- **Telemetry & Attributes**: Time-series data (telemetry) and key-value pairs (attributes) for device properties.
- **RPC Endpoints (Optional)**: Allows remote control or command invocation on the device.

D.1 Dashboards and Widgets

- **Dashboard**: A collection of widgets (charts, gauges, maps, switches) that visualize real-time or historical device data.
- **Latest Telemetry vs. Historical**: "Latest" widgets display only the most recent value; "Charts" or "Tables" show time-series data spanning minutes to months.
- **Customization**: You can rename widgets, choose update intervals, define color thresholds (e.g., red if *temp* > 25).

D.1.1 Data Flow to Widgets

- **Device Selection**: Widgets typically must be linked to a specific device entity or device group.
- **Key Mapping**: The widget looks for the data key (e.g., *temperature*, *humidity*) in the incoming telemetry stream.

E. Real-Time IoT Scenarios and Advanced Logic

Once you can post sensor data to the cloud and visualize it, you can add more sophisticated behaviors:

- **Alerting/Notifications**: If temperature exceeds a threshold, ThingsBoard can trigger an email, Slack message, or SMS.
- **Device Control (RPC)**: A switch widget on the dashboard can send commands back to the Pi or an Arduino, toggling actuators (buzzers, LEDs, motors).
- **Machine Learning Pipelines**: Data stored in the cloud can be used to train anomaly detection models or predictive maintenance routines.

E.1 Example: "Meteor Approaching"

- **Temperature Rising**: Node-RED function checks if `temp > 30`.
- **Buzzer Alarm**: If condition met, send a `HIGH` signal to a digital pin or via the **daemon** node to a Python script controlling the buzzer.
- **Dashboard Indicator**: A widget on ThingsBoard shows a "Meteor Approaching!" message and changes color to red.

F. Security and Reliability Concerns

- **Access Token Protection**: Anyone with your token can publish false data or spam the broker; keep it private.
- **TLS Encryption (Optional)**: MQTT over TLS can encrypt data in transit, though additional configuration is required.
- **Connection Monitoring**: Watch for "disconnected" or "reconnected" events in Node-RED or the ThingsBoard device logs to ensure data is flowing continuously.

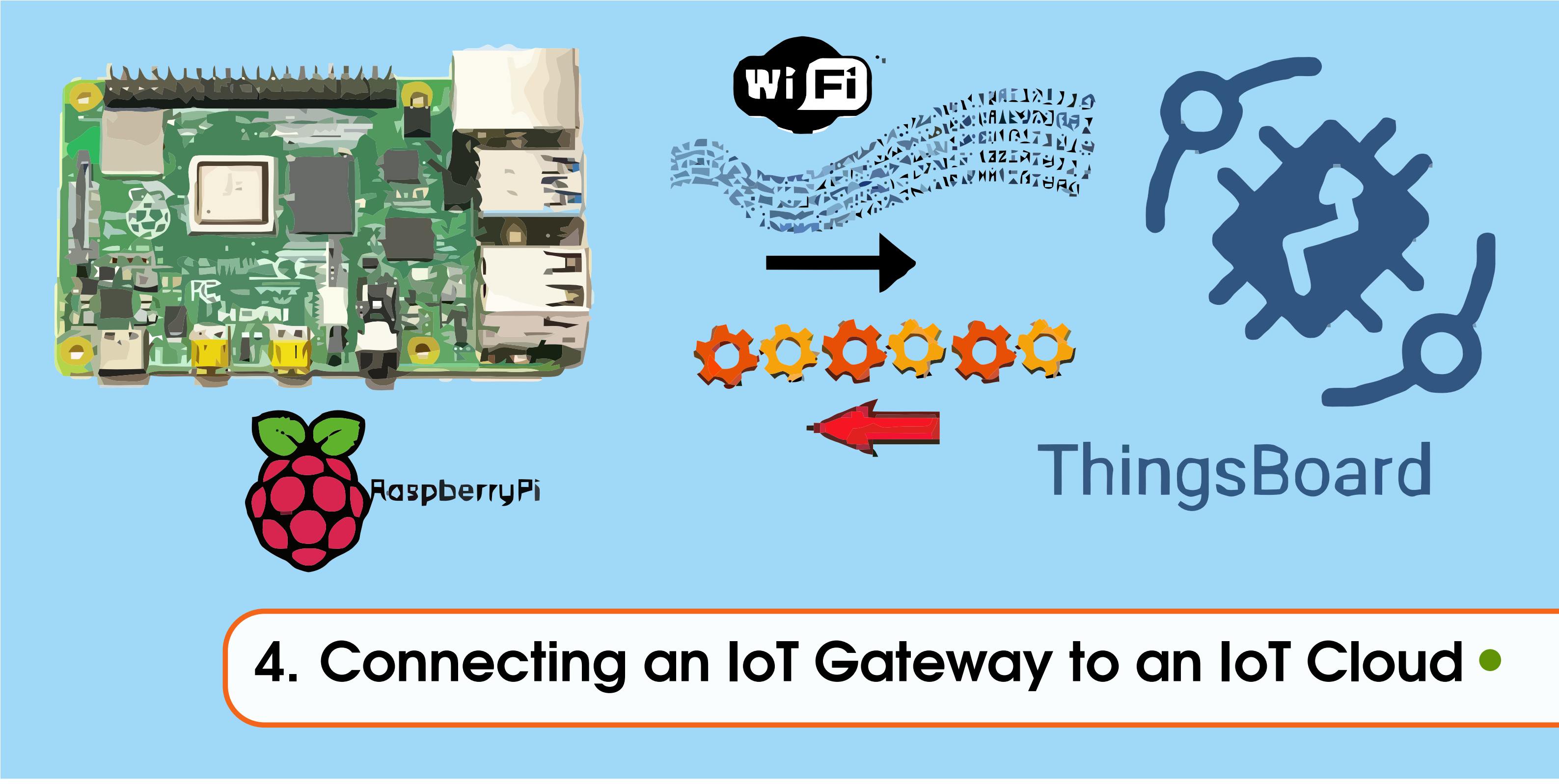
F.1 Scaling Up

- **Multiple Devices**: Each device has a unique token. Node-RED flows can handle sending different sets of telemetry to multiple devices or a single gateway device with sub-ids.
- **Edge Caching**: If the Pi goes offline, data can be cached locally and re-sent when connectivity returns (using an offline-first approach).

G. Best Practices and Future Exploration

- **Organized Node-RED Flows**: Keep sensor input, transformation logic, and MQTT output well-labeled for easier debugging.
- **Widget Variety**: Experiment with various widget types (line charts, bar charts, digital gauges) to find the clearest representation of your data.
- **Historical Analysis**: Enable time-series data storage in ThingsBoard to observe patterns or to conduct post-hoc analysis of sensor trends.
- **Event Triggers**: Explore ThingsBoard's "Rule Engine" to route device data to external services or to invoke custom scripts when certain conditions are detected.

Final Note.
By integrating **Node-RED** with **ThingsBoard** via **MQTT**, you gain a powerful framework for visualizing sensor data and orchestrating complex IoT workflows. Whether you're monitoring temperatures for a greenhouse or sounding alarms for hypothetical meteors, these fundamentals pave the way for more advanced solutions—automated alerting, predictive analytics, and robust multi-device ecosystems. Experimentation and incremental refinements will help you master this real-time, data-driven approach to IoT development.

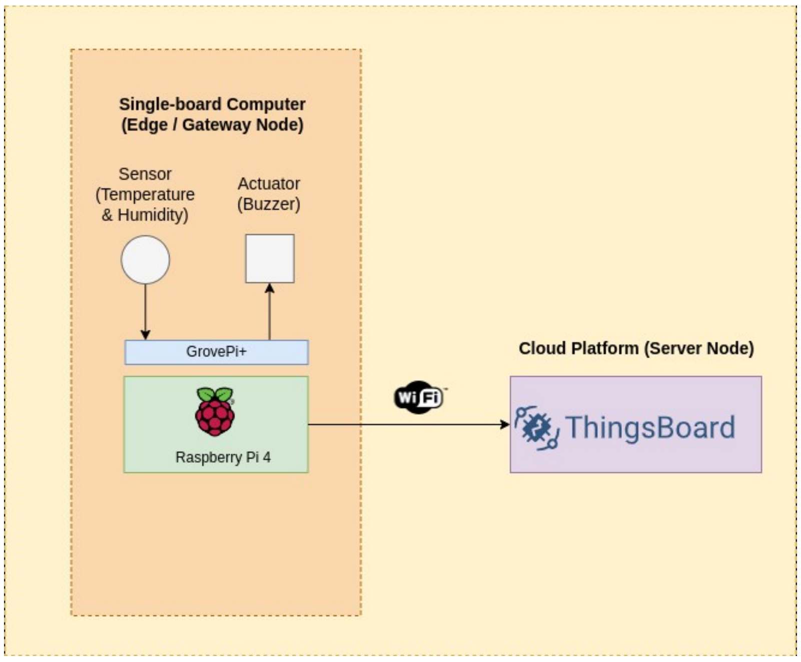


4. Connecting an IoT Gateway to an IoT Cloud

Objectives

- Learn how to configure an IoT cloud platform (ThingsBoard)
- Learn how to connect a gateway (Raspberry Pi) to an IoT cloud (ThingsBoard)
- Learn how to send data from an edge gateway to the cloud
- Learn how to receive commands/data from cloud back to edge

Lab Plan



In this lab, you will:

1. Create a new device in ThingsBoard to represent your gateway.

2. Write a Python script on the Raspberry Pi to send sensor data (Temperature & Humidity) to the cloud and receive remote commands (e.g., turning a buzzer on/off).
3. Use the MQTT library (`paho-mqtt`) to exchange messages with ThingsBoard.
4. Observe sensor telemetry in real time and add a switch widget to control the buzzer from the cloud.

Required Hardware Components

- **Raspberry Pi 4**
- **SD Card** with Raspberry Pi OS
- **Display and HDMI cable**
- **Keyboard and mouse**
- **Power supply**

Additionally, for sensors/actuators:

- **GrovePi+**
- **Temperature & Humidity Sensor (DHT11 or DHT20)**
- **Buzzer**

A. Prerequisites and Setup

1. **Lab 3 Completion Required:**
   You **must** have finished Lab **??**, where you configured Node-RED to publish temperature/humidity data to ThingsBoard. Ensure you have a stable connection and your data consistently reaches the cloud.

   > **— Why do this?.** This lab builds directly on the MQTT publishing setup from Lab **??**. If your Node-RED flows and device credentials are not properly configured, subsequent steps involving new Python scripts or data handling will fail or produce incomplete data on ThingsBoard.

2. **Check Sensor Wiring:**
   Make sure your hardware connections match the following:
   - **DHT Sensor** connected to port **D4**.
   - **Buzzer** connected to port **D8**.

   If Node-RED is currently running, stop it before proceeding to avoid conflicts:

```
node-red-stop

```

> Tips:
> - Double-check that the Grove cables are firmly inserted. A loose connection can result in no data or sporadic sensor readings.
> - If you receive errors about the port being unavailable, ensure Node-RED or other background services that use the same port (e.g., D4 for DHT sensor) are fully stopped.

3. **Install Python Scripts:**
   Download `cloud.py` from the GitLab repository and place it into your **/home/pi** directory.

— **Why do this?.** `cloud.py` is a Python script that establishes MQTT communication with ThingsBoard. It reads sensor data (temperature/humidity) and controls actuators (buzzer) while acting as a gateway. Keeping it in `/home/pi` standardizes script paths, making it easier to reference in commands.

4. **Set Permissions:**
   Run:

```
sudo chmod 755 cloud.py

```

   so that the script is executable by any user or service.

   Tips: If you rename the file or store it elsewhere, be sure to adjust the path accordingly when running the script (e.g., `./cloud.py` or `python3 /projects/cloud.py`). ■

5. **Install the Python MQTT Package:**
   The `paho-mqtt` library allows Python to communicate over MQTT with ThingsBoard:

```
pip3 install paho-mqtt

```

   — **Why do this?.** Without `paho-mqtt`, `cloud.py` cannot connect to the ThingsBoard MQTT broker. This library provides all necessary functions to publish messages and subscribe to topics. Once installed, your script can transmit sensor data and listen for server-side commands.

B. ThingsBoard Device Creation

6. **Create New Device (Non-Gateway).**
   Log in to `https://thingsboard.cs.cf.ac.uk/` using your group credentials or account details. In the **Devices** section:
   - Click the **+** icon to add a new device.
   - Name it something like **Buzzer Demo Device**.
   - **Important**: Do *not* select the "Gateway" option this time; let ThingsBoard automatically generate the device's token.

   — **Why do this?.** Previously, you created a device configured as a *gateway* for collecting data from multiple sensors. Now you'll set up a simpler, standard device to which you'll directly send telemetry (and potentially receive commands for controlling the buzzer).

7. **Update `cloud.py` with Your Access Token.**
   After creating the device, copy its auto-generated token from the ThingsBoard UI:
   - Go to **Devices** and locate your new "Buzzer Demo Device."
   - Click the device name to view its details.
   - Find **Credentials** or **Access Token**, and copy the token string.

   In `cloud.py`, locate the line that reads:

```
ACCESS_TOKEN = 'KV8ua9VXNu9cOQ8Op4DS'  # <== Insert your own token
```

   Paste your actual token in place of `KV8ua9VXNu9cOQ8Op4DS`.

Tips: If you rename the variable or file, be sure to keep references consistent in your script. Also, double-check you have no extra spaces or quotes around the token.

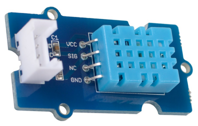
8. **Run the Python Script:**
   In a terminal on your Raspberry Pi, navigate to the directory containing `cloud.py` (usually `/home/pi`), and run:

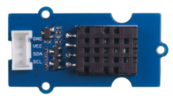
```
python3 cloud.py
```

   - If successful, you should see messages like `Session present: 0` and `Success temperature/humidity publishing` in your console.
   - Check ThingsBoard **Latest Telemetry** to confirm data (temperature, humidity, and buzzer state) is being received.

— **Which DHT sensor are you using?. DHT20** instead of DHT11?
Download `cloud_dht20.py` from GitLab if you have the DHT20 sensor (which communicates via I^2C).
Run Replace `cloud.py` with `cloud_dht20.py` in the above command, for example:

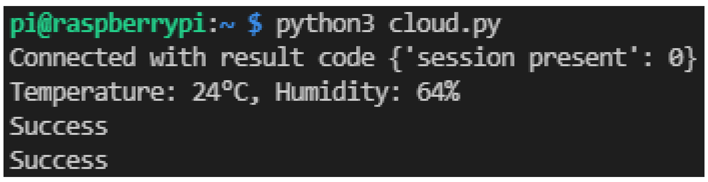
```
python3 cloud_dht20.py
```


(a) DHT11 (Digital Signal)


(b) DHT20 (I^2C)

C. Sending & Receiving Cloud Data

9. **Verify Python Output.**
   After running `cloud.py`, look for a console output similar to Figure 4.2:



```
pi@raspberrypi:~ $ python3 cloud.py
Connected with result code {'session present': 0}
Temperature: 24°C, Humidity: 64%
Success
Success
```

Figure 4.2: `cloud.py` Output

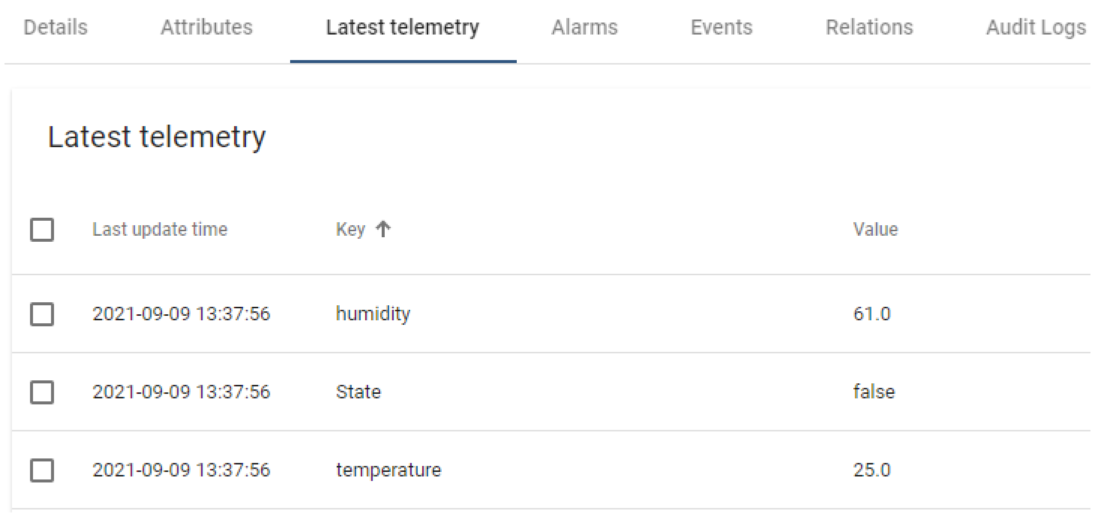
   - `{'session present': 0}` indicates your Raspberry Pi successfully established an MQTT connection to ThingsBoard.
   - `"Success <...> publishing"` lines confirm each piece of telemetry (temperature, humidity, and buzzer state) is being transmitted.

— **Why do this?.** Monitoring the script's console output helps confirm both MQTT connection status (`'session present': 0`) and the success of each publish event. If you see errors or no messages, it's easier to troubleshoot now—before checking ThingsBoard's console.

10. **Observe Data in ThingsBoard.**
    Return to ThingsBoard and open your device's details. In the **Latest Telemetry** tab (Figure 4.3), verify that:

- The `temperature` and `humidity` fields are updating.
- The buzzer state (`true/false` or `on/off`) also appears if `cloud.py` publishes it.

You can visualize this data by adding **time-series charts** or **gauge widgets** (similar to Lab 6). This way, you'll see how temperature/humidity change over time and whether the buzzer was triggered.



Latest telemetry

Last update time	Key ↑	Value
2021-09-09 13:37:56	humidity	61.0
2021-09-09 13:37:56	State	false
2021-09-09 13:37:56	temperature	25.0

Figure 4.3: ThingsBoard Device Telemetry

Tips:
- If telemetry data isn't showing up, re-check your `cloud.py` ACCESS_TOKEN, MQTT broker address, and port.
- Set up a **Card** or **Gauge** widget to see real-time numeric values, or a **Chart** widget for historical trends.
- Expand your usage by exploring the "Attributes" feature for static device properties (e.g., device location or model).

D. Controlling the Buzzer via RPC (Server to Edge)

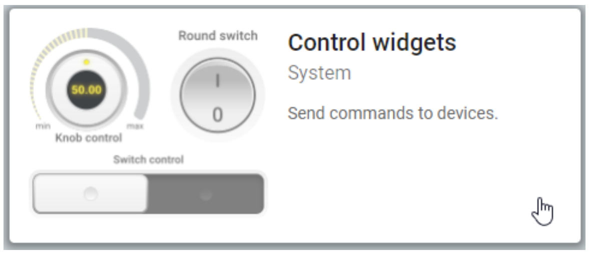
11. **Add a Widget for Buzzer Control.**
    Before making changes to your device's behavior, **stop** the Python script with `Ctrl + C`. This ensures the device is not actively sending or receiving data while we configure the dashboard.
    In your ThingsBoard dashboard:
    - Click the **pen icon** (bottom right) to enter edit mode.
    - Click the **+ icon** to create a new widget.
    - Click the **file icon** to choose from various widget bundles.

    Since we'll be using **Remote Procedure Calls (RPC)**, we need to pick a widget type that can send commands from the cloud to our buzzer.

    — Why do this?. RPC allows two-way communication. So far, you've been sending *telemetry* from edge to cloud. With RPC, the cloud (ThingsBoard) can instruct the edge device (Raspberry Pi) to change actuator states—like toggling the buzzer.

12. **Select Control Widgets.**

In the widget creation menu (Figure 4.4), choose a "control widget" suitable for sending RPC. This category includes switches, sliders, and other interactive elements that dispatch requests to your device.

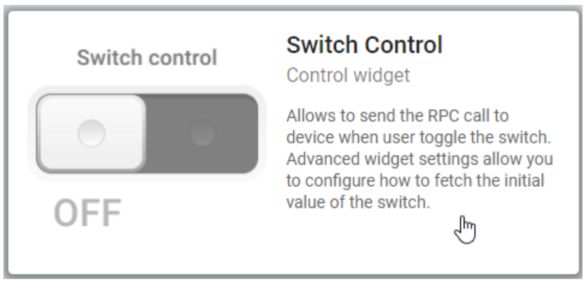


Control widgets
System
Send commands to devices.

Figure 4.4: Choosing a Control Widget

Tips:
- If you don't see *control widgets* listed, ensure you are in the right widget bundle or have installed all default ThingsBoard widgets.
- You can experiment with different control widgets—such as a button, switch, or slider—depending on how you want the buzzer to behave.

13. **Add the Switch Control Widget.**
As shown in Figure 4.5, select the *switch* widget. This widget sends a simple boolean `true/false` or `on/off` to your device.



Switch Control
Control widget
Allows to send the RPC call to device when user toggle the switch. Advanced widget settings allow you to configure how to fetch the initial value of the switch.

Figure 4.5: Switch Control

14. **Target Your Device.**

- Choose **Buzzer Demo Device** as the device you want to control.
- Set the "type" or "device type" to **Device Type**.
- Add the widget to your dashboard.
- If you see a "Request Timeout" error, that's normal if your device script (`cloud.py`) is not currently running to accept the RPC call.

Tips: If you have multiple devices, ensure you select the exact device associated with your `cloud.py` script. Mismatched devices/tokens can lead to silent failures when sending RPC commands.

15. **Re-run the Python Script.**
In a terminal on the Pi, restart the script:

```
python3 cloud.py

```

This time, watch both the ThingsBoard **dashboard** and the script's **console output**. The buzzer is connected to D8 by default in `cloud.py`; toggling the switch widget should now send an RPC command to your Pi to activate or deactivate the buzzer.

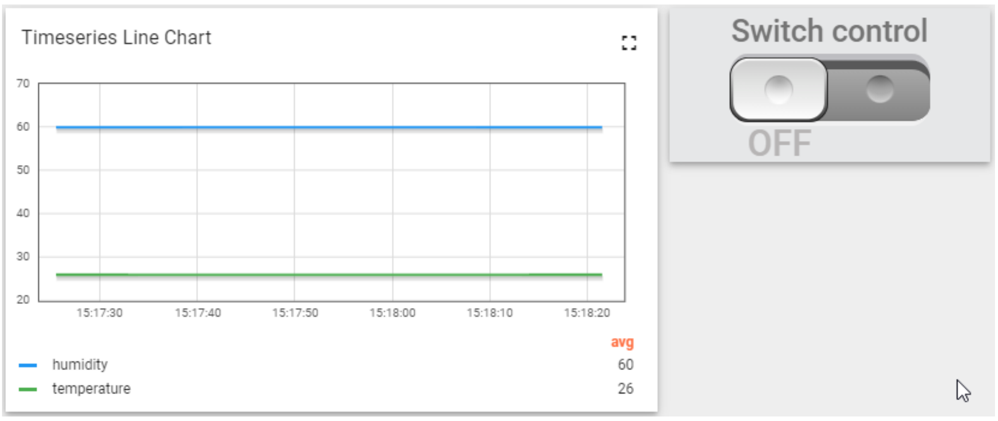


Figure 4.6: Dashboard with Buzzer Switch

— Why do this?. Seeing the switch widget on your dashboard and hearing the buzzer respond on the Pi side confirms two-way IoT interaction. Your device isn't just sending data (telemetry) to the cloud—it's also receiving commands (RPC) and updating actuators in real time.

Optional Final Task (Humidity Concern): Imagine you're worried about mold in your home. You can't constantly monitor humidity, so push humidity data to ThingsBoard. Have a **switch widget** or **slider widget** control some edge device (e.g., a fan) or an LED backlight. Your friend can log in from another computer to see humidity or turn on an alert.

You can now finalize your setup or close it out:

- **Stop the script** with `Ctrl + C`.
- Remove any Node-RED flows if you were testing it, or just leave them.
- Shut down the Pi if you're finished:

```
sudo shutdown now
```

Measuring Success

- **Cloud Telemetry**: Temperature/humidity data from `cloud.py` appear in the device's *Latest Telemetry*.
- **RPC Control**: Switch widget toggles the buzzer state in real time (BEEP or silent).
- **Python Output**: The script logs successful MQTT session, plus "Success" for sending/receiving data.

Further Reading

- **IoT Cloud and RPC in ThingsBoard**: `https://thingsboard.io/docs/user-guide/rpc/`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the theoretical underpinnings of connecting an edge gateway (e.g., Raspberry Pi) to an IoT cloud platform (e.g., ThingsBoard). By understanding these fundamentals, you will gain deeper insight into how data flows between local devices and the cloud, how MQTT publish/subscribe works, and how remote commands (RPC) facilitate two-way interactions in IoT systems.

A. IoT Gateway Concepts

An **IoT gateway** is a device or service that bridges local sensors/actuators to remote cloud platforms:

- **Local Connectivity:** Often communicates with sensors using protocols like I2C, SPI, or GPIO for reading data (e.g., temperature, humidity).
- **Network/Internet Connectivity:** Uses TCP/IP networking (Ethernet, Wi-Fi) to send data upstream to the cloud.
- **Transformation & Routing:** Translates raw sensor readings into a protocol like MQTT, then publishes these messages to an IoT cloud.

A.1 Why Use a Gateway?

- **Resource Management:** A Raspberry Pi can handle more complex processing or run multiple scripts compared to simpler microcontrollers.
- **Security Layer:** Gateways can locally encrypt or authenticate data before sending to the cloud.
- **Offline Processing:** If the cloud is unreachable, the gateway can store data locally or make local decisions temporarily.

A.1.1 Edge vs. Cloud Responsibilities

- **Edge Computing:** Filtering, preprocessing, or quick reactions (e.g., buzzer on/off).
- **Cloud Computing:** Long-term data storage, analytics, dashboards, or machine learning tasks.

B. Cloud Platforms (ThingsBoard)

ThingsBoard is an IoT platform that manages devices, telemetry, and visualizations:

- **Device Management:** Each device has a unique token and can send telemetry, receive commands, or store historical data.
- **Dashboard Widgets:** Graphs, gauges, switches, or sliders that provide real-time monitoring and control.
- **API/Protocol Support:** Typically supports MQTT, HTTP, and CoAP for data exchange.

B.1 Cloud Device Model

- **Non-Gateway Devices:** Directly connect to the cloud and represent a single physical device (like your Raspberry Pi with a single sensor).
- **Gateway Devices:** Aggregate data from multiple sensors or other devices, then pass them to the cloud under individual device “profiles.”

B.1.1 Access Tokens

- **Token-Based Authentication:** ThingsBoard auto-generates a token for each device. Your scripts must include this token to publish data successfully.
- **Security Best Practices:** Keep tokens private. If exposed, someone could spoof your device's identity on the cloud.

C. MQTT and Publish/Subscribe

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol widely used in IoT:

- **Broker-Based Architecture:** A central broker (ThingsBoard in this scenario) receives messages from publishers (your gateway) and distributes them to subscribers.
- **Topic Hierarchy:** Messages are sent to specific "topics" (e.g., `v1/devices/me/telemetry` for sending data to ThingsBoard).
- **QoS Levels:** MQTT can ensure at-least-once or exactly-once delivery, although many IoT scenarios use QoS 0 for reduced overhead.

C.1 Edge Publish/Cloud Subscribe

- **Publish Telemetry:** Your Python script uses the `paho-mqtt` library to send JSON data (temperature, humidity).
- **Subscribe to Commands:** The gateway can listen for incoming messages on a command topic (e.g., `rpc/request/+`) to control local actuators.

C.1.1 Data Format (JSON)

- **Key-Value Pairs:** E.g., `{"temperature": 24.5, "humidity": 60}`.
- **Extendable:** You can add fields like `"buzzerState": true` without changing the entire infrastructure.

D. Remote Procedure Calls (RPC)

RPC in IoT context allows the server (cloud) to invoke actions on the client (gateway):

- **Two-Way Communication:** Instead of just pushing data up, the gateway can receive commands to toggle a buzzer or update a local display.
- **Control Widgets in ThingsBoard:** A switch or slider widget can trigger an RPC to the gateway's topic, instructing it to perform an action.

D.1 Example: Buzzer Control

- **Cloud to Device:** When you click a switch in the ThingsBoard dashboard, a message is published to the device's RPC topic with the new "buzzer state."
- **Device Implementation:** Your Python script listens for RPC messages and, upon receiving them, sets a digital pin `HIGH` or `LOW`.

D.1.1 Handling Timeouts or Failures

- **Disconnects:** If the gateway loses connection, RPC calls will fail until reconnected.
- **Fallback Logic:** The script can handle unexpected values or default to a safe state (buzzer off, etc.).

E. Security and Access Control

- **TLS Encryption (Optional):** MQTT can be secured with SSL/TLS to protect data in transit.
- **Token Rotation:** If tokens are compromised, you can regenerate them in the ThingsBoard UI.
- **Network Security:** Gateways often sit behind firewalls or NAT; ensure inbound and outbound ports (e.g., 1883 for MQTT) are open or correctly forwarded.

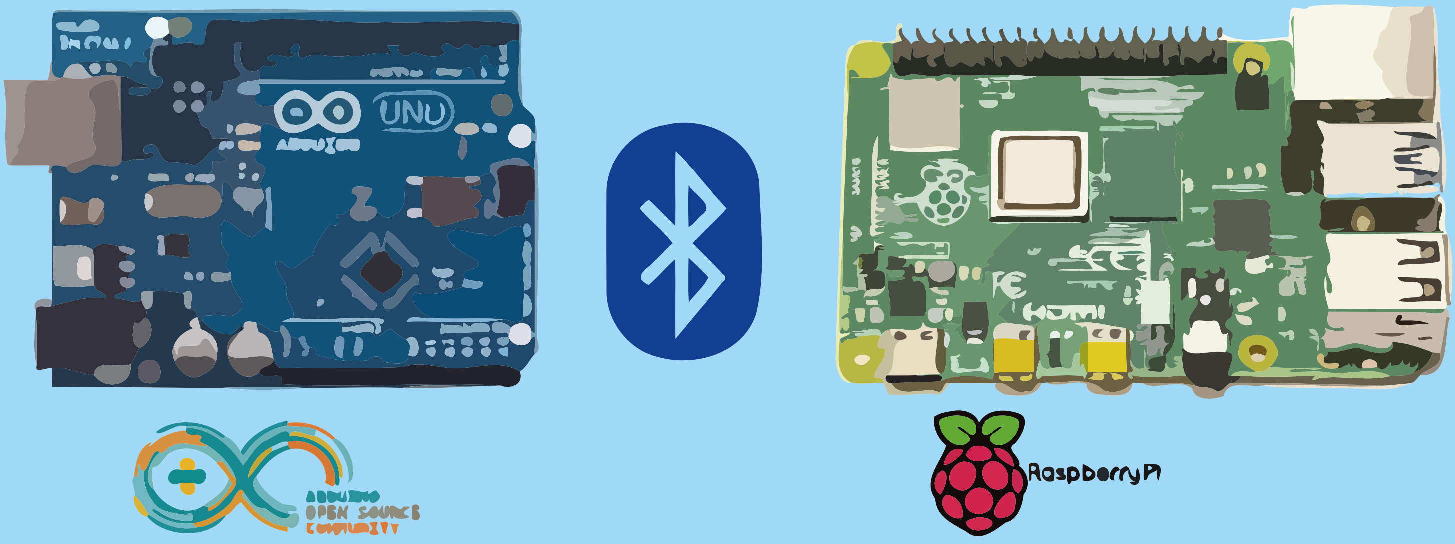
E.1 Edge Device Hardening

- **Regular Updates:** Keep your Pi's OS and packages updated (`sudo apt-get update && sudo apt-get upgrade`).
- **Secure SSH or Passwords:** If connecting remotely, ensure strong credentials or disable password authentication in favor of SSH keys.

F. Potential Extensions and Use Cases

- **Data Visualization:** Build more complex dashboards with historical trends, multi-series charts, or advanced analytics.
- **Alerts/Notifications:** Configure ThingsBoard to email or text you if temperature exceeds a certain threshold.
- **Multiple Devices/Gateways:** One Pi could act as a gateway for several microcontrollers, each reporting different sensors under separate IDs.
- **Edge-to-Edge Communication:** Use the cloud as a hub for two remote gateways to exchange messages (e.g., a Pi in one building controlling a device in another).

Final Note.
Understanding **IoT gateway architecture**, **MQTT publish/subscribe**, **remote procedure calls (RPC)**, and **cloud-based device management** is key to building scalable, interactive IoT solutions. As you refine your approach—adding authentication, encryption, advanced dashboards, or error handling—you'll gain the necessary skills to deploy secure, reliable, and feature-rich IoT applications.

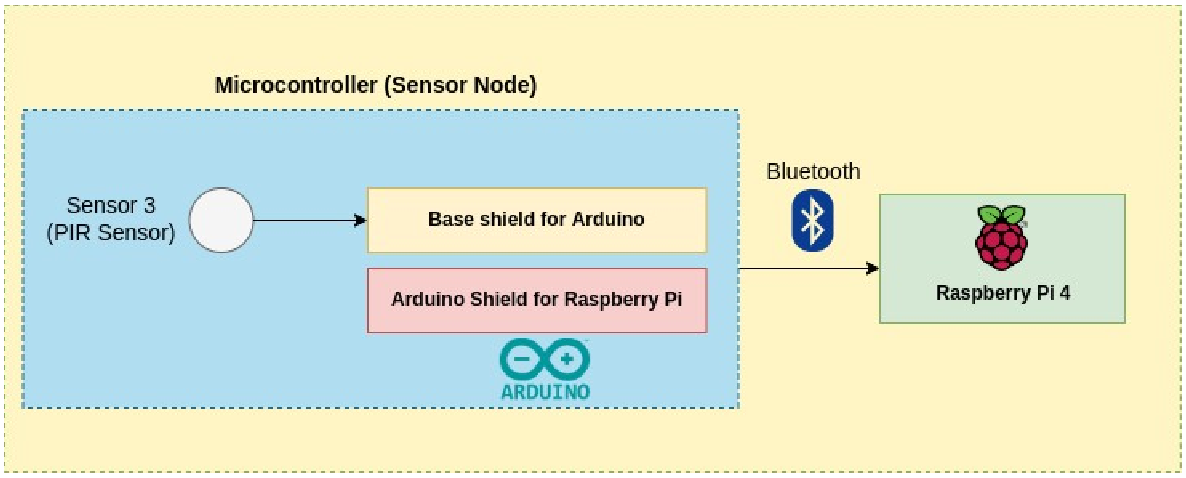


5. Connecting a Sensor Node to IoT Gateway

Objective

- Learn how to connect a sensor node (microcontroller-based) to an edge gateway node (Raspberry Pi single-board computer)
- Learn how to send data from the sensor node to the edge gateway
- Learn how to use Bluetooth for short-range communication

Lab Plan



In this lab, we will:

1. Configure a Raspberry Pi (gateway) with Bluetooth capabilities.
2. Program an Arduino (sensor node) to transmit motion sensor data over Bluetooth.
3. Pair the Pi (master) and Arduino (slave) via Bluetooth, and read data in Node-RED.
4. Optionally create a scenario (e.g., “Thief in the house”) using motion/light sensors.

Required Hardware Components

- **Raspberry Pi 4** with SD Card (Raspberry Pi OS)
- **Display and HDMI cable**
- **Keyboard and mouse**
- **Power supply**

Additionally, for sensor node:

- **Microcontroller** (e.g., Arduino)
- **Grove Base Shield for Arduino**
- **Grove PIR Motion Sensor** (connected to D2)
- **Grove Serial Bluetooth v3.0 or v3.01** (connected to D8)

A. Raspberry Pi Setup

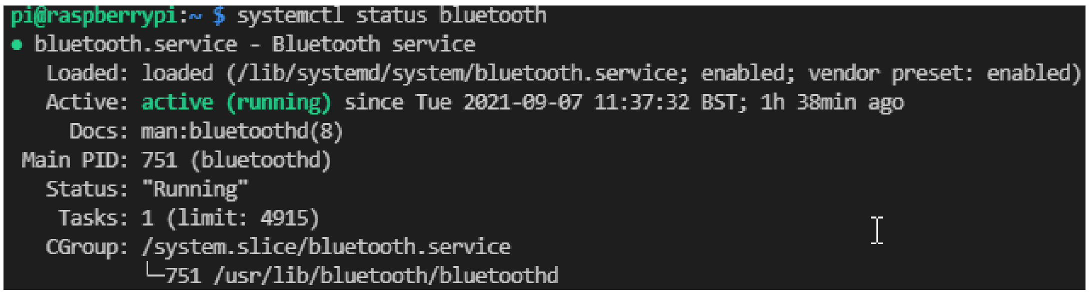
1. **Power Up and Connect to Wi-Fi.**
   Attach your Raspberry Pi to a monitor, keyboard, and mouse. Insert the microSD card (with Raspberry Pi OS), then plug in the power supply. Once the Pi boots to the desktop environment:
   - Click the network icon (top-right corner) to connect to **eduroam** or **CU-PSK** (or any other available Wi-Fi).
   - If you prefer using `raspi-config`, select `Network Options` and set the Wi-Fi SSID/-password there.

   — Why do this?. A reliable internet connection is essential for installing packages, troubleshooting, and configuring Bluetooth tools. By ensuring you're on **eduroam** or **CU-PSK**, you maintain consistent network access throughout the lab.

2. **Check Bluetooth Status.**
   Your Raspberry Pi has built-in Bluetooth hardware. In a terminal, type:

```
systemctl status bluetooth

```

   You should see an `active (running)` status, similar to Figure 5.1.



```
pi@raspberrypi:~ $ systemctl status bluetooth
● bluetooth.service - Bluetooth service
   Loaded: loaded (/lib/systemd/system/bluetooth.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2021-09-07 11:37:32 BST; 1h 38min ago
     Docs: man:bluetoothd(8)
 Main PID: 751 (bluetoothd)
   Status: "Running"
    Tasks: 1 (limit: 4915)
   CGroup: /system.slice/bluetooth.service
           └─751 /usr/lib/bluetooth/bluetoothd
```

Figure 5.1: Bluetooth Status

Tips:

- If you see `inactive` or `failed`, try:

```
sudo systemctl enable bluetooth
sudo systemctl start bluetooth
```

- Some Pi models require a full reboot to properly enable Bluetooth if it was disabled.

- Ensure no external Bluetooth dongle is plugged in; the Pi's onboard Bluetooth should be sufficient.

B. Arduino Programming (Sensor Node)

3. **Connect PIR Sensor & Bluetooth Module:**
   - **PIR Motion Sensor → D2**
     This digital port reads `HIGH` or `LOW` to indicate motion detection.
   - **Grove Bluetooth Module v3.0 → D8**
     This module will handle wireless serial communication between the Arduino "sensor node" and the Raspberry Pi "gateway."

   Next, plug your Arduino into a separate PC via USB. You will use the Arduino IDE on that computer to compile and upload sketches.

   — Why do this?. By attaching the PIR sensor and the Bluetooth module to digital pins (D2, D8), you create a simple input/output flow:
   - The PIR sensor provides motion data to the Arduino.
   - The Bluetooth module sends this data wirelessly to the Raspberry Pi gateway.

4. **Open Arduino IDE and Verify Board/Port.**

   - From the **Tools** menu, select the correct board (e.g., *Arduino Leonardo*) and COM port (Windows) or `/dev/ttyACM0` (Linux), `/dev/cu.usbmodem*` (macOS).
   - Upload an **empty sketch** to confirm that the IDE can communicate with your board. If you see "Done uploading" with no errors, you're ready to proceed.

   Tips: If you see an error like "board not found" or no ports listed:
   - Try a different USB cable or USB port (some cables are power-only).
   - Ensure you've selected "Arduino Leonardo" (or whichever model you're using) under **Tools** → **Board**.

5. **Upload the Bluetooth Code.**
   From the GitLab repository, download `arduino_bluetooth.ino` and open it in the Arduino IDE.
   - Review the code to understand how it initializes software serial communication, sends AT commands to configure the Bluetooth module, and reads the PIR sensor.
   - This sketch sets up a "Slave" role for the Bluetooth module (meaning it passively awaits a connection from a "master"—in our case, the Raspberry Pi).

   — Why do this?. The provided sketch handles both Bluetooth initialization (AT commands for naming/baud rate) and sensor reading (PIR on D2). Typing or reading through the code carefully helps you grasp each function's purpose and potential debugging points.

6. **Rename Your "Slave" ID (Optional).**
   Inside the `arduino_bluetooth.ino` sketch, find the line:

```
blueToothSerial.print("AT+NAMESlave");
```

   Change `"AT+NAMESlave"` to a unique name (e.g., `"AT+NAMEArduino123"`) if you want to distinguish your Bluetooth module from others in the lab. This name appears when scanning for Bluetooth devices.

Tips:

- Keep the name short and alphanumeric (no spaces or special characters).
- If other groups are using the same lab, pick something unique like "Team5Slave" or "CatSensorSlave" so you don't accidentally connect to the wrong module.

7. **Compile and Upload.**
   After customizing the name (if desired), click the **Verify** (check mark) icon, then the **Upload** (arrow) icon in the Arduino IDE.
   - Once uploaded successfully, open the **Serial Monitor**. The Grove Bluetooth module's LED should blink slowly, indicating it is **waiting for a connection**.
   - You may see some "OK" or "ERROR" messages in the serial monitor if you type AT commands. This is normal during testing.

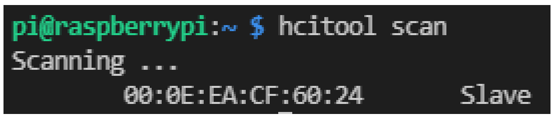
— Why do this?. A blinking LED signals that the Bluetooth module is discoverable but not yet paired with any master device. Leaving the serial monitor open lets you confirm the module's response to AT commands and see any debug prints from the PIR sensor (if coded to do so).

C. Pairing Raspberry Pi to Arduino via Bluetooth

8. **Scan for the Bluetooth Module on the Pi.**
   Switch back to your Raspberry Pi's terminal and run:

```
hcitool scan

```

This command looks for nearby discoverable Bluetooth devices. When the scan completes, look for your module's 48-bit MAC address (Figure 5.2), usually in the form `XX:XX:XX:XX:XX:XX`. Note it down; you'll need it in the following steps.



```
pi@raspberrypi:~ $ hcitool scan
Scanning ...
        00:0E:EA:CF:60:24       Slave
```

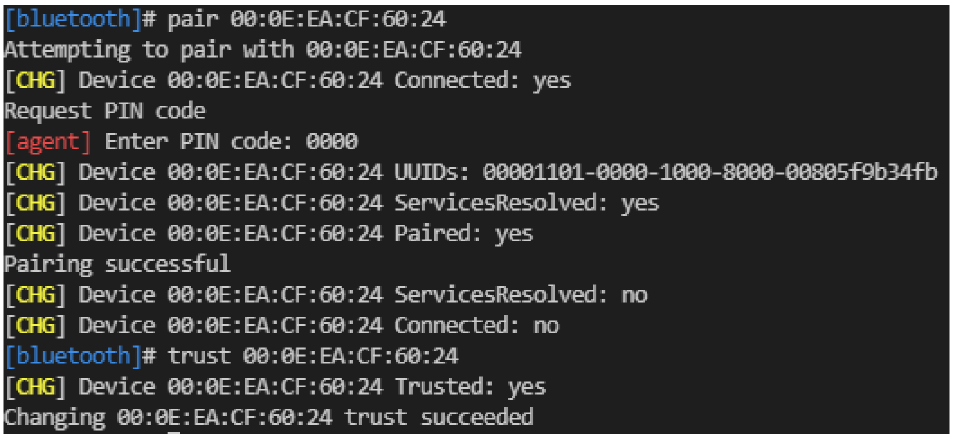
Figure 5.2: Finding the MAC Address of the Bluetooth

— Why do this?. The `hcitool scan` command reveals all Bluetooth devices broadcasting in your immediate area. By identifying your Arduino's MAC address, you ensure that subsequent pairing steps target the correct device.

9. **Pair and Trust the Device.**
   Next, enter the Bluetooth control shell by typing:

```
bluetoothctl
agent on
default-agent
scan on    // wait until you see your MAC
scan off
pair MAC_ADDRESS_HERE
trust MAC_ADDRESS_HERE
quit

```

Note: If prompted for a PIN, try `0000` or `1234`. Figure 5.3 shows a successful pairing output.



```
[bluetooth]# pair 00:0E:EA:CF:60:24
Attempting to pair with 00:0E:EA:CF:60:24
[CHG] Device 00:0E:EA:CF:60:24 Connected: yes
Request PIN code
[agent] Enter PIN code: 0000
[CHG] Device 00:0E:EA:CF:60:24 UUIDs: 00001101-0000-1000-8000-00805f9b34fb
[CHG] Device 00:0E:EA:CF:60:24 ServicesResolved: yes
[CHG] Device 00:0E:EA:CF:60:24 Paired: yes
Pairing successful
[CHG] Device 00:0E:EA:CF:60:24 ServicesResolved: no
[CHG] Device 00:0E:EA:CF:60:24 Connected: no
[bluetooth]# trust 00:0E:EA:CF:60:24
[CHG] Device 00:0E:EA:CF:60:24 Trusted: yes
Changing 00:0E:EA:CF:60:24 trust succeeded
```

Figure 5.3: Pairing Bluetooth

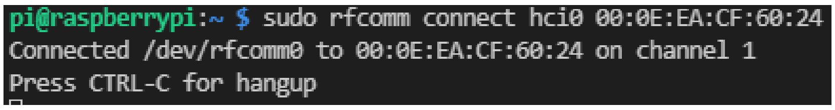
Tips:

- If the Pi fails to pair, confirm the Arduino's Bluetooth module is still in pairing mode (LED blinking).
- After pairing, `trust MAC_ADDRESS_HERE` ensures the Pi automatically reconnects without re-entering the PIN.
- You can exit `bluetoothctl` by typing `quit`.

10. **Connect to the Arduino.**
    To establish a serial connection (RFCOMM) with the module, type:

```
sudo rfcomm connect hci0 MAC_ADDRESS_HERE

```

If successful, you'll see an output similar to Figure 5.4, and the Grove Bluetooth LED will change from blinking to a continuous green light, indicating an active connection.



```
pi@raspberrypi:~ $ sudo rfcomm connect hci0 00:0E:EA:CF:60:24
Connected /dev/rfcomm0 to 00:0E:EA:CF:60:24 on channel 1
Press CTRL-C for hangup
```

Figure 5.4: Connecting Bluetooth

— Why do this?. The `rfcomm connect` command binds the Pi's Bluetooth interface (`hci0`) to your Arduino's MAC address, forming a wireless serial link. This "virtual serial port" allows the Pi to receive sensor data from the Arduino's PIR sensor or send commands back.

D. Node-RED Programming (Receiving Data)

11. **Open Node-RED on the Pi.**
    After you've established the Bluetooth connection (`rfcomm connect`), start Node-RED by:
    - Clicking the Raspberry icon (top-left) → **Programming** → **Node-RED**, *or*
    - Opening a terminal and typing:

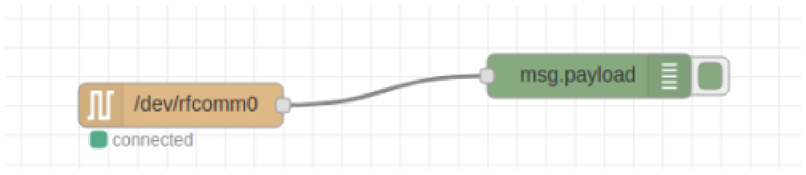
```
node-red
```

Once Node-RED is running, open a browser on the Pi and go to `http://localhost:1880` to access the flow editor interface.

> **— Why do this?.** Node-RED is a visual IoT workflow tool that simplifies hardware-software integration. By running it locally on the Pi, you can wire your new serial (Bluetooth) connection to a debug node without writing additional scripts.

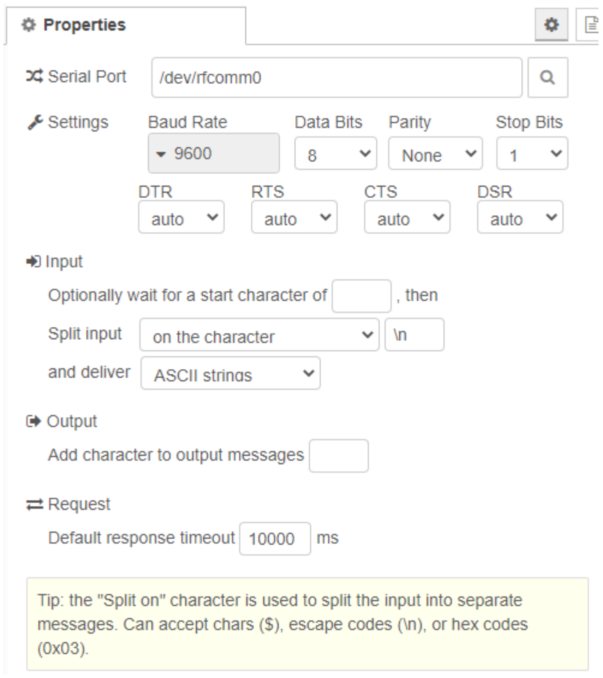
12. **Add a `serial in` Node & a Debug Node.**
On the left sidebar in the Node-RED editor, find the **serial in** node (under "Input" or "Advanced" categories, depending on your Node-RED version). Drag it onto the workspace, then add a **debug** node:
    - Connect the output (right side) of the `serial in` node to the input (left side) of the `debug` node.
    - This flow sends any incoming serial data from the Arduino (via Bluetooth) to the debug pane.

> **— Cannot Find `serial in` Node?.** **Check Manage Palette** Click the three parallel lines (top-right corner) of Node-RED, select **Manage Palette**, and look under the **Installed** or **Nodes** tab for `node-red-node-serialport`. If missing, you'll need to install or repair your Node-RED setup.
>
> **Download Repair Script** In some cases, you may need the `repair_nodered.sh` script from the LAB - General Solutions folder in GitLab. Save it to **/home/pi/Downloads/**.
>
> **Run the Script** For example:
>
> ```
> cd Downloads
> sh repair_nodered.sh
> ```
>
> This script:
> - Stops Node-RED (`node-red-stop`)
> - Downloads and installs a newer Node.js (v19.x)
> - Reinstalls Node-RED and **node-red-node-serialport**
>
> Once complete, restart Node-RED. If you still don't see `serial in`, try installing it manually via Manage Palette's **Install** tab.



13. **Configure the Serial Node.**
Double-click on the `serial in` node to open its properties:
    - **Serial Port**: Select `/dev/rfcomm0` (or whichever device was created by `rfcomm connect`).
    - **Baud Rate**: Match what your Arduino sketch uses (often `9600`), or any other speed specified in the `Serial.begin(...)` line of your code.
    - Leave other settings at defaults unless you require a specific data format or delimiter.

Click **Done** to confirm, similar to Figure 5.5.



Serial Port /dev/rfcomm0

Settings Baud Rate 9600 Data Bits 8 Parity None Stop Bits 1

DTR auto RTS auto CTS auto DSR auto

Input

Optionally wait for a start character of , then

Split input on the character \n

and deliver ASCII strings

Output

Add character to output messages

Request

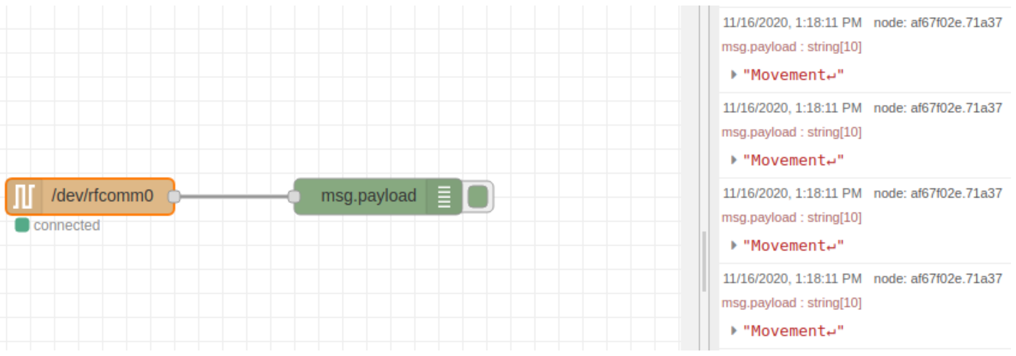
Default response timeout 10000 ms

Tip: the "Split on" character is used to split the input into separate messages. Can accept chars ($), escape codes (\n), or hex codes (0x03).

Figure 5.5: Serial Node Settings

— Why do this?. The `serial in` node receives data from the specified `/dev/rfcomm0` interface (Bluetooth RFCOMM link). Ensuring the baud rate is correct prevents garbled or unreadable data from the Arduino.

14. **Deploy and View Debug.**
Click the red **Deploy** button (top-right). If your Arduino is running the PIR sensor code and the Bluetooth connection (`rfcomm0`) remains active, you should:
    - See PIR-related messages (e.g., `"Motion detected!"` or `"No motion"`) in the Node-RED **Debug** panel.
    - The "Final Output" might look like Figure 5.6, confirming you have a live data stream from Arduino to Pi via Bluetooth.



```
11/16/2020, 1:18:11 PM  node: af67f02e.71a37
msg.payload : string[10]
▸ "Movement↵"
11/16/2020, 1:18:11 PM  node: af67f02e.71a37
msg.payload : string[10]
▸ "Movement↵"
11/16/2020, 1:18:11 PM  node: af67f02e.71a37
msg.payload : string[10]
▸ "Movement↵"
11/16/2020, 1:18:11 PM  node: af67f02e.71a37
msg.payload : string[10]
▸ "Movement↵"
```

Figure 5.6: Final Output: Node-RED Debug Panel Showing PIR Sensor Data over Bluetooth

Tips:
- If the debug panel is empty, re-check `rfcomm connect hci0 MAC_ADDRESS`, con-

firm the Arduino's LED (Bluetooth) is still solid, and verify you haven't disabled the debug node connection.
- If the Pi reboots or the session ends, you'll need to run `rfcomm connect` again to re-establish the link.

Measuring Success

- **Bluetooth Pairing**: Your Pi successfully pairs and trusts the Arduino module, and the `rfcomm connect` command shows a stable connection (solid LED on the Bluetooth module).
- **Node-RED Data**: The `serial in` node receives PIR motion sensor readings, clearly visible in the debug panel.
- **Arduino Reactions**: Whenever you wave in front of the PIR sensor, motion updates are reflected in Node-RED almost immediately, confirming end-to-end communication.

Use Case Scenario (Optional): Imagine your home is dark at night (light sensor reads a low value) and a "thief" triggers the motion sensor. In this scenario, Node-RED should log a message saying: *"There is a thief!"*

- **Add a Second Sensor (Light Sensor):**
  Connect a Grove light sensor to an **analog port** on your Arduino (e.g., A3). Modify your Arduino code (or create a new one) to read and transmit the light sensor value over Bluetooth alongside the PIR motion data.
- **Use a Function Node in Node-RED:**
  After receiving both `lightValue` and `motionDetected` in separate `serial in` nodes (or from a combined string/JSON if you merged them in Arduino code), feed them into a **function** node. For example:

```
if (lightValue < 200 && motionDetected) {
   msg.payload = "There is a thief!";
    return msg;
} else {
    return null;
}
```

- **Testing the Scenario:**
  - **Dim the Light Sensor**: Cover the light sensor to simulate nighttime.
  - **Trigger the PIR**: Wave your hand or walk by the sensor to generate motion data.
  - **Node-RED Output**: If both conditions are true (dark + motion), the function node sends a message with `"There is a thief!"` to the debug panel or another output node.

— **Why It Matters?.** This basic conditional logic illustrates how you can combine multiple sensors to detect unusual conditions—in this case, motion in a dark environment. In real-world IoT security systems, this concept can be extended to trigger alarms, send notifications, or even lock doors automatically.

Further Reading

- **Arduino Software Serial Reference**:
  `https://www.arduino.cc/en/Reference/softwareSerial`

Theory Deep Dive: Underlying Principles and Concepts

Theory Deep Dive: Sensor Node to Gateway Communication

This section delves into the underlying principles of connecting a microcontroller-based sensor node to an edge gateway (e.g., Raspberry Pi). You will learn how short-range wireless communication (Bluetooth) fits into IoT architectures, how data flows from sensors to Node-RED, and how event-driven logic can be implemented in such systems.

A. The Sensor Node Concept

A **sensor node** is typically a low-power microcontroller board (e.g., Arduino) equipped with one or more sensors (like a PIR motion sensor). Key attributes:

- **Dedicated Task:** Continuously read from sensors and transmit data to a gateway.
- **Resource Constraints:** Limited CPU speed, memory, and power (often battery operated).
- **Communication Interface:** Could use wired (UART, I2C) or wireless (Bluetooth, Wi-Fi, LoRa, etc.).

A.1 Why Use a Separate Sensor Node?

- **Distributed Sensing:** Place sensor nodes in remote or inaccessible areas; only the gateway needs full network/Internet connectivity.
- **Flexibility and Modularity:** You can add multiple independent sensor nodes, each focusing on specific tasks (light monitoring, motion detection, etc.).
- **Reduced Load on Gateway:** Offload some data acquisition or preprocessing from the main Raspberry Pi gateway.

A.1.1 Microcontroller vs. SBC for Sensing

- **Microcontroller (Arduino)**: Simpler, cheaper, often lower power consumption; ideal for continuous sensing tasks.
- **Single-Board Computer (Raspberry Pi)**: Runs a full OS, more powerful, but higher cost and power usage. Often acts as a gateway or local server.

B. Bluetooth Basics for IoT

Bluetooth is a short-range wireless protocol commonly used for device pairing and data transfer:

- **Range:** Typically up to 10 m (Class 2), though some devices can reach 100 m (Class 1).
- **Profiles:** SPP (Serial Port Profile) emulates a virtual COM port for straightforward data transmission.
- **Pairing and Bonding:** Devices exchange a PIN or key to establish a trusted relationship.

B.1 Bluetooth 3.0 Modules on Arduino

- **Serial Communication:** The Grove Bluetooth v3.0 module typically uses TX/RX lines (pins D8, D9, etc.) to replicate a UART link.
- **AT Commands:** Some modules allow configuration (e.g., renaming the device) via AT commands at startup.

- **Master/Slave Roles:** In this lab, the Pi acts as the master, initiating the connection; the Arduino module is a slave, waiting for requests.

B.1.1 Connection Flow

- **1) Discovery:** The gateway (Pi) scans for nearby devices.
- **2) Pairing:** An exchange of keys or PIN (`0000`, `1234`, etc.).
- **3) RFCOMM Binding:** A virtual serial port (`/dev/rfcomm0`) is created for data.

C. Data Transmission from Arduino to Pi

Serial data flow involves sending bytes over the RX/TX lines:

- **Arduino Code:** Typically uses `Serial.println()` or SoftwareSerial to push sensor data (e.g., "MOTION DETECTED").
- **Gateway Process (Pi)**: Reads these bytes as if they were coming from a standard serial port (`dev/rfcomm0`).

C.1 Event-Driven Sensor Reporting

- **Interrupt/Triggered Reads:** The PIR sensor can set a digital pin HIGH when motion is detected, and the Arduino immediately sends a message.
- **Polling Approach:** The Arduino might check the PIR pin in its `loop()` and send data only when it changes state.

C.1.1 Data Protocol on Serial

- **Plaintext Strings:** E.g., `motion=1\n` or `motion=0\n`
- **JSON-Like Format (Optional):** E.g., `{motion: 1, sensorID: 'PIR_1'}`

D. Node-RED Integration at the Gateway

Node-RED is a flow-based tool for rapidly connecting hardware, APIs, and online services:

- **Serial In Node:** Listens on `/dev/rfcommX` for incoming sensor data.
- **Debug Node:** Displays the sensor data in real time in the Node-RED Debug sidebar.
- **Optional Logic/Filtering:** A `function` node can parse text, filter events, or trigger further actions (e.g., email alerts).

D.1 Using Node-RED with Bluetooth

- **Palette Management:** Ensure `node-red-node-serialport` is installed for serial capabilities.
- **Deployment Cycle:** After configuring the `serial in` node (baud rate, port), click **Deploy** to apply changes.

D.1.1 Handling Disconnections

- **Re-Connection Script:** If `rfcomm connect` drops, you may need to rerun it or automate via a startup script.
- **Error Handling in Node-RED:** The `serial in` node might emit errors if the port is suddenly unavailable.

E. Designing a Use-Case Scenario

- **Alarm System:** If the PIR detects motion while a "night mode" flag is set, Node-RED can log "Intruder Alert!" or trigger a buzzer connected to the Pi.

- **Light-Triggered Logging:** Combine a light sensor so the node only sends motion data if it's dark; reduces false positives in well-lit areas.
- **Power Constraints:** If running the Arduino on battery, consider sleeping or reducing the sensor poll rate to conserve energy.

E.1 Extending the Lab

- **Multiple Sensor Nodes:** Each with a unique Bluetooth module. The Pi scans, pairs, and listens for data from each node in turn.
- **Short-Range Mesh Networks:** Explore BLE (Bluetooth Low Energy) or Zigbee for a multi-node setup if your application requires lower power or more flexible topologies.

F. Security Considerations in Bluetooth IoT

- **PIN or Passkey**: The default `0000` or `1234` can be changed on some modules for added security.
- **Visibility and Discovery**: The node might remain in *non-discoverable* mode after pairing to prevent unauthorized connections.
- **Encryption**: Bluetooth supports link-level encryption, but simpler modules may have limited capabilities. For critical data, consider additional encryption on top of the serial data.

F.1 Node-RED Access Control

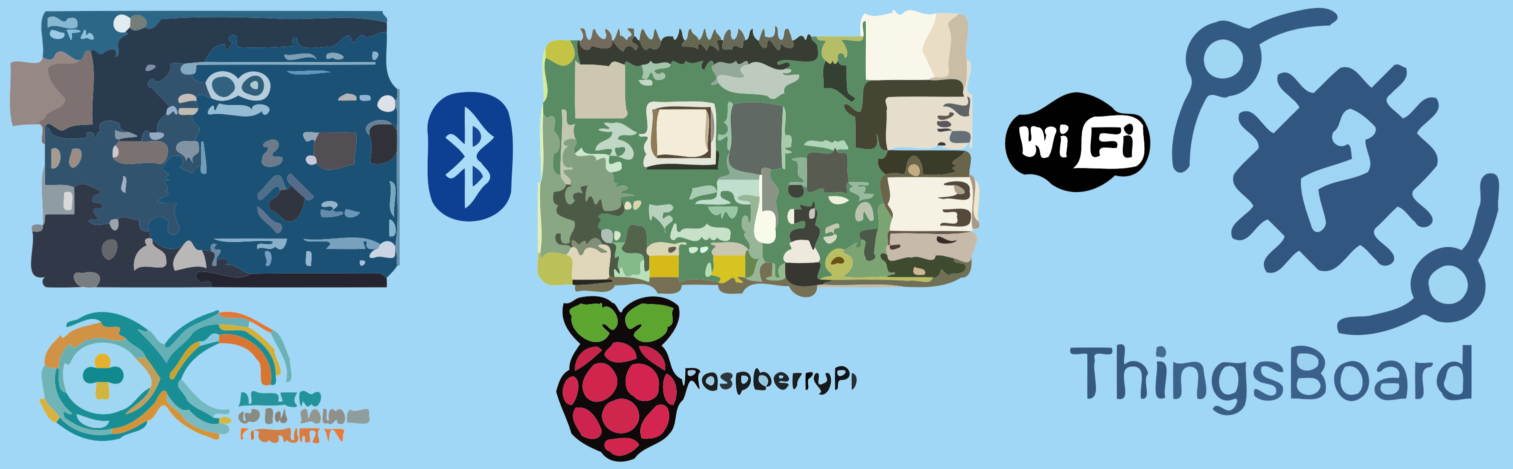
- **Password-Protect the Editor**: For production environments, enable adminAuth in Node-RED's settings to prevent unauthorized users from altering flows.
- **System-Level Security**: Keep your Raspberry Pi updated, use strong SSH credentials, and avoid running as `root` unnecessarily.

G. Final Thoughts and Next Steps

- **Automation Potential**: Once the data is in Node-RED, you can easily integrate with cloud services or local notifications.
- **Scaling Up**: For larger systems, replicate this approach with more nodes, or move to more robust protocols (MQTT over Wi-Fi, for example).
- **Battery Optimization**: Explore lower-power libraries or hardware modifications if your sensor node must run unplugged for extended periods.

Final Note.

Mastering **Bluetooth communication** between an Arduino sensor node and a Raspberry Pi gateway is a valuable skill for short-range IoT solutions. You've seen how Node-RED can integrate serial data, enabling rapid prototyping of event-driven behaviors like alarms or notifications. In real-world deployments, you may extend these concepts to advanced security measures, multiple sensor nodes, or more energy-efficient protocols. Experimentation with different sensors and logic flows will enhance both your technical and creative abilities in embedded IoT development.

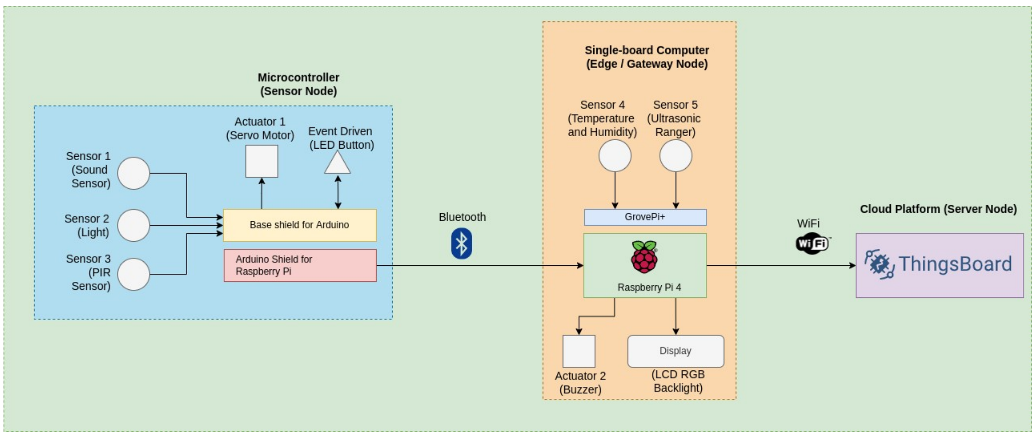


6. End to End Full Stack IoT Development

Objectives

- Learn how to connect a sensor node to an edge gateway node
- Learn how to send data from the sensor node to the edge gateway (over Bluetooth)
- Learn how to develop an end-to-end IoT stack (edge–fog–cloud)

Lab Plan


Microcontroller (Sensor Node)
Sensor 1 (Sound Sensor)
Sensor 2 (Light)
Sensor 3 (PIR Sensor)
Actuator 1 (Servo Motor)
Event Driven (LED Button)
Base shield for Arduino
Arduino Shield for Raspberry Pi
Bluetooth
Single-board Computer (Edge / Gateway Node)
Sensor 4 (Temperature and Humidity)
Sensor 5 (Ultrasonic Ranger)
GrovePi+
Raspberry Pi 4
Actuator 2 (Buzzer)
Display
(LCD RGB Backlight)
WiFi
Cloud Platform (Server Node)
ThingsBoard

In this lab, you will combine knowledge from previous labs to create a complete IoT stack:

1. Sensor Node (Arduino) sends data to Edge Gateway (Raspberry Pi) via Bluetooth.
2. Edge Gateway processes and/or forwards data to Cloud (ThingsBoard).
3. Cloud (ThingsBoard) displays dashboards, triggers alarms, or controls actuators.

Required Hardware Components

- **All hardware from Labs 1, 2, 3, and 4** (except Grove-Servo)

- This typically includes:
  - Raspberry Pi (with Node-RED, Bluetooth, etc.)
  - Arduino microcontroller with Grove Base Shield
  - Sensors: Temperature & Humidity (DHT), PIR, Sound (loudness), Ultrasonic, LED button, buzzer, LCD, etc.

A. Overview and Preparation

1. **Recall Previous Labs.**
   This lab combines key concepts from:
   - **Lab 1:** Arduino microcontroller programming and basic sensor reading.
   - **Lab 2 & 3:** Raspberry Pi setup, Node-RED flows, and sending data to the cloud (ThingsBoard).
   - **Lab 4:** Bluetooth communication between an Arduino sensor node and a Pi gateway.

   By integrating all these, you'll create an **end-to-end** IoT solution. Sensor data flows from Arduino to Pi (via Bluetooth), then Node-RED forwards it to ThingsBoard, and you can visualize or act upon it.

   > **— Why do this?.** Real IoT projects often involve multiple components—microcontrollers, single-board computers, sensors, actuators, and cloud dashboards. Understanding how to unify these elements into a single pipeline is essential for deploying complete, production-ready systems.

2. **Set Up Basic Connections.**

   - **Arduino (Sensor Node)**: Connect any sensors/actuators (e.g., PIR sensor, light sensor, buzzer) as done in earlier labs.
   - **Raspberry Pi (Gateway)**: Runs Node-RED, forwarding data to ThingsBoard (using MQTT or HTTP). Communicates with the Arduino via **Bluetooth**, just like in Lab 4.
   - **Optional Desktop/Monitor**: For viewing the Pi's Node-RED interface and verifying data on the ThingsBoard dashboard.

   > **Tips:**
   > - Double-check your Bluetooth pairing steps from Lab 4 if you haven't paired the Arduino and Pi in a while; you may need to re-run `hcitool scan` and `bluetoothctl` commands.
   > - Make sure Node-RED is still configured to send data to your existing ThingsBoard device credentials if you plan to reuse a previous device or gateway.
   >
   > ■

3. **(Optional) No Servo.**
   For simplicity, you do **not** need a Grove-Servo in this lab. If you want to practice controlling an actuator, you can integrate a servo as an advanced extension:
   - Use one of the PWM digital ports (e.g., D5) on Arduino.
   - Add servo logic in your Arduino code to react to sensor data or Node-RED commands.

   > **— Why skip the Servo?.** Servos can complicate wiring and code. The main focus is on **data flow** (sensors $\rightarrow$ Pi $\rightarrow$ cloud). Once you master this pipeline, adding a servo is straightforward—but it's optional if you only need sensor monitoring and logging.

B. First Task: Temperature Monitoring System

4. **Create a Temperature/Humidity + Buzzer + LCD Setup.**
   Assemble the following components on your Arduino:
   - **DHT Sensor (Temperature & Humidity)** — attaches to a digital port (e.g., D4) or I^2C (if using DHT20).
   - **LCD Backlight Display** — connects to an I^2C port on your Grove Base Shield (e.g., I2C-1 or I2C-2).
   - **Buzzer** — uses a digital port (e.g., D8) to sound alarms.

   The Arduino will read temperature/humidity locally, then send these values via Bluetooth to the Raspberry Pi.

   — Why these components?. This setup illustrates a classic environmental monitoring system. The DHT sensor measures conditions, the LCD provides instant local feedback, and the buzzer offers an auditory alert. Together, they form a mini “warning station” that can also push data to the cloud.

5. **Desired Behavior:**
   In this scenario:
   - If **temperature** exceeds **25°C**, show a *warning* (e.g., “HIGH TEMP!”) on the LCD display.
   - If **humidity** reaches **80%** or higher, **activate the buzzer**.
   - If *both* conditions occur simultaneously, the LCD warning and buzzer alarm should trigger *together*.

   Tips:
   - Start by printing raw temperature/humidity to the LCD so you can confirm your DHT readings are accurate.
   - Write the logic in either Node-RED (with function nodes) *or* in Arduino code. Decide which side handles threshold comparisons.

6. **Send Data to ThingsBoard.**
   Use Node-RED’s MQTT integration (from Lab 6) to send **temperature** and **humidity** to your ThingsBoard device. For visualization:
   - **Time-series charts** let you observe how conditions change over minutes/hours.
   - **Cards widgets** can display live numerical values for quick monitoring.
   - You can also configure thresholds on ThingsBoard to highlight when temperature or humidity surpass normal values.

   — Why do this?. By publishing data to ThingsBoard, you have an off-device record of environmental conditions and alerts. This approach aligns with real IoT systems where sensor data is aggregated in the cloud for analytics, logging, or remote notifications.

7. **Implementation Tips:**

   - **Arduino side**: Continuously read the DHT sensor. Send data (and optionally threshold triggers) over Bluetooth to the Pi.
   - **Raspberry Pi side**:
     - In **Node-RED**, create flows that receive Bluetooth (serial) messages.
     - Compare temperature/humidity with your set thresholds (25°C, 80%) in a `function`

node or separate logic nodes.
- **If above thresholds**: Use GPIO or GrovePi+ nodes to drive the buzzer or send commands back to the Arduino for the buzzer pin.
- **LCD updates**: You can write to the LCD in Node-RED (using a daemon node running a Python `lcd.py` script) *or* let the Arduino code handle updates if it receives threshold triggers from Node-RED.
- **Publish to ThingsBoard** via `mqtt out` (or the method from Lab 6).

Tips:
- Test each part separately: (1) DHT sensor readings, (2) buzzer control, (3) LCD output, and (4) data push to the cloud. Integrate them step by step.
- A short `delay()` in the Arduino code (or a `time.sleep()` in Python) can help stabilize sensor readings before sending them.
- If you find the threshold logic easier on the Arduino, do that. If you prefer Node-RED's visual approach, parse the sensor data in Node-RED and trigger events from there.

C. Second Task: Home Security System

8. **Configure Sensors for Suspicious Activity.**
   - **PIR sensor** detects motion. - **Sound (loudness)** sensor picks up loud noises. - **Ultrasonic Ranger** for distance-based triggers (optional).
9. **LED Button for Mode Switching.**
   Program two modes:
   (a) **Waiting Mode**: Display "Waiting" on LCD. The system logs sensor activity but does not alarm.
   (b) **Armed Mode**: If motion or loud noise is detected, the **buzzer** rings. A Node-RED flow sends a message to ThingsBoard, indicating a break-in.
10. **Advanced Feature: Remote Disable.**
    Upload the Node-RED flow to cloud or connect an **RPC widget** on ThingsBoard so you can *remotely* disable the system after it's been activated. This way, from any browser, you can see the security state and silence the buzzer if needed.
11. **Implementation Tips:**

    - **Arduino** reads sensors (PIR, loudness) over relevant pins.
    - **LED button** (D3 port) toggles states from *Waiting* to *Armed*.
    - **Pi Node-RED** receives sensor data via Bluetooth. If in *Armed* mode, triggers *buzzer* and updates ThingsBoard with an "Intruder Alert!" event.

Measuring Success
- **Temperature Monitoring**
  - **LCD Warning Above 25°C**: When the temperature exceeds your set threshold, the display should clearly indicate a high-temp alert (e.g., "HIGH TEMP!").
  - **Buzzer on High Humidity**: Once the humidity hits or surpasses 80%, the buzzer should sound and remain active until conditions improve or the system logic silences it.
  - **ThingsBoard Logging**: All temperature and humidity readings should flow contin-

uously into your ThingsBoard dashboard, updating graphs or cards in real time.

- **Home Security System**
  - **Waiting Mode (LCD Feedback)**: LCD reads "Waiting," no alarms or alerts, even if the sensors detect motion/noise.
  - **Armed Mode (Alarms & Cloud Alerts)**: Any motion or loud noise triggers a buzzer alarm. Node-RED also sends an "Intruder Alert!" or similar message to your ThingsBoard device.
  - **Remote Disable**: Optionally, you can toggle the system mode from a ThingsBoard or Node-RED dashboard widget, confirming that commands from the cloud silence the alarm in real time.
- **Full Stack Operation**
  - **Arduino $\leftrightarrow$ Pi (Bluetooth)**: Verified bidirectional data flow; sensors on Arduino, logic on Pi. The Arduino properly reports sensor values, while Pi can respond or send control signals (e.g., to toggle the buzzer).
  - **Pi (Node-RED) $\leftrightarrow$ ThingsBoard (MQTT/RPC)**: Data successfully arrives at ThingsBoard, where you can monitor live readings or trigger remote actions (e.g., RPC to disable alarms).
  - **End-to-End Visibility**: Both local feedback (LCD/buzzer) and remote dashboards (ThingsBoard) reflect real-time sensor and alarm states, forming a cohesive "edge-to-cloud" system.

■

Once you confirm these behaviors, you have successfully implemented an end-to-end IoT stack bridging sensor node (Arduino), edge/fog node (Pi), and cloud (ThingsBoard). Feel free to experiment with more sensors or advanced triggers.

— Further Reading
- `https://www.arm.com/glossary/iot-cloud`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the theoretical underpinnings of creating a complete IoT stack, from sensor node to cloud. By understanding these architectural layers—sensor node (Arduino), edge/fog node (Raspberry Pi), and the IoT cloud (ThingsBoard)—you can design end-to-end solutions that gather data, process it locally, and provide remote monitoring or control.

A. Edge–Fog–Cloud Architecture

A full-stack IoT system typically spans three tiers:

- **Sensor Node (Edge)**: Low-power microcontroller that directly interfaces with sensors/actuators.
- **Gateway or Fog Node (Edge Computing)**: A more capable device (e.g., Raspberry Pi) that aggregates data from multiple sensors, offers local intelligence, and handles connectivity to the cloud.
- **Cloud Platform**: Provides large-scale data storage, dashboards, analytics, and remote control.

A.1 Why Multiple Layers?

- **Bandwidth Efficiency**: Filtering or preprocessing at the edge reduces raw data sent to the cloud.
- **Local Autonomy**: When the cloud is unreachable, the gateway can still respond to events (e.g., triggering a buzzer).
- **Scalability**: Distributing functionality across nodes prevents a single point of failure and allows many sensor nodes to connect through a single gateway.

A.1.1 Fog vs. Edge Terminology

- **Fog Computing**: Often used interchangeably with “edge” computing. Fog nodes might be slightly more powerful or handle more complex tasks than a simple “edge” device.
- **Edge Node**: A device that interfaces directly with sensors/actuators, usually with limited resources.

B. Sensor Node Fundamentals (Arduino)

Microcontroller-based sensor nodes read environmental data (temperature, humidity, motion, etc.) and transmit results:

- **Low-Power Operation**: Many nodes can run on batteries, sleeping between measurements to conserve energy.
- **Dedicated Functionality**: Typically programmed in C/C++ with libraries for each sensor (e.g., DHT library, PIR read, etc.).
- **Communication Protocols**: UART, I2C, SPI, or wireless (Bluetooth, Wi-Fi shield, etc.).

B.1 Handling Multiple Sensors

- **Pin Assignments**: Each sensor or actuator uses specific pins (e.g., D2 for PIR, D8 for Bluetooth).
- **Sensor Fusion**: The node might combine data (e.g., *temperature* and *humidity*) or perform small computations before sending it to the gateway.

B.1.1 Event-Driven Alerts

- **Threshold Checking**: The Arduino can buzz an alarm if temperature > 30°C, *even without the gateway’s command*.
- **Interrupts**: Some sensors (like PIR) can trigger an `interrupt` routine for quick response.

C. Gateway/Fog Node (Raspberry Pi)

A **Raspberry Pi** typically runs a full OS (Raspberry Pi OS), making it an ideal gateway:

- **Local Processing (Edge Intelligence)**: The Pi can parse data, check thresholds, or run Node-RED flows.
- **Protocol Translation**: Converts Bluetooth data from the Arduino into MQTT messages for the cloud.
- **User Interfaces**: Can be directly connected to a monitor/keyboard/mouse, or accessed via SSH/VNC for headless operation.

C.1 Bluetooth Integration

- **Pairing and RFCOMM**: The Pi discovers and pairs with the Arduino's Bluetooth module, creating a virtual serial port (`/dev/rfcomm0`).
- **Node-RED Serial Node**: Listens on `/dev/rfcomm0` to read sensor data, logs it, and/or sends it to the cloud.

C.1.1 Local Decision-Making

- **Alarms/Actuators**: If temperature/humidity are out of range, the Pi can drive a buzzer or LED without waiting for a cloud response.
- **Filtering or Aggregation**: The Pi might average multiple readings before sending them upstream, reducing bandwidth usage.

D. Cloud Layer (ThingsBoard or Similar)

ThingsBoard or another IoT platform provides:

- **Device Registry**: Each sensor node or gateway has an ID/token to authenticate.
- **Telemetry Storage**: Stores time-series data (temperature, humidity, motion events).
- **Dashboards & Alerts**: Visual widgets (charts, gauges) and rules (email notifications, SMS) for triggered conditions.

D.1 MQTT or HTTP Integration

- **MQTT**: Often the default for real-time, lightweight messaging (topics like `v1/devices/me/telemetry`).
- **REST APIs**: Some IoT clouds allow data to be pushed via HTTP `POST` requests, though MQTT is generally more efficient for streaming sensor data.

D.1.1 Remote Procedure Calls (RPC)

- **Command & Control**: The cloud can send a message back to the Pi (or directly to the Arduino) to toggle an alarm or LED if, for instance, temperature is critically high.
- **Dashboard Switches**: Users can remotely flip a virtual switch on the dashboard that triggers a local action on the gateway node.

E. Integrating the Layers: End-to-End Flow

- **1. Sensor Node (Arduino)**:
  - Reads temperature, humidity, or detects motion.
  - Sends data via Bluetooth serial (`Serial.println({temp, hum})`).
- **2. Gateway (Raspberry Pi + Node-RED)**:
  - Receives data on `/dev/rfcomm0`.
  - Processes data (threshold checks, logging).
  - Publishes data to IoT Cloud (e.g., MQTT to ThingsBoard).
  - Optionally triggers local buzzer or LCD updates.
- **3. Cloud (ThingsBoard)**:
  - Stores time-series data.
  - Displays real-time charts and can trigger user-defined alarms.
  - Sends RPC commands back to the gateway if needed.

E.1 Offline Considerations

- **Caching at the Gateway**: If the Pi loses Internet connectivity, Node-RED can queue or buffer sensor data until the connection is restored.

- **Local Fallback**: The Pi can continue critical operations (like sounding an alarm) even without cloud access.

F. Example Use Cases and Future Extensions

- **Greenhouse Monitoring**: Combine temperature/humidity thresholds, fans, or irrigation triggers, plus data logging to the cloud for trend analysis.
- **Multi-Sensor Security**: PIR + door sensors + noise level detection, all aggregated by the Pi, with cloud-based event logging and phone notifications.
- **Machine Learning at the Edge**: Use the Pi's higher processing power to run simple ML models (e.g., anomaly detection) before sending summarized data to the cloud.
- **Scalability**: Expand from one sensor node to many, each using Bluetooth or other protocols (Zigbee, Wi-Fi) to communicate with a single gateway.

G. Best Practices and Pitfalls

- **Network Reliability**: Ensure your Pi's Wi-Fi or Ethernet is stable; unexpected drops cause data loss if not handled.
- **Security**: Use strong credentials on your Pi and consider enabling encryption (SSL/TLS) if sending sensitive data.
- **System Logging**: Keep logs for each layer (Arduino serial prints, Node-RED debug, cloud logs) to aid troubleshooting.
- **Power Management**: If the sensor node or Pi is battery-powered, plan for sleep modes or scheduled transmissions to extend battery life.

Final Note.

Mastering an **end-to-end IoT stack** entails blending electronics know-how (sensor wiring, microcontroller coding), embedded software (Arduino sketches, Node-RED flows), and cloud configuration (MQTT, dashboard widgets, RPC). By uniting these layers, you gain the flexibility to create responsive, data-driven systems—whether for home automation, environmental monitoring, or industrial IoT applications. Continual experimentation with new sensors, advanced analytics, and different communication methods will deepen your skillset in building robust, real-world IoT solutions.

7. Introduction to Wireshark on Raspberry Pi

Objectives

- Install and configure Wireshark on a Raspberry Pi (Debian-based OS).
- Capture and analyze network packets in real time.
- Filter, search, and store captured data.
- Learn basic Wireshark features such as colorization and packet statistics.

A. What is Wireshark?

Wireshark is an industry-standard, open-source network protocol analyzer. It captures and displays real-time or recorded network traffic at a highly granular level, making it invaluable for troubleshooting, security analysis, and learning about networking protocols. Because it's cross-platform (available on Linux, Windows, and macOS) and widely adopted, Wireshark sees use everywhere from classroom labs to large enterprises like Verizon or Boeing.

- **Live Capture**:
  - Intercept packet data in real time from a chosen network interface (Ethernet, Wi-Fi, etc.).
  - Observe ongoing connections and data flows as they happen.
- **Packet Analysis**:
  - Open and inspect saved `.pcap` or `.pcapng` files, generated by tools such as `tcpdump`, `WinDump`, or other instances of Wireshark.
  - Drill down into protocol details (Ethernet, IP, TCP, HTTP, DNS, etc.) to study headers and payload.
- **Filtering and Reporting**:
  - Apply powerful filters (e.g., `ip.addr == 192.168.1.100`) to isolate relevant packets.
  - Export or save captured data in various formats and generate statistics or graphical representations of the traffic.

— **Why use Wireshark?.** Wireshark allows you to see exactly what's happening on your network at a packet level. This level of visibility helps you:

- **Diagnose** connectivity issues by analyzing traffic flows and error responses.
- **Secure** your environment by spotting suspicious packets or anomaly traffic.
- **Learn** about networking protocols through a hands-on, visual exploration of how packets traverse networks.

Whether you're investigating a performance bottleneck, debugging an application, or learning network fundamentals, Wireshark is one of the most comprehensive tools you can use.

For more info, see the official docs: `https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html#ChIntroWhatIs`

B. Installing Wireshark

There are two common ways to install Wireshark on a Raspberry Pi (Debian-based OS):

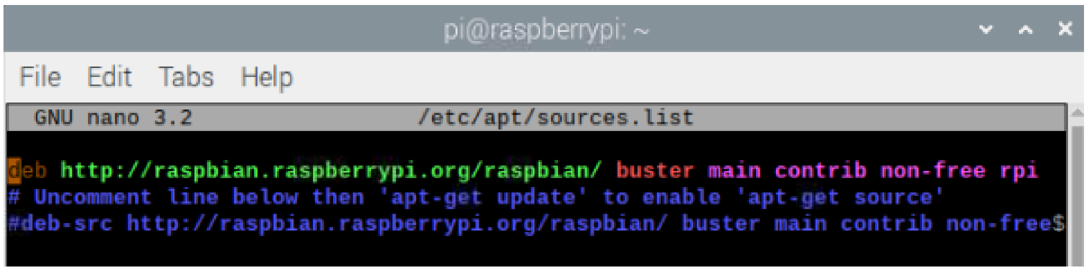
1. **Installing from the repository (preferred)**
2. **Installing from source** (if you need a newer version than the repository offers)

Installing from the Repository

1. **Identify your repository name.** Open a terminal (`Ctrl + Alt + T`) and type:

```
sudo nano /etc/apt/sources.list
```

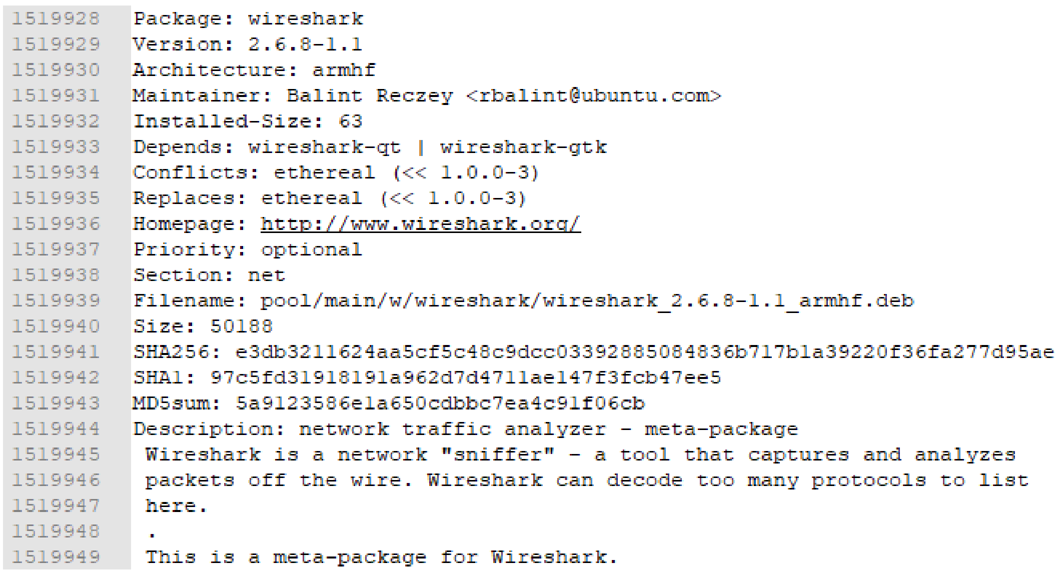
Look for a line indicating your repository name, e.g., **buster** (Figure 7.1).



```
  GNU nano 3.2                 /etc/apt/sources.list

deb http://raspbian.raspberrypi.org/raspbian/ buster main contrib non-free rpi
# Uncomment line below then 'apt-get update' to enable 'apt-get source'
#deb-src http://raspbian.raspberrypi.org/raspbian/ buster main contrib non-free$
```

Figure 7.1: Repository Name

2. **Check if Wireshark is available.** Go to `http://archive.raspbian.org/raspbian/dists/buster/main/binary-armhf/` and download the **Packages** file. Open it in any text editor, then search for "`package: wireshark`." If found (Figure 7.2), it's in the repository.



```
Package: wireshark
Version: 2.6.8-1.1
Architecture: armhf
Maintainer: Balint Reczey <rbalint@ubuntu.com>
Installed-Size: 63
Depends: wireshark-qt | wireshark-gtk
Conflicts: ethereal (<< 1.0.0-3)
Replaces: ethereal (<< 1.0.0-3)
Homepage: http://www.wireshark.org/
Priority: optional
Section: net
Filename: pool/main/w/wireshark/wireshark_2.6.8-1.1_armhf.deb
Size: 50188
SHA256: e3db3211624aa5cf5c48c9dcc03392885084836b717b1a39220f36fa277d95ae
SHA1: 97c5fd31918191a962d7d4711ae147f3fcb47ee5
MD5sum: 5a9123586e1a650cdbbc7ea4c91f06cb
Description: network traffic analyzer - meta-package
 Wireshark is a network "sniffer" - a tool that captures and analyzes
 packets off the wire. Wireshark can decode too many protocols to list
 here.
 .
 This is a meta-package for Wireshark.
```

Figure 7.2: Wireshark Package

3. **Install Wireshark via apt.**

```
sudo apt install wireshark
```

   Approve any dependencies by typing "y." It may ask if non-superusers can run Wireshark. You can usually select **YES** here, allowing normal users to capture packets.

4. **Run Wireshark.** After installation:
   - From the desktop menu: **Internet** → **Wireshark** (Figure 7.3).
   - Or in a terminal:

```
sudo wireshark
```

     (Using sudo avoids permission warnings.)

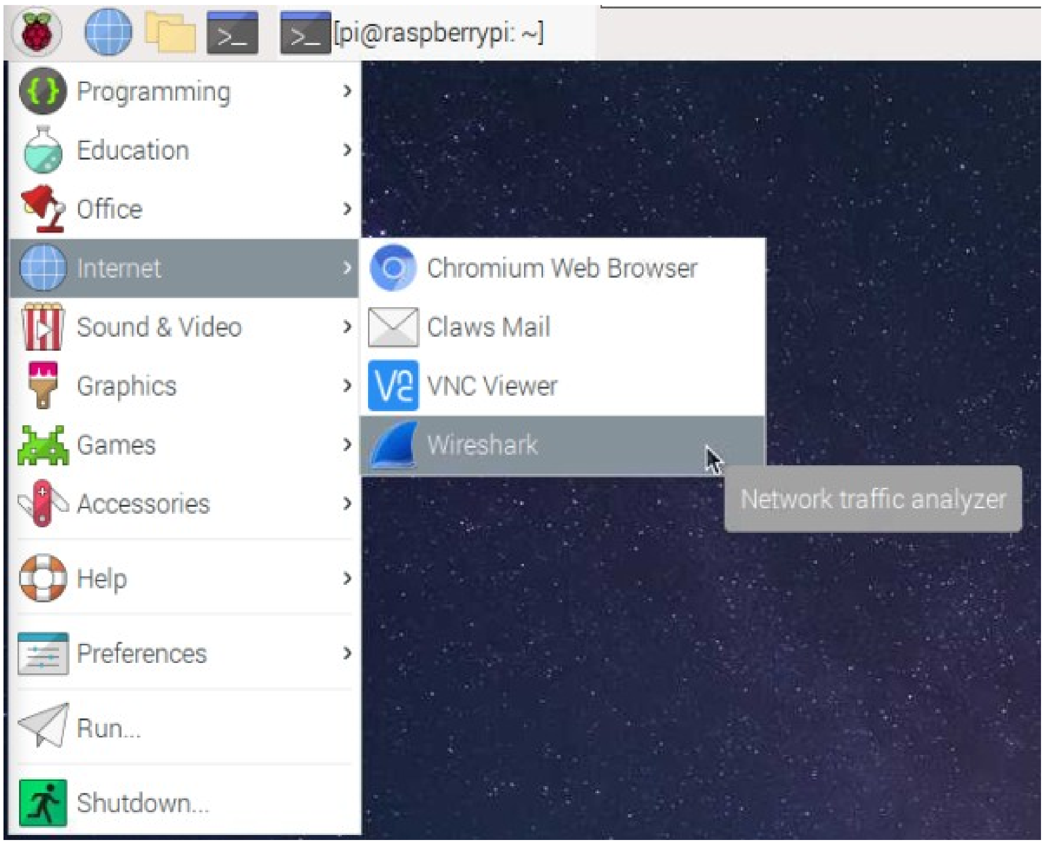


Figure 7.3: Running Wireshark

5. **Check Command Line Options (Optional).** Type:

```
wireshark -h
```

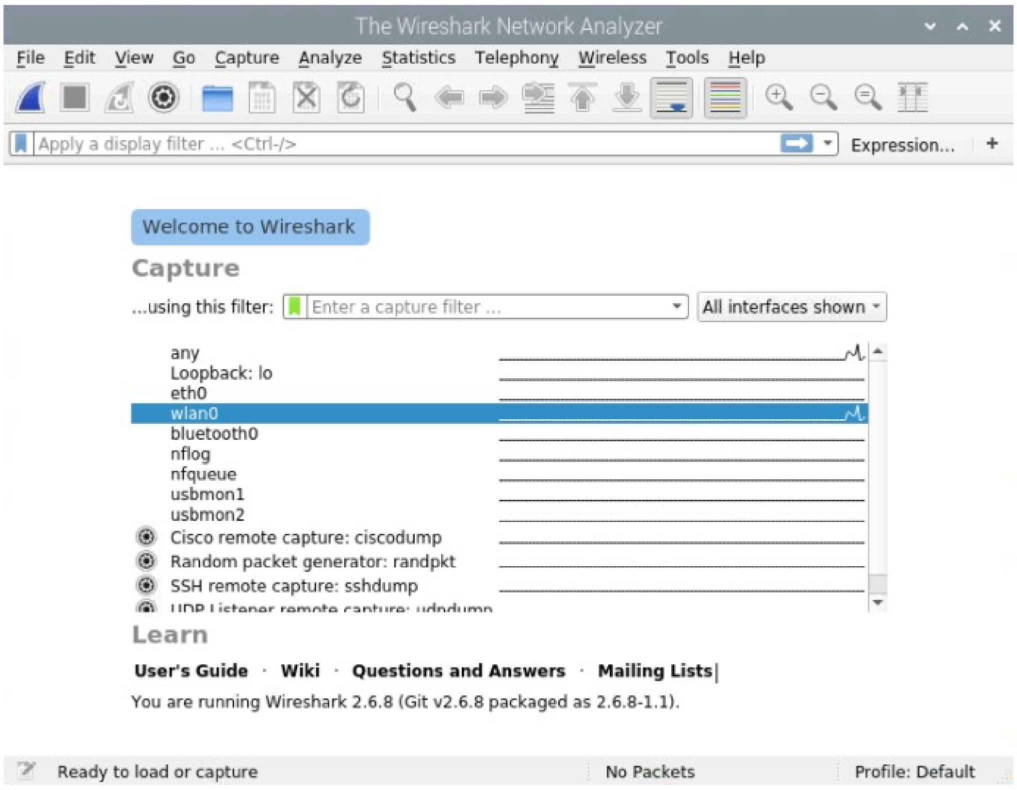
   for a list of available flags and usage tips.

C. Capturing and Analyzing Packets

1. **Launch Wireshark.**
   Once installed, open Wireshark from the **Internet** menu or by running

```
wireshark

```

   in a terminal. You should see a main window listing available network interfaces such as `eth0` (Ethernet), `wlan0` (Wi-Fi), etc., as shown in Figure 7.4.


The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
Welcome to Wireshark
Capture
...using this filter:
Enter a capture filter ...
All interfaces shown
any
Loopback: lo
eth0
wlan0
bluetooth0
nflog
nfqueue
usbmon1
usbmon2
Cisco remote capture: ciscodump
Random packet generator: randpkt
SSH remote capture: sshdump
UDP Listener remote capture: udpdump
Learn
User's Guide · Wiki · Questions and Answers · Mailing Lists
Ready to load or capture
No Packets
Profile: Default

You are running Wireshark 2.6.8 (Git v2.6.8 packaged as 2.6.8-1.1).

Figure 7.4: Wireshark Main Window

2. **Select an Interface & Start Capture.**

   - If you want to monitor Wi-Fi traffic, select `wlan0`.
   - Click the *blue shark fin* icon (Figure 7.5) to begin capturing packets in real time.
   - To capture wired traffic, pick `eth0` (assuming you have an Ethernet cable connected).

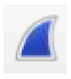


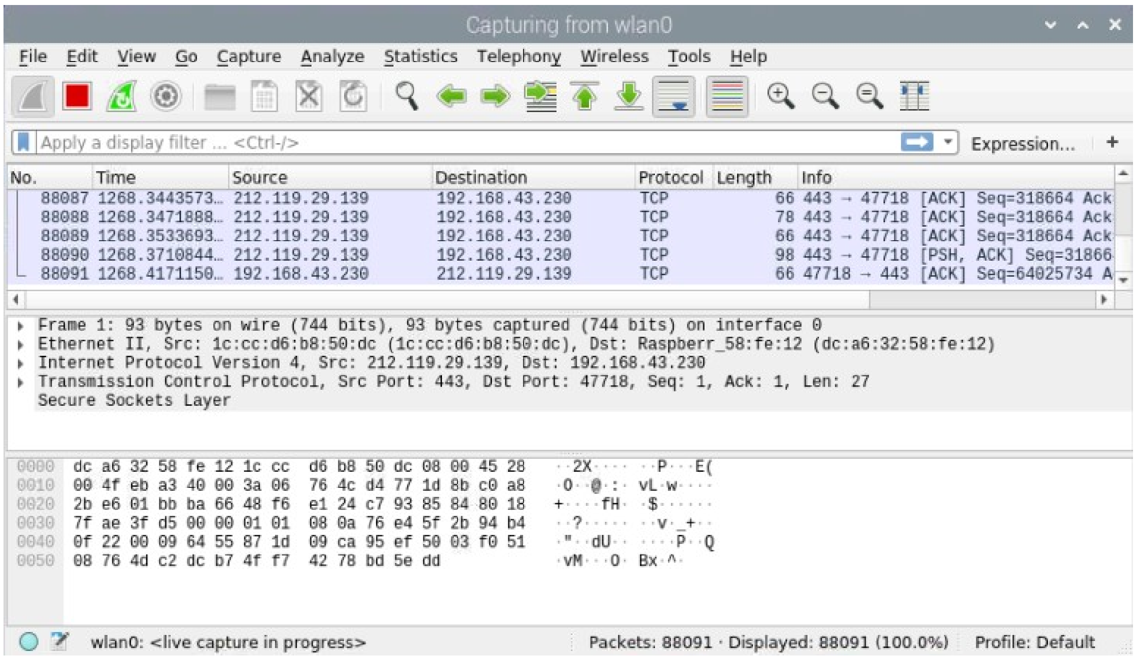
Figure 7.5: Capture Icon (Blue Shark Fin)

— Why do this?. Choosing the correct interface is crucial for seeing the traffic you're interested in. For instance:
- **eth0** for wired/Ethernet captures.
- **wlan0** for wireless captures (although some Wi-Fi drivers limit packet monitoring).
- **any** (on some systems) to capture from all interfaces simultaneously.

Clicking the *blue fin* icon starts the capture process, letting you view data in real time.

3. **Examine Captured Data.**
   As soon as you start capturing, Wireshark displays a live feed of packets (Figure 7.6). You can stop capturing at any time by clicking the **red stop icon**.
   - **Packet List Pane**: Displays each captured packet with summary info (No, Time, Source, Destination, Protocol, Length, Info).
   - **Packet Bytes Pane**: Shows the raw `hex` dump of the selected packet.
   - **Filter Toolbar**: Enter expressions (e.g., `ip.addr == 192.168.0.10`) to narrow down

visible packets.


Capturing from wlan0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
+

```
No.    Time            Source          Destination     Protocol Length Info
88087 1268.3443573… 212.119.29.139  192.168.43.230  TCP        66 443 → 47718 [ACK] Seq=318664 Ack
88088 1268.3471888… 212.119.29.139  192.168.43.230  TCP        78 443 → 47718 [ACK] Seq=318664 Ack
88089 1268.3533693… 212.119.29.139  192.168.43.230  TCP        66 443 → 47718 [ACK] Seq=318664 Ack
88090 1268.3710844… 212.119.29.139  192.168.43.230  TCP        98 443 → 47718 [PSH, ACK] Seq=31866
88091 1268.4171150… 192.168.43.230  212.119.29.139  TCP        66 47718 → 443 [ACK] Seq=64025734 A
```

```
Frame 1: 93 bytes on wire (744 bits), 93 bytes captured (744 bits) on interface 0
Ethernet II, Src: 1c:cc:d6:b8:50:dc (1c:cc:d6:b8:50:dc), Dst: Raspberr_58:fe:12 (dc:a6:32:58:fe:12)
Internet Protocol Version 4, Src: 212.119.29.139, Dst: 192.168.43.230
Transmission Control Protocol, Src Port: 443, Dst Port: 47718, Seq: 1, Ack: 1, Len: 27
Secure Sockets Layer
```

```
0000  dc a6 32 58 fe 12 1c cc  d6 b8 50 dc 08 00 45 28   ··2X····  ··P···E(
0010  00 4f eb a3 40 00 3a 06  76 4c d4 77 1d 8b c0 a8   ·O··@·:·  vL·w····
0020  2b e6 01 bb ba 66 48 f6  e1 24 c7 93 85 84 80 18   +····fH·  ·$······
0030  7f ae 3f d5 00 00 01 01  08 0a 76 e4 5f 2b 94 b4   ··?·····  ··v·_+··
0040  0f 22 00 09 64 55 87 1d  09 ca 95 ef 50 03 f0 51   ·"··dU··  ····P··Q
0050  08 76 4d c2 dc b7 4f f7  42 78 bd 5e dd            ·vM···O·  Bx·^·
```

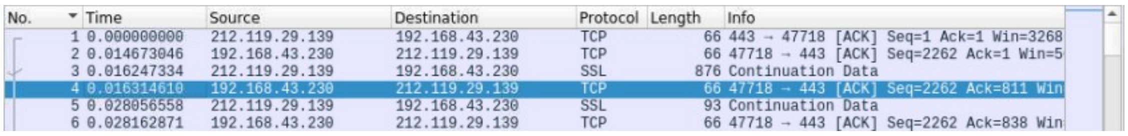
```
wlan0: <live capture in progress>    Packets: 88091 · Displayed: 88091 (100.0%)    Profile: Default
```

Figure 7.6: The Wireshark GUI

Tips:

- **Filtering**: The green filter bar indicates a valid syntax (red means invalid). See all filter syntax at `https://www.wireshark.org/docs/dfref/`.
- **Stop/Restart**: Use the red stop icon to halt live capture and the blue fin icon to start a new one.

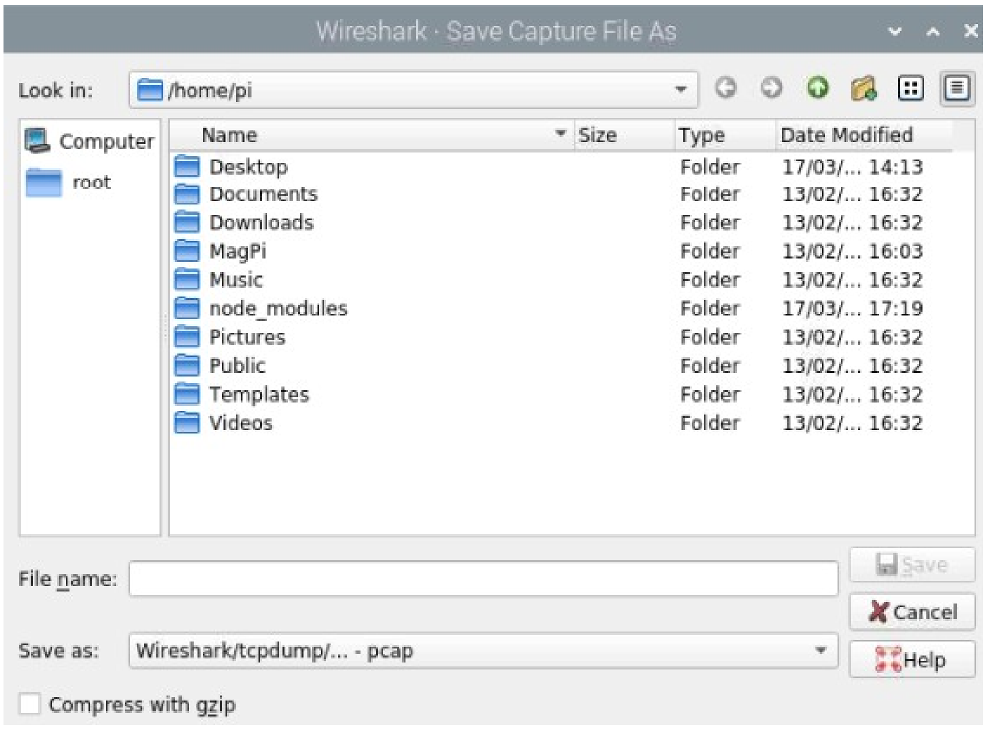
4. **Symbols and Packet Details.**
   Some packets have special symbols in the **No.** column, such as a check/tick (Figure 7.7) to denote an acknowledgment. Visit `https://www.wireshark.org/docs/wsug_html_chunked/ChUsePacketListPaneSection.html` for a complete legend.



```
No.   Time          Source          Destination     Protocol Length Info
  1 0.000000000  212.119.29.139  192.168.43.230  TCP        66 443 → 47718 [ACK] Seq=1 Ack=1 Win=3268
  2 0.014673046  192.168.43.230  212.119.29.139  TCP        66 47718 → 443 [ACK] Seq=2262 Ack=1 Win=5
  3 0.016247334  212.119.29.139  192.168.43.230  SSL       876 Continuation Data
  4 0.016314610  192.168.43.230  212.119.29.139  TCP        66 47718 → 443 [ACK] Seq=2262 Ack=811 Win
  5 0.028056558  212.119.29.139  192.168.43.230  SSL        93 Continuation Data
  6 0.028162871  192.168.43.230  212.119.29.139  TCP        66 47718 → 443 [ACK] Seq=2262 Ack=838 Win
```

Figure 7.7: Acknowledgment Symbol

5. **Saving or Opening** `.pcap` **Files.**
   If you want to preserve a capture session for later analysis or share it with colleagues:
   - Go to **File** → **Save As...** (Figure 7.8) to store the capture in **.pcap** or **.pcapng** format.
   - Open previously saved `.pcap` files anytime in Wireshark using **File** → **Open**.
   - These files are widely used in cybersecurity to dissect potential attacks, debug network issues, or study IoT traffic patterns.



Name	Size	Type	Date Modified
Desktop		Folder	17/03/... 14:13
Documents		Folder	13/02/... 16:32
Downloads		Folder	13/02/... 16:32
MagPi		Folder	13/02/... 16:03
Music		Folder	13/02/... 16:32
node_modules		Folder	17/03/... 17:19
Pictures		Folder	13/02/... 16:32
Public		Folder	13/02/... 16:32
Templates		Folder	13/02/... 16:32
Videos		Folder	13/02/... 16:32

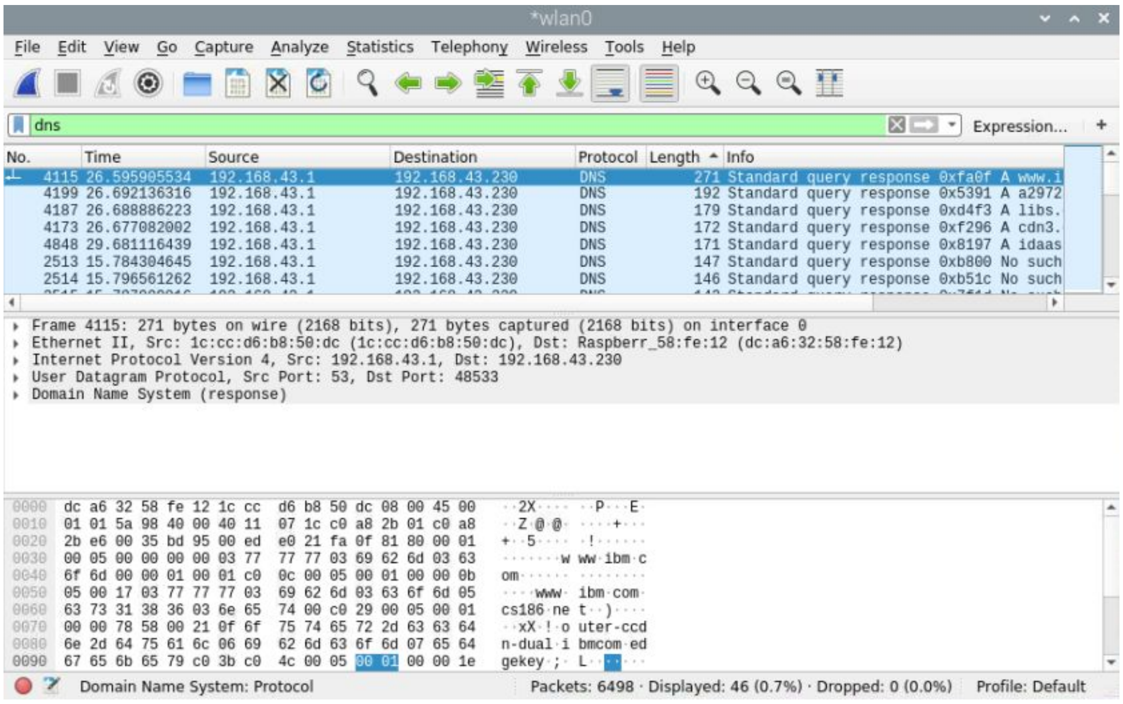
Figure 7.8: Saving a PCAP File

— **Why do this?.** Saving captures lets you:
- **Revisit an event**: Inspect suspicious traffic or debug an issue at your own pace.
- **Share with others**: Collaborate with teammates or support engineers by sending them the same capture file.
- **Build a library**: Over time, you can gather typical traffic patterns or interesting attack scenarios for training and educational purposes.

D. Filtering, Coloring, and Statistics

7. **Applying Filters.** Use the filter bar to display only certain packets. For example:
   - **dns**: Shows only DNS traffic (Figure 7.9).
   - **ip.addr == 194.13.12.12**: Shows packets where `194.13.12.12` is source or destination.

   If the filter is valid, the bar is **green**; if invalid, it's **red**.


*wlan0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
dns
Expression...
+

```
No.   Time          Source         Destination     Protocol Length Info
 4115 26.595905534  192.168.43.1   192.168.43.230  DNS      271 Standard query response 0xfa0f A www.i
 4199 26.692136316  192.168.43.1   192.168.43.230  DNS      192 Standard query response 0x5391 A a2972
 4187 26.688886223  192.168.43.1   192.168.43.230  DNS      179 Standard query response 0xd4f3 A libs.
 4173 26.677082002  192.168.43.1   192.168.43.230  DNS      172 Standard query response 0xf296 A cdn3.
 4848 29.681116439  192.168.43.1   192.168.43.230  DNS      171 Standard query response 0x8197 A idaas
 2513 15.784304645  192.168.43.1   192.168.43.230  DNS      147 Standard query response 0xb800 No such
 2514 15.796561262  192.168.43.1   192.168.43.230  DNS      146 Standard query response 0xb51c No such
```

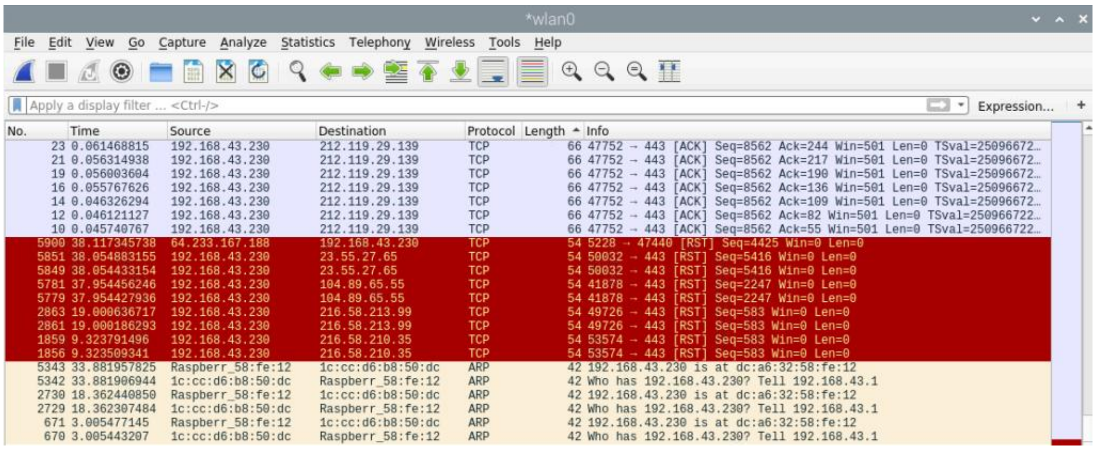
```
Frame 4115: 271 bytes on wire (2168 bits), 271 bytes captured (2168 bits) on interface 0
Ethernet II, Src: 1c:cc:d6:b8:50:dc (1c:cc:d6:b8:50:dc), Dst: Raspberr_58:fe:12 (dc:a6:32:58:fe:12)
Internet Protocol Version 4, Src: 192.168.43.1, Dst: 192.168.43.230
User Datagram Protocol, Src Port: 53, Dst Port: 48533
Domain Name System (response)
```

```
0000  dc a6 32 58 fe 12 1c cc  d6 b8 50 dc 08 00 45 00   ··2X···· ··P···E·
0010  01 01 5a 98 40 00 40 11  07 1c c0 a8 2b 01 c0 a8   ··Z·@·@· ····+···
0020  2b e6 00 35 bd 95 00 ed  e0 21 fa 0f 81 80 00 01   +··5···· ·!······
0030  00 05 00 00 00 00 03 77  77 77 03 69 62 6d 03 63   ·······w ww·ibm·c
0040  6f 6d 00 00 01 00 01 c0  0c 00 05 00 01 00 00 0b   om······ ········
0050  05 00 17 03 77 77 77 03  69 62 6d 03 63 6f 6d 05   ····www· ibm·com·
0060  63 73 31 38 36 03 6e 65  74 00 c0 29 00 05 00 01   cs186·ne t··)····
0070  00 00 78 58 00 21 0f 6f  75 74 65 72 2d 63 63 64   ··xX·!·o uter-ccd
0080  6e 2d 64 75 61 6c 06 69  62 6d 63 6f 6d 07 65 64   n-dual·i bmcom·ed
0090  67 65 6b 65 79 c0 3b c0  4c 00 05 00 01 00 00 1e   gekey·;· L·······
```

Domain Name System: Protocol   Packets: 6498 · Displayed: 46 (0.7%) · Dropped: 0 (0.0%)   Profile: Default

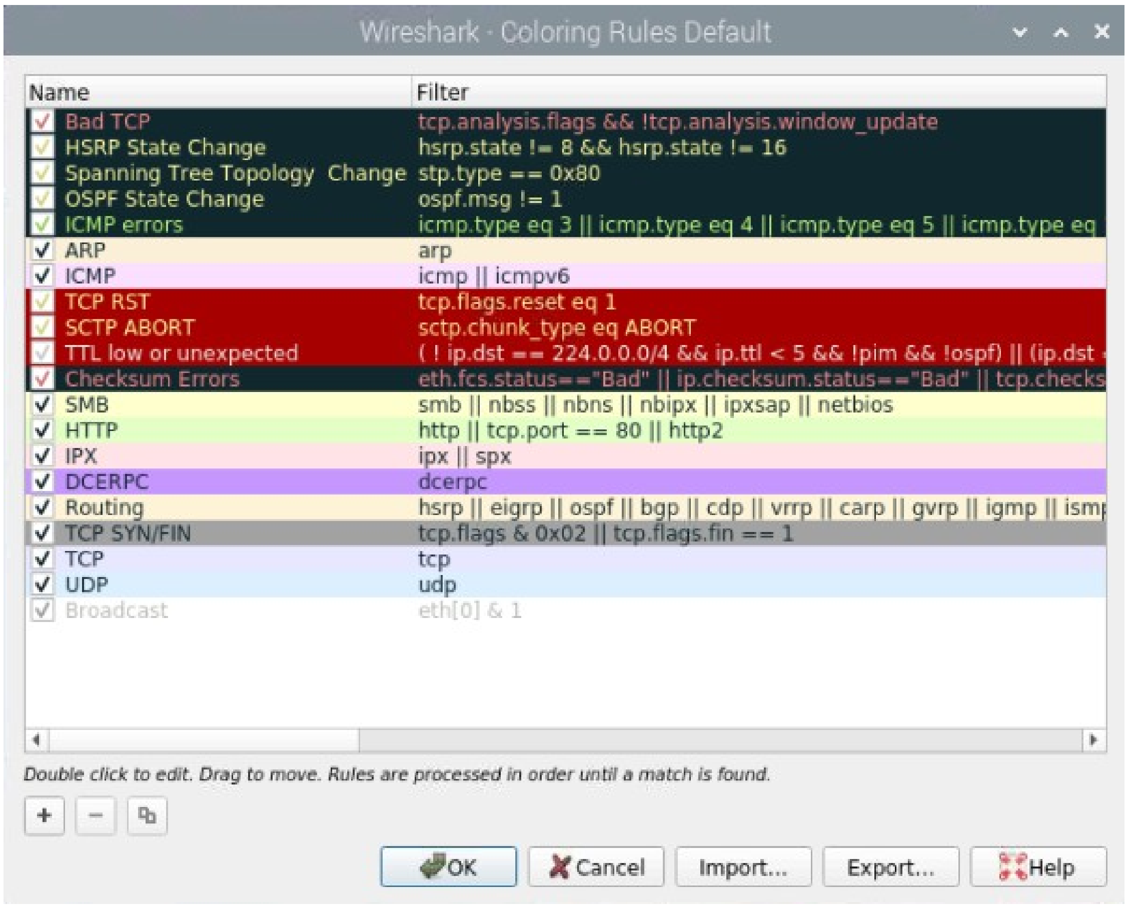
Figure 7.9: Applying a DNS Filter

8. **Colorization Rules.** Wireshark color-codes traffic for quick identification (e.g., HTTP in green, DNS in blue).
   In **View** → **Coloring Rules**, you can customize or add new rules (Figures 7.10 & 7.11).


*wlan0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
+

```
No.   Time          Source             Destination        Protocol Length Info
   23 0.061468815   192.168.43.230     212.119.29.139     TCP      66 47752 → 443 [ACK] Seq=8562 Ack=244 Win=501 Len=0 TSval=25096672…
   21 0.056314938   192.168.43.230     212.119.29.139     TCP      66 47752 → 443 [ACK] Seq=8562 Ack=217 Win=501 Len=0 TSval=25096672…
   19 0.056003604   192.168.43.230     212.119.29.139     TCP      66 47752 → 443 [ACK] Seq=8562 Ack=190 Win=501 Len=0 TSval=25096672…
   16 0.055767626   192.168.43.230     212.119.29.139     TCP      66 47752 → 443 [ACK] Seq=8562 Ack=136 Win=501 Len=0 TSval=25096672…
   14 0.046326294   192.168.43.230     212.119.29.139     TCP      66 47752 → 443 [ACK] Seq=8562 Ack=109 Win=501 Len=0 TSval=25096672…
   12 0.046121127   192.168.43.230     212.119.29.139     TCP      66 47752 → 443 [ACK] Seq=8562 Ack=82 Win=501 Len=0 TSval=250966722…
   10 0.045740767   192.168.43.230     212.119.29.139     TCP      66 47752 → 443 [ACK] Seq=8562 Ack=55 Win=501 Len=0 TSval=250966722…
 5900 38.117345738  64.233.167.188     192.168.43.230     TCP      54 5228 → 47440 [RST] Seq=4425 Win=0 Len=0
 5851 38.054883155  192.168.43.230     23.55.27.65        TCP      54 50032 → 443 [RST] Seq=5416 Win=0 Len=0
 5849 38.054433154  192.168.43.230     23.55.27.65        TCP      54 50032 → 443 [RST] Seq=5416 Win=0 Len=0
 5781 37.954456246  192.168.43.230     104.89.65.55       TCP      54 41878 → 443 [RST] Seq=2247 Win=0 Len=0
 5779 37.954427936  192.168.43.230     104.89.65.55       TCP      54 41878 → 443 [RST] Seq=2247 Win=0 Len=0
 2863 19.000636717  192.168.43.230     216.58.213.99      TCP      54 49726 → 443 [RST] Seq=583 Win=0 Len=0
 2861 19.000186293  192.168.43.230     216.58.213.99      TCP      54 49726 → 443 [RST] Seq=583 Win=0 Len=0
 1859 9.323791496   192.168.43.230     216.58.210.35      TCP      54 53574 → 443 [RST] Seq=583 Win=0 Len=0
 1856 9.323509341   192.168.43.230     216.58.210.35      TCP      54 53574 → 443 [RST] Seq=583 Win=0 Len=0
 5343 33.881957825  Raspberr_58:fe:12  1c:cc:d6:b8:50:dc  ARP      42 192.168.43.230 is at dc:a6:32:58:fe:12
 5342 33.881906944  1c:cc:d6:b8:50:dc  Raspberr_58:fe:12  ARP      42 Who has 192.168.43.230? Tell 192.168.43.1
 2730 18.362440850  Raspberr_58:fe:12  1c:cc:d6:b8:50:dc  ARP      42 192.168.43.230 is at dc:a6:32:58:fe:12
 2729 18.362307484  1c:cc:d6:b8:50:dc  Raspberr_58:fe:12  ARP      42 Who has 192.168.43.230? Tell 192.168.43.1
  671 3.005477145   Raspberr_58:fe:12  1c:cc:d6:b8:50:dc  ARP      42 192.168.43.230 is at dc:a6:32:58:fe:12
  670 3.005443207   1c:cc:d6:b8:50:dc  Raspberr_58:fe:12  ARP      42 Who has 192.168.43.230? Tell 192.168.43.1
```

Figure 7.10: Packet Colorization

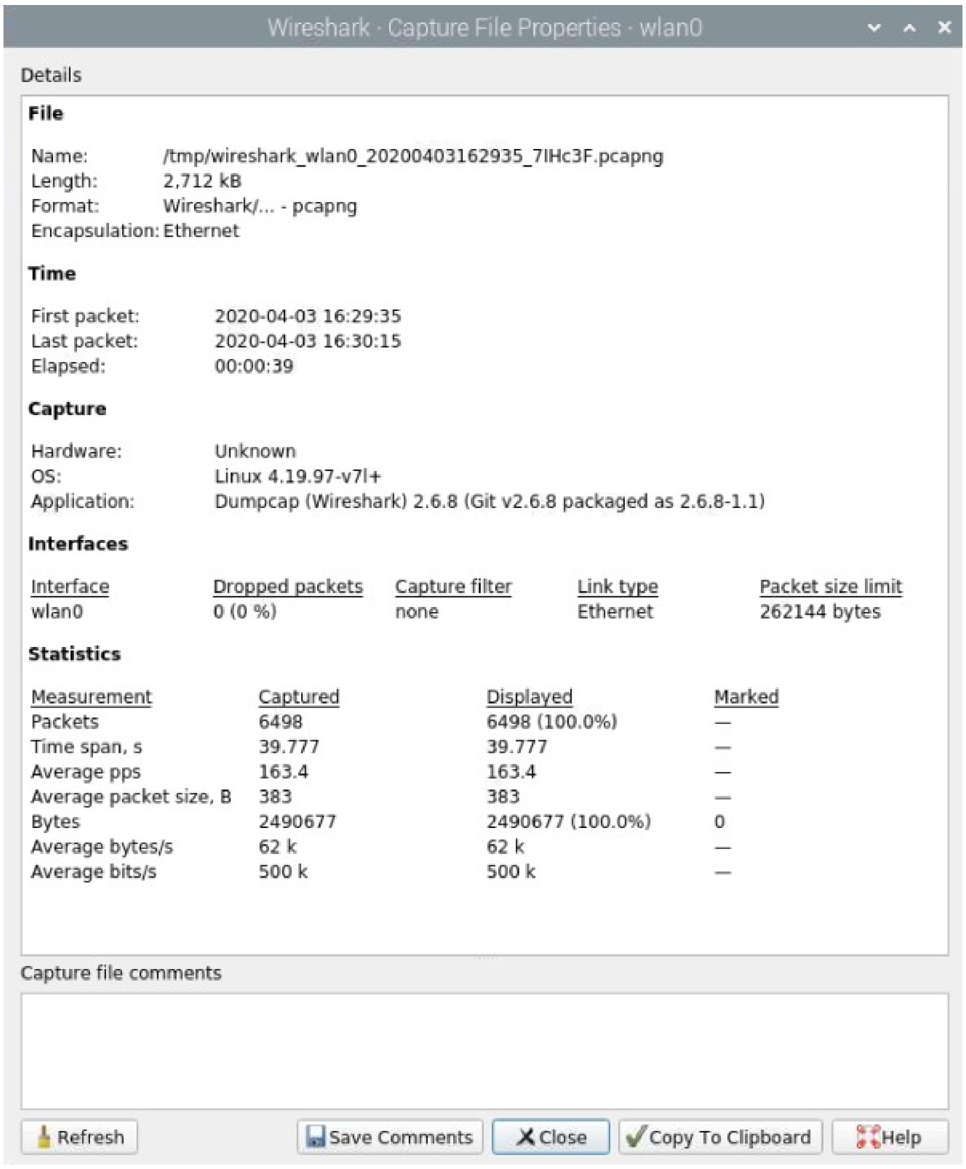


Name	Filter
✓ Bad TCP	tcp.analysis.flags && !tcp.analysis.window_update
✓ HSRP State Change	hsrp.state != 8 && hsrp.state != 16
✓ Spanning Tree Topology Change	stp.type == 0x80
✓ OSPF State Change	ospf.msg != 1
✓ ICMP errors	icmp.type eq 3 || icmp.type eq 4 || icmp.type eq 5 || icmp.type eq
✓ ARP	arp
✓ ICMP	icmp || icmpv6
✓ TCP RST	tcp.flags.reset eq 1
✓ SCTP ABORT	sctp.chunk_type eq ABORT
✓ TTL low or unexpected	(! ip.dst == 224.0.0.0/4 && ip.ttl < 5 && !pim && !ospf) || (ip.dst
✓ Checksum Errors	eth.fcs.status=="Bad" || ip.checksum.status=="Bad" || tcp.checks
✓ SMB	smb || nbss || nbns || nbipx || ipxsap || netbios
✓ HTTP	http || tcp.port == 80 || http2
✓ IPX	ipx || spx
✓ DCERPC	dcerpc
✓ Routing	hsrp || eigrp || ospf || bgp || cdp || vrrp || carp || gvrp || igmp || ism
✓ TCP SYN/FIN	tcp.flags & 0x02 || tcp.flags.fin == 1
✓ TCP	tcp
✓ UDP	udp
✓ Broadcast	eth[0] & 1

Double click to edit. Drag to move. Rules are processed in order until a match is found.

Figure 7.11: Editing Color Rules

9. **Generating Statistics.** Under **Statistics** in the menu, you can view “Summary,” “Conversations,” or other advanced options (Figure 7.12). This helps identify top talkers, protocol breakdown, or create graphs of throughput.



File

```
Name:          /tmp/wireshark_wlan0_20200403162935_7IHc3F.pcapng
Length:        2,712 kB
Format:        Wireshark/... - pcapng
Encapsulation: Ethernet
```

Time

```
First packet:    2020-04-03 16:29:35
Last packet:     2020-04-03 16:30:15
Elapsed:         00:00:39
```

Capture

```
Hardware:        Unknown
OS:              Linux 4.19.97-v7l+
Application:     Dumpcap (Wireshark) 2.6.8 (Git v2.6.8 packaged as 2.6.8-1.1)
```

Interfaces

Interface	Dropped packets	Capture filter	Link type	Packet size limit
wlan0	0 (0 %)	none	Ethernet	262144 bytes

Statistics

Measurement	Captured	Displayed	Marked
Packets	6498	6498 (100.0%)	—
Time span, s	39.777	39.777	—
Average pps	163.4	163.4	—
Average packet size, B	383	383	—
Bytes	2490677	2490677 (100.0%)	0
Average bytes/s	62 k	62 k	—
Average bits/s	500 k	500 k	—

Figure 7.12: Statistics Menu

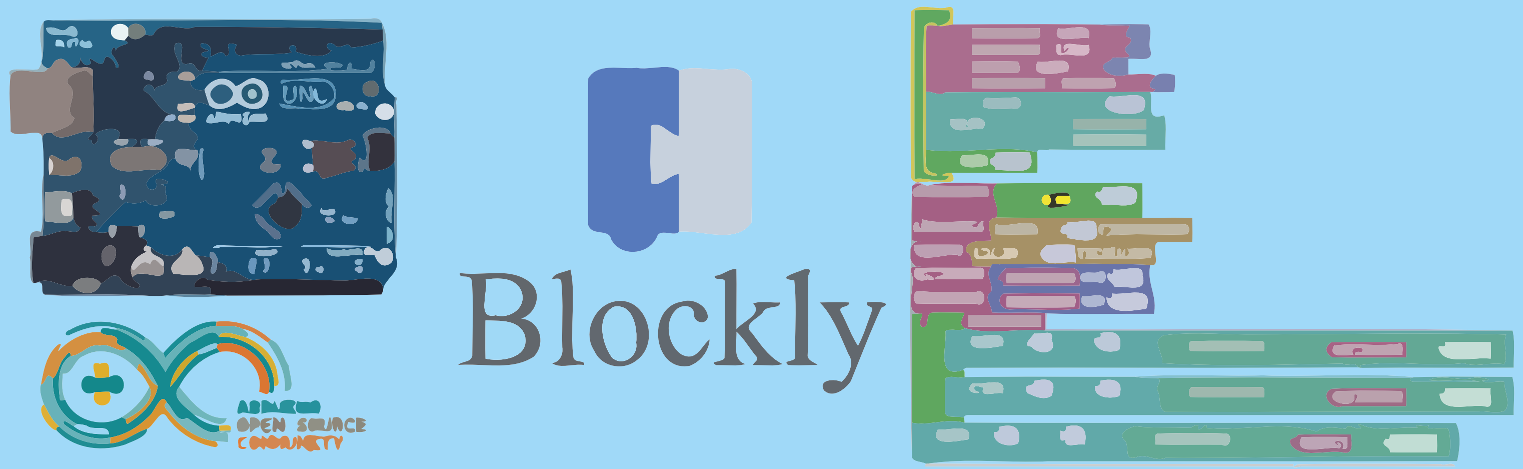
Measuring Success

- **Wireshark Installed & Running**: You can launch Wireshark from the desktop menu or via terminal with `sudo wireshark` without encountering error messages or permission issues.
- **Capturing Live Traffic**: The packet list pane updates continuously for your chosen interface (`wlan0`, `eth0`, etc.), indicating successful real-time capture.
- **Using Filters Successfully**: Entering filters like `dns` or `ip.addr == X` in the green filter bar narrows the displayed packets, demonstrating that you can isolate relevant traffic.
- **Saving/Opening PCAP Files**: You can store your session in a `.pcap` or `.pcapng` file and reopen it later in Wireshark for replay or deeper analysis.

■

— Further Reading

- **Wireshark Docs**: `https://www.wireshark.org/docs/`

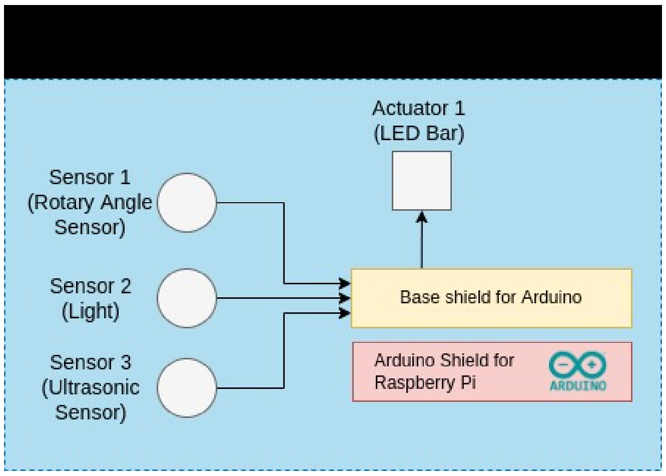


8. Programming Arduino with Blockly

Objectives

- Learn how to program an Arduino via Blockly
- Learn how to read data from various sensors
- Learn how to program a LED Bar

Lab Plan



In this lab, we'll use Codecraft to generate Arduino code by dragging and dropping blocks (Blockly-style). We'll control an LED Bar using input from a rotary angle sensor. We'll also gather data from other sensors (light, ultrasonic) and observe their outputs.

Required Hardware Components

- **Grove - Rotary Angle Sensor v1.2**

- **Grove - LED Bar v2.1**
- **Grove - Light Sensor v1.2**
- **Grove - Ultrasonic Ranger v2.0**
- **Arduino Expansion Shield for Raspberry Pi B+ (V2.0)**
- **Grove - Base Shield**

A. Setting Up Codecraft

1. **Go to** `www.tinkergen.com`.
   In your web browser, navigate to the TinkerGen website. At the top menu, click on **Codecraft** → **Programming Online**. This opens the Codecraft environment—a web-based, block-programming interface for Arduino and other microcontrollers.

   > **— Why Codecraft?.** Codecraft provides a **Blockly**-style, drag-and-drop interface for writing Arduino sketches. It's especially beginner-friendly since you don't need to remember syntax or manage libraries manually.

2. **Choose *Arduino(Uno/Mega/BeginnerKit)*.**
   After the Codecraft environment loads, it will prompt you to select a hardware platform.
   - Click **Arduino(Uno/Mega/BeginnerKit)**: This sets up your workspace to generate code tailored for typical Arduino boards (e.g., Uno, Mega, etc.).
   - Other platforms like **Micro:bit** or **mCookie** differ in available blocks and libraries, so ensure you pick the Arduino option for compatibility.

   > **Tips:**
   > - If you're using a different Arduino-based board (e.g., Leonardo), most blocks will still work, but confirm pin mappings or any advanced features (e.g., multiple serial ports).
   > - Keep your browser up to date to avoid compatibility issues with Codecraft's online editor.

3. **Workspace Overview.**
   You should see an interface similar to Figure 8.1, where you can **drag blocks** from the left sidebar into the main workspace area.
   - **Blocks Palette**: Organized by categories (e.g., *Control*, *Logic*, *Variables*, *Sensors*). Click a category to see the blocks inside.
   - **Main Workspace**: Where you assemble your program by snapping blocks together.
   - **Code Panel Switch**: Usually at the top-right. Allows you to view and copy the generated **C++ code** if you want to paste it into the Arduino IDE directly.

   > **— Why drag-and-drop?.** Visual block coding significantly reduces syntax errors and helps beginners focus on **logic** rather than **syntax**. Once comfortable, you can switch to viewing or editing the underlying C++ code to learn more advanced techniques.

B. Setting Up the Arduino

4. **Attach Grove Base Shield.**
   Carefully line up the male pins on the underside of the Base Shield with the female pin headers on your Arduino (e.g., Arduino Leonardo). Gently press down until it sits flush. Next, plug in your Grove components:

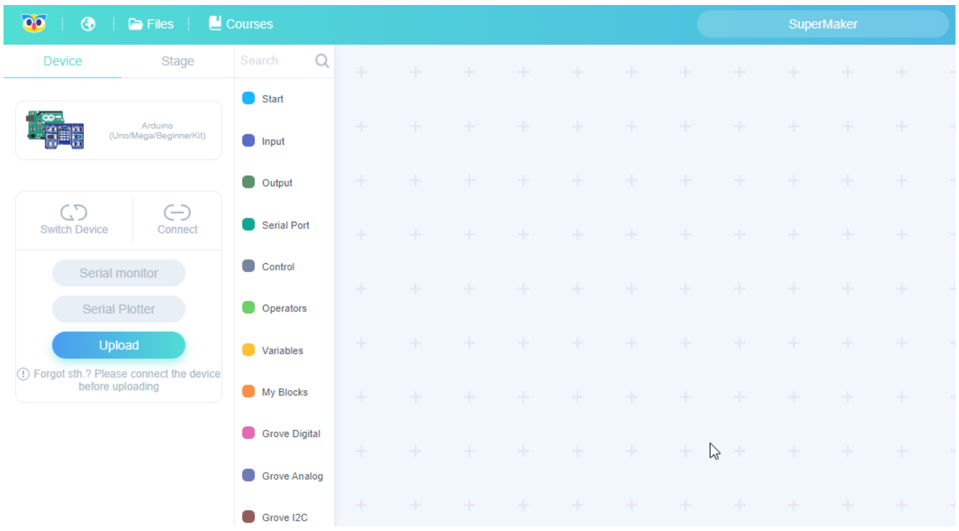


Figure 8.1: Codecraft Workspace

- **LED Bar** → **D4**
- **Rotary Angle Sensor** → **A3**
- **Ultrasonic Ranger** → **D3**
- **Light Sensor** → **A0**

Ensure the Base Shield's voltage switch is set to **5V**, which suits most Grove sensors and actuators.

> **— Why 5V?.** Some Grove sensors (like the LED Bar) function best at 5V, and many Arduino boards naturally operate at 5V logic. Double-check each sensor's specs if you're unsure. Setting the shield to 3.3V for components that need 5V may cause them to behave incorrectly or fail to power on.

5. **Connect Arduino to PC.**
   Use a standard USB cable (type A to micro/mini/USB-C, depending on your board) to connect the Arduino to your computer. Launch the Arduino IDE and configure:
   - **Tools** → **Board** → *Arduino Leonardo* (or your specific board type).
   - **Tools** → **Port** → select the correct COM port (Windows) or /dev/`ttyACM*` (Linux/macOS) associated with your Arduino.

> **Tips:**
> - If you're unsure which port to pick, unplug the Arduino and see which port disappears; then plug it back in to confirm.
> - For some boards (e.g., Arduino Uno, Arduino Mega), the board selection in the IDE should match the actual board you're using.

6. **Verify Basic Setup.**
   It's often wise to upload a blank (empty) sketch or a simple "Blink" example to confirm your Arduino is recognized:
   - Click **File** → **New**, then upload the empty template (just `setup()` and `loop()`).
   - If "Done uploading" appears with no errors, your board, drivers, and USB connection are all functioning.

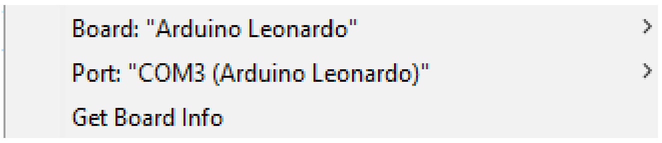

Board: "Arduino Leonardo"
Port: "COM3 (Arduino Leonardo)"
Get Board Info

Figure 8.2: Arduino Leonardo Settings

Figure 8.2 shows an example of Arduino Leonardo settings under the **Tools** menu.

C. Programming Sensors with Codecraft

In this section, you'll learn how to configure and generate Arduino code for three different sensors—Ultrasonic Ranger, Light Sensor, and Rotary Angle Sensor—using the Codecraft (Blockly) environment.

Ultrasonic Ranger

7. **Install the Ultrasonic Ranger Library (If Needed).**
   Some Codecraft blocks rely on libraries that the Arduino IDE might not have by default.
   - Download the file "`Seeed_Arduino_UltrasonicRanger-master.zip`" from GitLab.
   - In the Arduino IDE, go to **Sketch** → **Include Library** → **Add .ZIP Library...** and select the downloaded file.

   This ensures your generated sketch can compile properly when referencing ultrasonic functions.
8. **Create a Codecraft Setup (Figure 8.3).**
   In the Codecraft (Blockly) workspace:
   - Drag blocks to **initialize** and **read distance** from the Ultrasonic Ranger.
   - Make sure the pin selection matches your hardware connection (e.g., `D3`).
   - Set an appropriate measurement interval (e.g., read distance every 500ms).

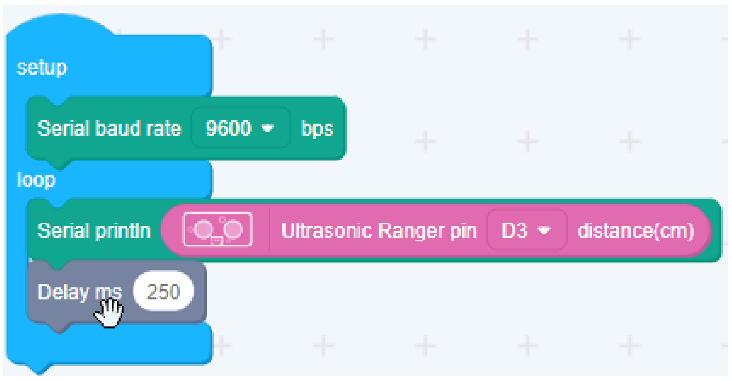

setup
Serial baud rate 9600 bps
loop
Serial println Ultrasonic Ranger pin D3 distance(cm)
Delay ms 250

Figure 8.3: Codecraft Ultrasonic Ranger Setup

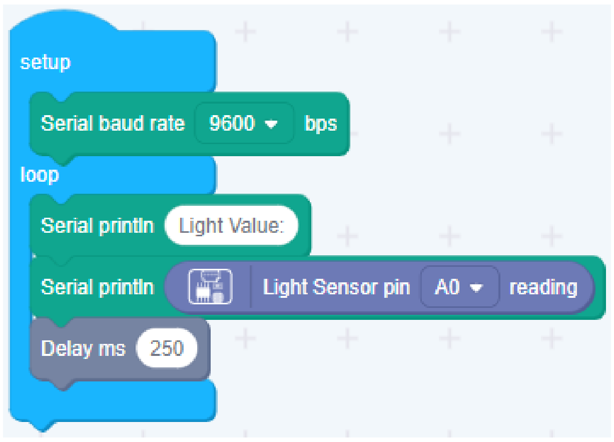
9. **Switch to Text Code.**
   Codecraft automatically translates your blocks into C++ code:
   - Click the "**workspace switch**" icon (Figure 8.4) in the top-right corner.
   - A text window with `.ino` code will appear. Copy the code to your clipboard.
10. **Upload and Observe.**

Figure 8.4: Workspace Switch to View Code

- Open the Arduino IDE, create a new sketch, and paste the copied code.
- Compile and upload. Then, open the **Serial Monitor**:
  - You should see distance readings in centimeters.
  - Move your hand or an object closer/farther to the sensor to see real-time changes.

Light Sensor

11. **Codecraft Setup (Figure 8.5).**
    For the Light Sensor connected to `A0`:
    - Drag the relevant **analog read** blocks in Codecraft.
    - Switch the pin to `A0` in the block's dropdown (or however Codecraft prompts you).
    - Convert to text code just like before.



```
setup
  Serial baud rate 9600 bps
loop
  Serial println Light Value:
  Serial println Light Sensor pin A0 reading
  Delay ms 250
```

Figure 8.5: Codecraft Light Sensor Setup

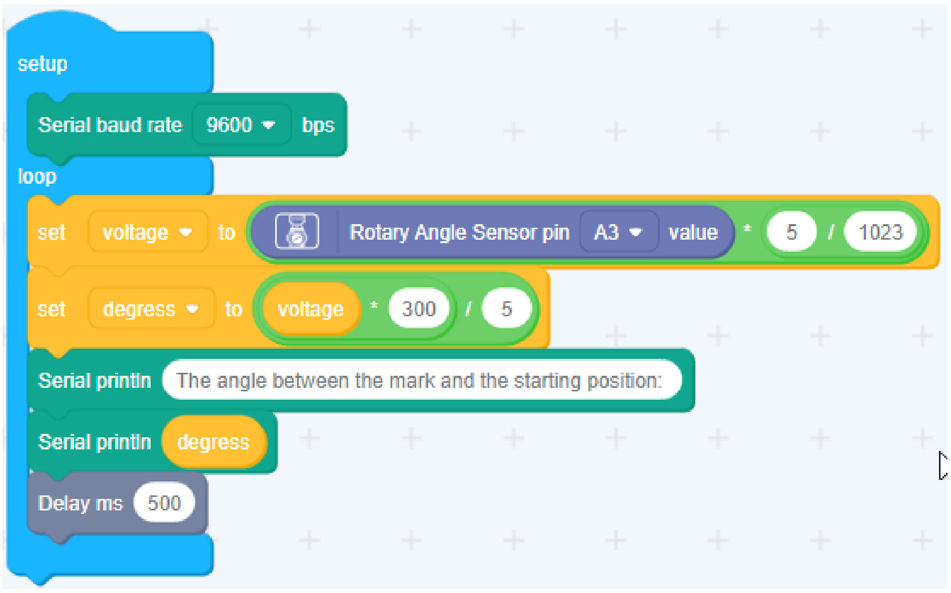
12. **Check Serial Monitor.**
    After pasting and uploading the generated code in the Arduino IDE:
    - Open the **Serial Monitor** to see `lightValue` in real time.
    - Vary the lighting conditions (e.g., shine a flashlight or cover the sensor) to observe how the analog value changes.

Rotary Angle Sensor

13. **Blocks for Reading Voltage/Angle (Figure 8.6).**
    In Codecraft, drag blocks that read the analog pin (`A3`) and convert the returned value into a voltage or angle. Check the code to ensure it calculates:
    - `voltage = (analogValue) * 5 / 1023;`
    - Or it may generate a block-based function referencing a library call.
14. **Minor Code Adjustments.**
    Sometimes Codecraft might add extra parentheses. For instance:

```
voltage = (analogRead(A3)) * 5 / 1023;

```

You can remove the unnecessary parentheses for clarity:



setup
Serial baud rate 9600 bps
loop
set voltage to Rotary Angle Sensor pin A3 value * 5 / 1023
set degress to voltage * 300 / 5
Serial println "The angle between the mark and the starting position:"
Serial println degress
Delay ms 500

Figure 8.6: Codecraft Rotary Angle Sensor Setup

```
voltage = analogRead(A3) * 5 / 1023;

```

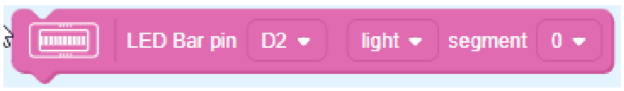
(It functions identically, but it's often cleaner to keep the code simpler.)

15. **Upload and Test.**
    Upload the code to your Arduino, then:
    - Open the **Serial Monitor**.
    - Rotate the sensor shaft and watch how the `angle/voltage` value changes.

St

D. Programming the LED Bar with Codecraft

16. **Install Grove LED Bar Library.**
    Download "Grove_LED_Bar-master.zip" from GitLab. In Arduino IDE, do **Sketch** → **Include Library** → **Add .ZIP Library...**.
17. **Test the Provided Example.**
    In GitLab, there's a file `groveLEDBar.ino`. Copy it to your Arduino IDE and upload. This code cycles the LED bar in different patterns.
18. **Reproduce It in Codecraft.**
    You can find the LED Bar block in **Grove Digital** (Figure 8.7). Build a flow that matches the behavior you just tested.



LED Bar pin D2 light segment 0

Figure 8.7: Codecraft LED Bar Node

19. **Upload and Observe.**
    The LED bar should light up in discrete segments, showing the pattern you chose in your block logic.

Measuring Success

- **Ultrasonic Ranger:**
  - Your Serial Monitor displays distance in centimeters, changing dynamically as you move objects closer or farther from the sensor.
  - The readings are stable and update at a predictable interval (e.g., every 500ms) without significant noise or unexpected zeros.
- **Light Sensor:**
  - The `lightValue` (analog reading) in the Serial Monitor rises under bright light and falls when covered or in darker settings.
  - Minor changes in ambient lighting are reflected promptly in the displayed values.
- **Rotary Angle Sensor:**
  - Rotating the knob yields clear changes in angle/voltage in the Serial Monitor.
  - The value transitions smoothly from low (knob turned fully one way) to high (knob turned fully the other way).
- **LED Bar:**
  - When running your Codecraft-generated or `groveLEDBar.ino` code, the LED bar illuminates discrete segments in the pattern you specified.
  - Animations cycle correctly (e.g., from 1 to 10 segments, back down, or in custom sequences), demonstrating that the library is properly controlling the bar.

■

Use Case Scenario (Optional): Imagine a *smart window* system where multiple sensors and actuators collaborate:

- **LED Bar** acts as a visual indicator of the window's "open level." (e.g., 10 segments lit = fully open, 0 segments lit = fully closed)
- **Light Sensor** detects sunlight intensity. When light is above a threshold, the system decides to "close" the window.
- **Rotary Angle Sensor** can serve as a manual override or a "window position" dial in Codecraft. Turning it adjusts the open/close level (mapped to LED Bar segments).

— Implementation Thoughts. .

- **Mapping Light Sensor to LED Bar:** In Codecraft, read the `lightValue` from `A0` and compare it to a threshold (e.g., 600). If above that threshold, set the LED Bar to fewer segments (closing the window). If below, set more segments (opening it).
- **Integrate Rotary Angle Sensor:** Use the rotary sensor's analog reading (`A3`) as a secondary input. For example, allow the user to override the automatic "close" by turning the dial, which forces the LED Bar to a specified level.
- **Avoid Conflicts:** If both *light-based logic* and *user dial* control the window, define which one has priority or combine them (e.g., the window auto-closes **unless** the user rotates the dial above a certain point).
- **Use a Function Node or Condition Block:**

```
if (lightValue > 600) {
    // it's very bright
    setLEDBarLevel(2); // almost closed
} else {
    // not too bright
    setLEDBarLevel(8); // quite open
}
```

```
// Rotary sensor override
int dialLevel = map(rotaryValue, 0, 1023, 0, 10);
if (userOverride) {
    setLEDBarLevel(dialLevel);
}
```

- **Enhance with More Logic:** For instance, implement a hysteresis or time-based smoothing so the window doesn't rapidly open/close when the light sensor reading hovers around the threshold.

This scenario demonstrates how multiple Grove components—sensors (light, rotary) and actuators (LED bar)—can interact in a single, cohesive project.

— Further Reading

- **Codecraft Documentation**: `https://www.yuque.com/tinkergen-help-en/codecraft?language=en-us`
- **Arduino Reference**: `https://www.arduino.cc/`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the fundamentals of using a block-based development environment (Blockly, via Codecraft) to program an Arduino. We will also cover sensor inputs, analog-to-digital conversion, and discrete outputs (e.g., the LED bar). By understanding the theory behind these abstractions, you can better grasp how visual blocks translate into the underlying C/C++ code on a microcontroller.

A. Block-Based Programming with Blockly

Block-based programming environments like **Codecraft** or **Scratch** allow users to write programs by dragging puzzle-like blocks instead of typing lines of code:

- **Visual Logic:** Each block represents a specific programming construct (loops, conditionals, functions) or hardware action (reading a sensor, setting an LED).
- **Syntax-Free Editing:** Blocks automatically fit together, reducing the chance of syntax errors such as mismatched brackets or missing semicolons.
- **Immediate Feedback:** A real-time "code view" can show the equivalent text-based C/C++ code, helping learners connect visual logic to traditional programming.

A.1 How Codecraft Generates Arduino Code

- **Blockly Parser:** Internally, each block has an XML or JSON definition. Codecraft's engine translates these definitions into C/C++ functions or variables.
- **Arduino Headers:** The generated code automatically includes the correct libraries (`<Arduino.h>`, sensor libraries, etc.).
- **Upload Process:** Codecraft or the user copies this generated code into the Arduino IDE, compiles, then uploads via USB just like any other sketch.

A.1.1 Benefits for Beginners

- **Reduced Cognitive Load:** Students can focus on **logic** and **algorithmic thinking** without worrying about syntax details.
- **Rapid Prototyping:** Quick iteration of ideas—drag new blocks, re-upload, observe changes.
- **Gradual Transition to Text Code:** Seeing both block-based logic and generated text code fosters a smoother learning curve toward C/C++.

B. Sensor Inputs and Analog-to-Digital Conversion (ADC)

Microcontrollers like the Arduino use **Analog-to-Digital Converters (ADCs)** to measure voltage levels from sensors such as **light**, **rotary angle**, or **temperature** sensors:

- **Range:** Typically 0–5 V (on many Arduino boards) is mapped to a digital value (0–1023 for a 10-bit ADC).
- **Resolution:** Each increment of `analogRead()` is approximately 4.9 mV ($5V/1024 \approx 0.0049V$).
- **Sensor Output:** The sensor's internal circuitry provides a variable voltage proportional to the physical quantity (e.g., brightness, angle).

B.1 Rotary Angle and Light Sensors

- **Rotary Potentiometer (Angle Sensor):** A variable resistor whose resistance changes with rotation. The voltage on the output pin shifts accordingly.
- **Light Sensor (Photoresistor or Photodiode):** Its resistance decreases in brighter environments, changing the voltage at the analog pin.

B.1.1 Interpreting Raw ADC Values

- **Calibration:** You can convert the raw ADC reading into a specific range (0–5 V or a computed angle in degrees).
- **Thresholding/Mapping:** If the environment is bright, the raw reading might be near 800–900. You can set conditions (e.g., `if lightValue > 700`).

C. Discrete Outputs: LED Bar

The **LED Bar** is a **digital output** device with multiple LEDs in a row, each of which can be controlled on or off, or in some designs, set to intermediate brightness levels:

- **Driver IC:** Many LED bar modules have an onboard controller that interfaces via a digital pin or SPI-like interface.
- **Addressing Segments:** Code or library calls typically let you light up LED 1 through LED 10 in a pattern.
- **Use Cases:** Visual feedback for sensor reading levels (light meter, volume meter, progress bar, etc.).

C.1 Programming Patterns

- **For Loops:** You can iterate from LED 1 to 10, turning each on in sequence for a "chase" effect.
- **Mapping Analog Value:** If your sensor reading is from 0–1023, you can map that to 0–10 segments. E.g., `segments = map(sensorValue,0,1023,0,10)`.
- **Brightness Control (PWM):** Some LED bar modules allow brightness control of each segment if the driver hardware supports it.

C.1.1 When to Use an LED Bar

- **Visual Cues for Ranges:** Indicate a sensor's approximate magnitude, e.g., 1 LED = low, 10 LEDs = high.
- **Aesthetic/Interaction:** Eye-catching display for interactive projects or quick status checks.

D. Integrating Sensors and Outputs in Codecraft

Using Codecraft blocks, you can quickly link sensor inputs to output devices:

- **Loop Blocks:** Continuously poll the sensor's ADC value.
- **Conditional Blocks:** If the value is above a threshold, light more segments on the LED bar.
- **Math and Map Blocks:** Convert from `0-1023` to `0-10` or other ranges for display.

D.1 Serial Print vs. Visual Feedback

- **Serial Monitor**: Great for debugging or data logging (e.g., `Serial.println(lightValue)`).
- **LED Bar**: Instant real-world feedback. No need to connect to a computer screen to interpret sensor states.

D.1.1 Expanding to Multiple Sensors

- **Ultrasonic Ranger**: Use another block to read distance in cm. Combine that logic with the LED bar to show proximity levels (e.g., fewer segments if object is far away).
- **Light + Rotary Combination**: For more complex tasks, you can combine conditions (e.g., "If it's dark **and** the knob is turned past halfway, light the LED bar fully.").

E. Advantages and Limitations of Block-Based Approaches

- **Advantages:**
  - **Beginner-Friendly**: Users focus on logical connections rather than syntax.
  - **Rapid Prototyping**: Quick iteration cycles, especially for classroom environments.
  - **Teaches Core Concepts**: Even advanced features (loops, conditionals, variables) are visually represented.
- **Limitations:**
  - **Less Flexible for Complex Code**: Large-scale projects might require finer control or advanced libraries not fully supported in block-based tools.
  - **Library Constraints**: Some block environments might not offer all Arduino libraries out of the box, requiring custom expansions or manual code editing.

E.1 Transition to Text Coding

- **Hybrid Workflow**: Many advanced users start with blocks, then switch to text mode for final refinements.
- **Deeper Control**: Understanding the **generated .ino code** helps troubleshoot issues and tap into the Arduino's full feature set.

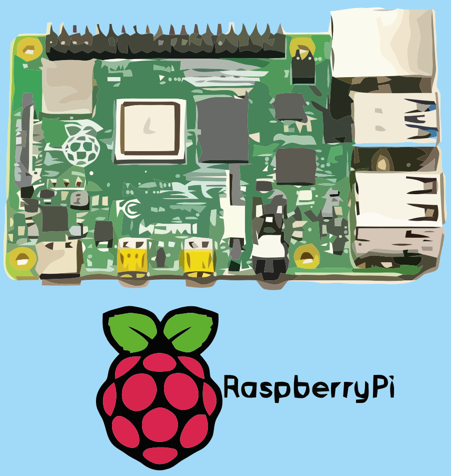
F. Example Use Cases and Extensions

- **Visual Sensor Dashboard**: Use multiple sensors (light, ultrasonic, rotary) and map each reading to an LED bar or an LCD readout.
- **Interactive Art Installation**: Combine block-based logic with creative enclosures or mechanical elements for an exhibit or museum piece.
- **Educational Games**: A "hot/cold" game with an LED bar that lights up more as a seeker gets closer (detected by ultrasonic sensor).

F.1 Next Steps

- **Add Wireless Modules**: Incorporate an ESP8266/ESP32 or Bluetooth module for sending sensor data to a phone or IoT platform.
- **Multi-Node Systems**: Build multiple Arduino nodes, each block-programmed, to experiment with distributed sensing or LED bar arrays.

Final Note.
By leveraging **block-based programming**, you bridge the gap between high-level logic building and hands-on microcontroller coding. Whether reading analog sensors or controlling the LED bar, Codecraft's visual approach promotes rapid learning and experimentation. As you grow more comfortable, exploring the generated C/C++ code can offer deeper insights—empowering you to tackle larger, more advanced Arduino projects with confidence.

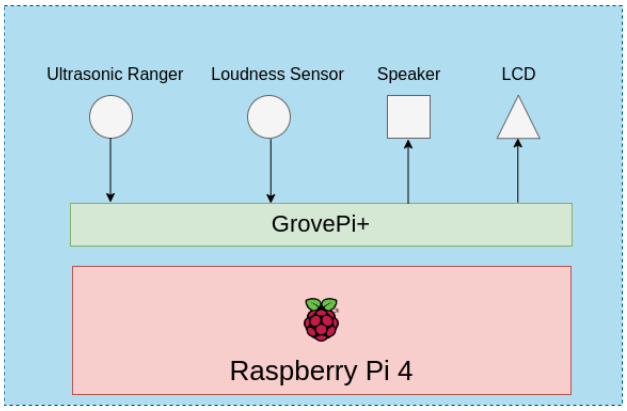


9. Programming Raspberry Pi with Python

Objectives

- Learn how to execute Python scripts on a Raspberry Pi
- Learn how to read data from multiple sensors via Python
- Learn how to utilize a speaker (buzzer) and LED display (LED Bar) together

Lab Plan



In this lab, we will connect Grove sensors to the Raspberry Pi using **GrovePi+** and then create or run Python scripts to interact with them. We'll demonstrate two approaches: creating scripts via the terminal and using the Thonny Python IDE.

Required Hardware Components

- **Raspberry Pi 4** (with an SD card that has Raspberry Pi OS)
- **Display and HDMI cable**
- **Keyboard and mouse**

- **Power supply**

Additionally, we'll use:

- **GrovePi+**
- **Grove Loudness Sensor v0.9b**
- **Grove - LED Bar v2.1**
- **Grove Ultrasonic Ranger v2.0**
- **Grove - Buzzer v1.2**

A. Setting Up the Raspberry Pi & GrovePi+

1. **Attach GrovePi+ to the Pi.**
   Carefully align the 40-pin GPIO header on the Raspberry Pi with the 40-pin connector on the GrovePi+. Gently push the GrovePi+ board down until it sits firmly on the Pi's header pins.

   — Why do this?. The GrovePi+ is an expansion board that maps the Pi's GPIO pins to Grove-compatible ports, making sensor/actuator wiring much simpler. Without it, you'd have to manually connect individual pins for power, ground, and data signals.

2. **Connect Sensors/Modules.**
   Using Grove cables, plug your devices into the GrovePi+ as follows:
   - **Buzzer** $\rightarrow$ **D8** (digital port 8 for generating tones or beeps)
   - **LED Bar** $\rightarrow$ **D5** (digital port 5 supports PWM and can drive the LED bar levels)
   - **Loudness Sensor** $\rightarrow$ **A2** (analog port 2 to read varying voltage levels based on sound intensity)
   - **Ultrasonic Ranger** $\rightarrow$ **D4** (digital port 4 for distance measurements via ultrasound pings)

   Tips:
   - Ensure cables are fully inserted so they do not come loose during testing.
   - Double-check each sensor matches the intended port (digital vs. analog). Plugging an analog sensor into a digital port (or vice versa) can cause incorrect readings or no data at all.

3. **Power On and Check I2C.**

   (a) Connect your Raspberry Pi to a monitor, keyboard, and mouse.
   (b) Attach the Pi's power supply to turn it on. Wait for the OS to boot up.
   (c) Open a terminal window and run:

   ```
   sudo i2cdetect -y 1

   ```

   (d) Look for an address labeled "**04**" in the output. If present, the Pi successfully detects the GrovePi+ board via the I^{2}C bus.

   — Why do this?. GrovePi+ communicates with the Raspberry Pi over I^{2}C. Detecting "04" confirms that:
   - The bus is enabled in `raspi-config` (or **Raspberry Pi Configuration** under **Interfaces**).
   - Your hardware connection is solid.

- You're ready to control sensors and actuators through the GrovePi+ libraries or Node-RED flows.

B. Programming the Buzzer (Terminal Method)

4. **Create a Python Script.**
   Open a terminal on your Raspberry Pi (press `Ctrl + Alt + T`) and type:

```
cd ~
touch buzzer.py

```

This commands the system to:

- `cd` — navigate to your **home directory** (typically `/home/pi`).
- `touch buzzer.py` — create an empty file named `buzzer.py`.

— Why do this?. Organizing your scripts in the home directory (`/home/pi`) keeps them easy to find and manage. Creating an empty file with `touch` gives you a placeholder ready to paste code into.

5. **Copy `buzzer.py` from GitLab.**
   1. Visit the GitLab repository to find the `buzzer.py` code. 2. **Open** the file and copy its contents. 3. In your terminal, type:

```
nano buzzer.py

```

4. **Paste** the code into the Nano text editor, then press `Ctrl + O` to save and `Ctrl + X` to exit.

Tips:

- If you prefer a graphical text editor, open **Thonny** or another editor, then paste the code into `buzzer.py`.
- Ensure the Grove Buzzer is connected to the **D8** port on your GrovePi+ as indicated by the original code.

6. **Run the Script.**
   Finally, execute your code:

```
python3 buzzer.py

```

- You should hear a brief beep from the buzzer.
- Press **Ctrl + C** to stop the script if it loops or continues running.

— Why do this?. Running Python scripts directly in the terminal is a fast way to test hardware interactions without relying on Node-RED or any other graphical tool. Once you confirm the buzzer beeps, you know your wiring, Python environment, and GrovePi+ connections are correct.

C. Programming the LED Bar (Thonny Method)

7. **Open Thonny Python IDE.**
   From the Raspberry Pi's main menu (top-left corner), go to **Programming** → **Thonny Python IDE**, as shown in Figure 9.1.

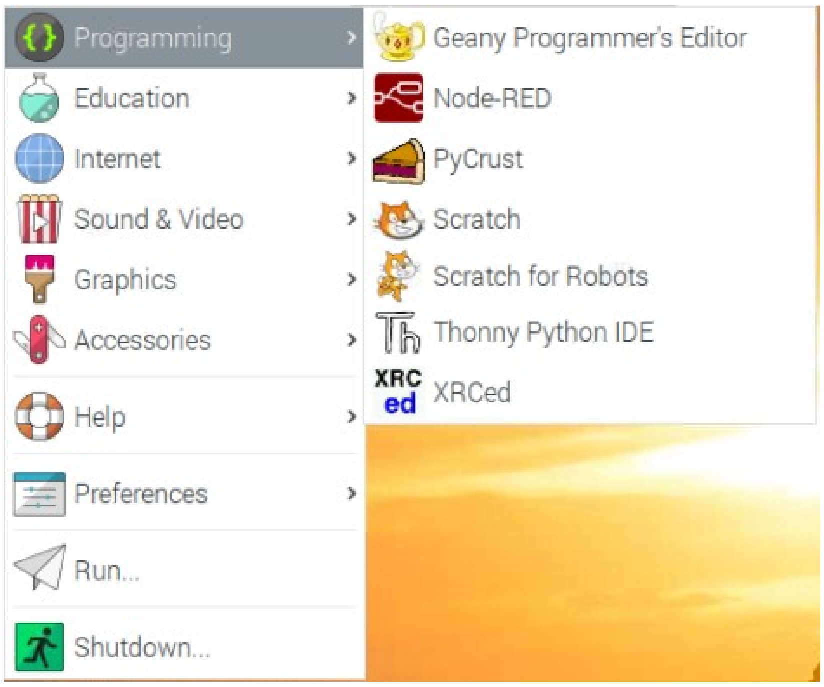
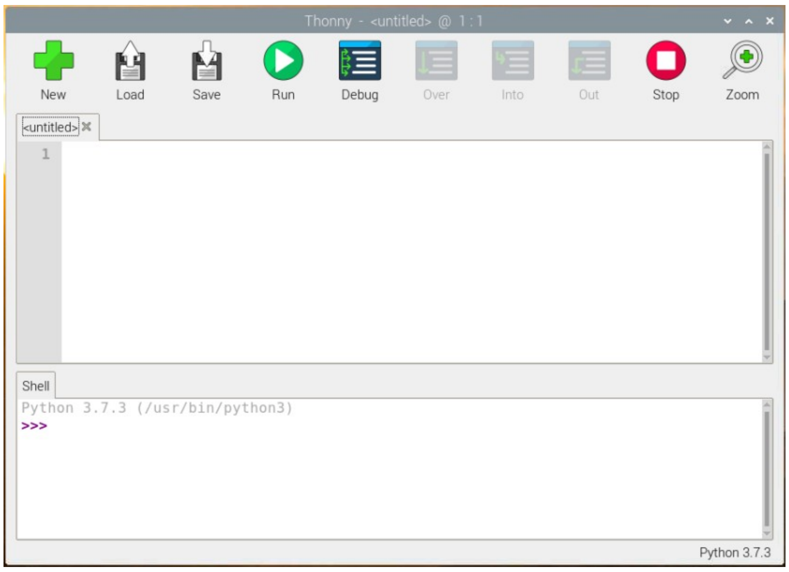


Figure 9.1: Opening Thonny Python IDE

— **Why use Thonny?.** Thonny is a beginner-friendly Python IDE pre-installed on many Raspberry Pi images. It features a clean interface, easy debugging, and straightforward file management, making it ideal for quick development and testing of hardware-interacting scripts.

8. **Create a New Script.**
   Once Thonny launches:
   - Click the **+** icon or go to **File** → **New** to open a blank editor tab (Figure 9.2).
   - Name your file `ledBar.py` (save it in `/home/pi` or a preferred folder).



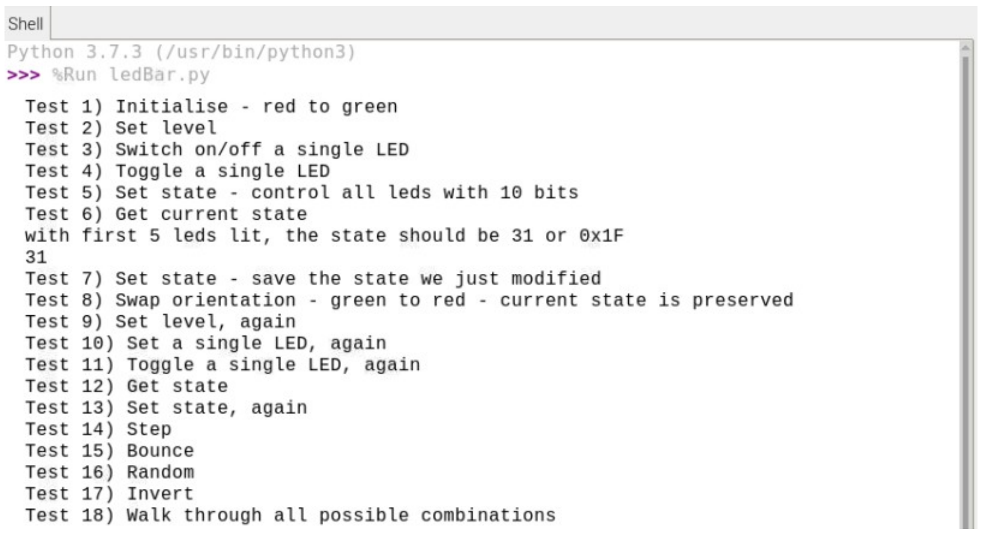
```
Python 3.7.3 (/usr/bin/python3)
>>>
```

Figure 9.2: Creating a New Thonny Python IDE Script

Tips:
- Use concise filenames like `ledBar.py` or `barTest.py` for clarity.
- If you already have a script open, you can open another tab for `ledBar.py` without closing Thonny.

9. **Copy `ledBar.py` from GitLab.**
   Navigate to the lab repository in your browser and open `ledBar.py`:
   - **Copy** its entire content.
   - Return to Thonny, **paste** into your new file (`ledBar.py`).
   - **Save** the file (Ctrl+S or **File** → **Save**).
10. **Run the Script.**

   - In Thonny, click the green **Run** button (or press `F5`).
   - Watch the **LED Bar** as the code cycles through 18 different behaviors/patterns (such as sweeping from 0 to 10 segments, blinking patterns, etc.).
   - Observe the Thonny Shell (output window) as it prints log messages indicating each behavior (Figure 9.3). If you need to stop the program, click the red **Stop** button.



```
Python 3.7.3 (/usr/bin/python3)
>>> %Run ledBar.py
  Test 1) Initialise - red to green
  Test 2) Set level
  Test 3) Switch on/off a single LED
  Test 4) Toggle a single LED
  Test 5) Set state - control all leds with 10 bits
  Test 6) Get current state
  with first 5 leds lit, the state should be 31 or 0x1F
  31
  Test 7) Set state - save the state we just modified
  Test 8) Swap orientation - green to red - current state is preserved
  Test 9) Set level, again
  Test 10) Set a single LED, again
  Test 11) Toggle a single LED, again
  Test 12) Get state
  Test 13) Set state, again
  Test 14) Step
  Test 15) Bounce
  Test 16) Random
  Test 17) Invert
  Test 18) Walk through all possible combinations
```

Figure 9.3: Thonny Python IDE Shell Output

— **Why do this?.** By running the code in Thonny:
- You see immediate feedback in the **shell** regarding each LED Bar step (useful for debugging).
- You can easily edit, re-run, and iterate on your script without leaving the IDE.

Once you confirm the LED Bar lights up correctly and the shell logs match your expectations, you know the GrovePi+ and your wiring are correct.

D. Other Sensors with Python

In addition to the buzzer and LED bar, you can also read data from other Grove sensors using Python scripts on the Raspberry Pi. Below are examples for the Loudness Sensor and Ultrasonic Ranger.

Loudness Sensor

11. **Obtain `loudness.py` from GitLab.**
    In the lab repository, find the file `loudness.py`:
    - **Download or copy** the script to your Pi (e.g., into `/home/pi`).
    - This script assumes the loudness sensor is connected to **A2**.
12. **Run and Test.**
    - In a terminal or Thonny, run:

```
python3 loudness.py
```

    - Clap or produce noise near the loudness sensor. The script prints changing values in real time, reflecting the intensity of the sound.
    - If you see no changes, try:
      - Adjusting the threshold or sensitivity in the script.
      - Moving the sensor to a different analog port (e.g., A1 or A0).

— **Why might it be sensitive?.** Loudness sensors often have a built-in amplifier and can register even small sounds. If it's *too* sensitive, you may see values fluctuate. You can tweak the script's averaging or thresholds to get more stable readings.

Ultrasonic Ranger

13. **Obtain `ultrasonicRanger.py` from GitLab.**
    From the repository, download or copy the script. It references an Ultrasonic Ranger on **D4**.
14. **Run and Observe Distances.**
    - In your chosen environment (terminal or Thonny), type:

```
python3 ultrasonicRanger.py
```

    - The script should continuously print distance in centimeters.
    - Wave your hand or an object closer/farther from the sensor to see the readings update in real time.

Tips:
- If you get sporadic "Out of range" or zero values, ensure there's a clear path in front of the sensor. Ultrasonic modules can be sensitive to angles or soft surfaces.
- Double-check the sensor's connection (D4) and that the script matches that pin configuration.

Measuring Success

- **Buzzer Scripts Work**:
  - Running `buzzer.py` produces an audible beep without any Python errors.
  - Pressing `Ctrl + C` stops the beep, confirming the script and hardware both function.
- **LED Bar Patterns**:
  - Executing `ledBar.py` in Thonny cycles through 18 different patterns.
  - Thonny's output log matches each pattern change, and the LED bar visually reflects these changes on the board.
- **Loudness Sensor Readings**:
  - The `loudness.py` script prints changing numeric values when you clap or produce

noise near the sensor.
    - Values remain stable in a quiet environment, demonstrating correct sensitivity.
- **Ultrasonic Distance**:
    - `ultrasonicRanger.py` continuously prints distance measurements (in cm).
    - Moving an object closer or farther produces predictable changes in the displayed values, confirming accurate ultrasonic readings.

■

Use Case Scenario (Optional):
Imagine turning your Raspberry Pi + GrovePi+ into a basic *intrusion detection system*, where multiple sensors work together to detect unusual activity:

- **LED Bar (Sound Level Indicator):** The loudness sensor reads ambient noise. Map its values to the LED bar so that more segments light up if the environment gets louder (e.g., an intruder making noise).
- **Ultrasonic Ranger (Door Monitor):** Attach the ultrasonic sensor near a door or hallway. If the measured distance changes suddenly (e.g., from 30 cm to 10 cm), assume the door opened or someone passed by. Trigger the buzzer as an alarm.
- **Combining Sensor Readings:** In your Python script (or Node-RED flow), you can:
    - Continuously read the `loudness` and `ultrasonic` sensors.
    - Update the LED bar to reflect current noise level.
    - If the ultrasonic distance changes abruptly, activate the buzzer.
    - (Optional) Log these events to a file or cloud service so you can review timestamps of "suspicious" noises or door openings.
- **Example Pseudocode:**

```
while True:
    soundValue = readLoudness()          # Range depends on the sensor
    distance = readUltrasonic()          # Distance in cm
    setLEDBarLevel(map(soundValue, 0, 600, 0, 10))

    if distance < 10:
        buzzerOn()
    else:
        buzzerOff()

    time.sleep(0.2)
```

- **Enhancements:**
    - **Threshold Tuning**: Adjust the noise mapping or the distance threshold to reduce false alarms.
    - **Logging/Alerts**: Save the events to a file (`intruder.log`) or send an email/SMS using Python libraries if a real intrusion is detected.
    - **Integration with Other Sensors**: Incorporate a PIR sensor or camera module for more sophisticated detection.

This scenario showcases how multiple sensors (loudness, ultrasonic) can interact in a single script to create a rudimentary security application, complete with real-time visual feedback (LED bar) and a buzzer alarm.

— Further Reading

- More Python examples with GrovePi:
  `https://github.com/DexterInd/GrovePi/tree/master/Software/Python`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the concepts behind programming a Raspberry Pi with Python, focusing on sensor reading (via analog and digital pins) and controlling actuators. We will also discuss how Python scripts interact with hardware through libraries like GrovePi+, and how integrated development environments (IDEs) such as Thonny streamline the development process.

A. Python on the Raspberry Pi

Python is a high-level, interpreted language that comes preinstalled on many Raspberry Pi OS distributions:

- **Interpreted Execution:** You can run code immediately without a compile step, making iteration fast and convenient.
- **Rich Ecosystem:** Access to thousands of libraries (e.g., NumPy, OpenCV) for data processing, machine learning, or networking.
- **Beginner-Friendly**: The syntax is more readable compared to lower-level languages like C/C++.

A.1 Why Python for IoT?

- **Fast Prototyping:** You can quickly write scripts to read sensors or control outputs.
- **Large Community**: Many IoT libraries and tutorials exist, helping you integrate sensors, cloud services, or GUIs.
- **Integration with Node-RED, Docker, etc.**: The Pi can run multiple services in parallel (Node-RED flows, custom Python scripts, databases).

A.1.1 Terminal vs. IDE Approaches

- **Terminal Method**: Use a text editor (e.g., nano, vim) to write Python scripts, then run with `python3 script.py`.
- **IDE Method (Thonny)**: Offers a GUI with syntax highlighting, debugging tools, and a built-in Python shell—well-suited for beginners.

B. GrovePi+ and Hardware Abstraction

GrovePi+ is an add-on board that sits on the Pi's 40-pin GPIO header:

- **I2C Interface**: The Pi communicates with the GrovePi+ over I2C, and the GrovePi+ manages analog/digital pins on behalf of the Pi.
- **Convenient Ports**: Each sensor/actuator plug is labeled (D for digital, A for analog, I2C for specialized sensors).
- **Library Support**: By installing `grovepi` or `grove.py` Python packages, you get high-level functions like `grovepi.analogRead()` or `grovepi.pinMode()`.

B.1 Analog, Digital, and I2C Sensors

- **Analog Sensors (e.g., Loudness Sensor)**: Provide a voltage proportional to the measured quantity (sound level, light intensity, etc.). The GrovePi+ includes onboard ADC capability.
- **Digital Sensors (e.g., Buzzer, Ultrasonic Ranger)**: Output or accept simple HIGH-/LOW signals or perform digital pulse measurement (e.g., ultrasonic distance).
- **I2C Sensors**: Communicate over I2C lines (SDA, SCL). Examples include temperature/humidity or OLED displays.

B.1.1 Why Use GrovePi+?

- **Simplicity**: Fewer wiring errors thanks to standardized connectors.
- **Modularity**: Quickly swap sensors without rewiring breadboards.
- **Educational Focus**: Ideal for labs where multiple sensor types are tested in short sessions.

C. Python Scripts for Sensor Reading and Actuation

When you write a Python script (e.g., `buzzer.py` or `loudness.py`), you typically:

- **Import Libraries**: `import grovepi`, `import time`, etc.
- **Set Pin Modes**: `grovepi.pinMode(pin, 'OUTPUT')` or `grovepi.pinMode(pin, 'INPUT')`.
- **Main Loop**: Continuously read or write values, possibly with `time.sleep()` in between.

C.1 Buzzer Control

- **Digital Write**: Sending a digital HIGH or LOW to turn the buzzer on/off. Some buzzers may produce a single tone.
- **PWM (Optional)**: Some advanced scripts might use software PWM to produce different beep patterns or frequencies, but that depends on the library support.

C.1.1 Typical Usage Pattern

```python
import grovepi
import time

buzzer_pin = 8
grovepi.pinMode(buzzer_pin,"OUTPUT")

while True:
    grovepi.digitalWrite(buzzer_pin,1)  # ON
    time.sleep(1)
    grovepi.digitalWrite(buzzer_pin,0)  # OFF
    time.sleep(1)
```

D. LED Bar: Discrete or Pattern-Based

An **LED Bar** typically has 10 segments that can be individually controlled:

- **Driver Interface**: The Grove LED Bar might use a specialized driver chip accessible via `grove_led_bar.py` or a similar library.
- **Set Level**: A function like `ledBar.setLevel(n)` might light up the first `n` segments.
- **Patterns**: Some scripts cycle through 0–10 segments or alternate in interesting sequences.

D.1 Linking Sensors to the LED Bar

- **Mapping Values**: E.g., for a loudness sensor reading from 0–1023, you can map it to 0–10 segments. If loudness is 512, you might light up 5 segments.
- **Visual Thresholds**: If distance $< 10\,\text{cm}$ (ultrasonic), light all segments to indicate "too close."

E. Combining Multiple Sensors in Python

Multiple sensor scripts can be merged into one. For instance:

- **Read Loudness** on A2, **read Distance** on D4, **control LED Bar** on D5, **buzzer** on D8.
- **Single Main Loop**: Each iteration checks loudness, checks distance, updates the LED bar, possibly triggers the buzzer.

E.1 Handling Real-Time Constraints

- **Loop Frequency**: If you read sensors too rapidly, you may overload the I2C bus or print too many lines in the console.
- **Delays or Sleep**: Using `time.sleep(0.1)` or `time.sleep(1)` can help regulate the loop rate.

F. Thonny IDE vs. Command-Line Workflow

Thonny provides a graphical interface for Python on the Pi:

- **One-Click Run**: Easy to test code without switching to a terminal.
- **Built-In Debugger**: Step through code, inspect variables, set breakpoints.
- **Beginner-Friendly Interface**: Hides some complexities of Python environments, but advanced users might prefer command-line editors like `vim` or `nano`.

F.1 Adapting to Larger Projects

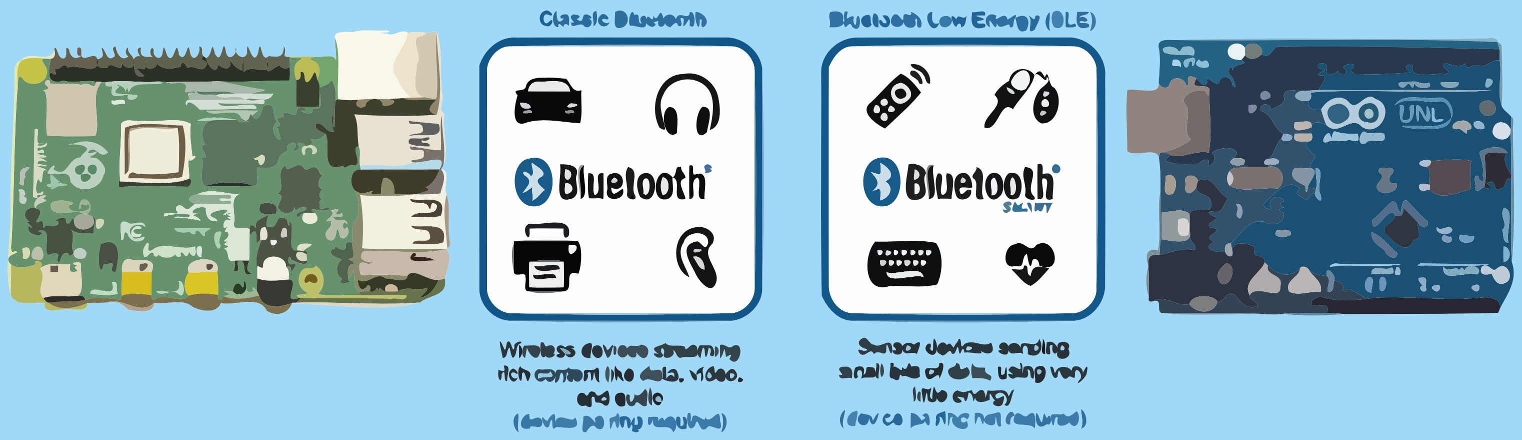
- **Organize Modules**: As your project grows, place sensor logic in separate Python files (modules) and import them in your main script.
- **Version Control (Git)**: For multi-file projects, using Git (even on the Pi) is recommended for tracking changes and collaborating.

G. Example Applications and Next Steps

- **IoT Integration**: Combine Python scripts with cloud services (MQTT, HTTP). For instance, send loudness or ultrasonic readings to an IoT dashboard.
- **Local Data Logging**: Store sensor data in a CSV or an SQLite database for later analysis.
- **Machine Learning on Pi**: With libraries like `tensorflow-lite` or `scikit-learn`, you can run simple ML models locally.
- **GUI with Tkinter or PyQt**: Create local GUIs to display sensor readings in real time without needing an external monitor or a separate server.

Final Note.

Programming the Raspberry Pi with Python opens up a world of possibilities for embedded systems. With libraries like **grovepi** to simplify sensor/actuator communication, you can rapidly develop prototypes or advanced projects—controlling buzzers, LED bars, reading distance or loudness, and beyond. As you expand your skillset, consider integrating more sensors, connecting to the cloud, or applying data analysis techniques for robust, real-world IoT solutions.

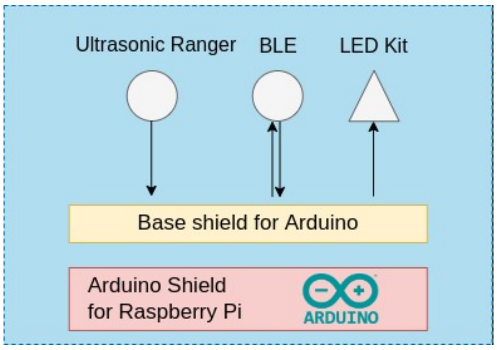

Classic Bluetooth
Bluetooth
Wireless devices streaming rich content like data, video, and audio
(device pairing required)
Bluetooth Low Energy (BLE)
Bluetooth SMART
Sensor devices sending small bits of data, using very little energy
(device pairing not required)
UNL

10. Bluetooth Low Energy (BLE) Based Systems

Objectives

- Learn how to configure a BLE module
- Learn how to use a BLE application on your phone
- Learn how to transmit sensor data over BLE
- Learn how to display or receive data from a BLE device

Lab Plan



In this lab, we will establish a Bluetooth Low Energy (BLE) connection between an "edge development board" (Arduino-like device) and a mobile phone. We will use AT commands (Hayes command set) to configure the BLE module and then transmit sensor data for display on the phone.

Required Hardware Components

- **Arduino Shield for Raspberry Pi** (which has an onboard Arduino Leonardo chip)
- **Grove Base Shield**
- **Grove BLE v1.0**
- **Grove Ultrasonic Ranger v2.0**

- **Grove LED Socket Kit v1.5**
- **Loudness Sensor**

A. Arduino Shield Setup

1. **Identify the Arduino Shield for RPi.**
   You'll be working with a specialized Arduino shield for the Raspberry Pi, often called an "Arduino Shield for Raspberry Pi." It features an **Arduino Leonardo** chip on-board, meaning you can treat it as a standard Arduino when programming (i.e., select "Arduino Leonardo" in the IDE).

   > **— Why does this matter?.** The shield effectively combines a microcontroller (Leonardo) with the Raspberry Pi's pinout, enabling you to use Arduino libraries and sketches while physically attached to the Pi. It's ideal for bridging Raspberry Pi projects (Python, Node-RED, etc.) with the Arduino ecosystem.

2. **Attach the Base Shield & Sensors.**
   After identifying the shield, connect a Grove Base Shield on top (if provided) or directly plug in the Grove cables to the shield's Grove ports:
   - **BLE → UART**
     (for Bluetooth Low Energy communication)
   - **LED Kit → D5**
     (digital pin 5, supports PWM if needed)
   - **Loudness Sensor → A2**
     (analog input for reading sound intensity)
   - **Ultrasonic Ranger → D4**
     (digital pin for distance measurements)

   > **Tips:**
   > - Double-check polarity and labels on each Grove connector to ensure correct orientation.
   > - If your shield has a voltage switch (3.3V/5V), verify it's set appropriately for your sensors (most Grove modules require 5V).
   >
   > ■

3. **Connect to PC.**
   To program the shield's **Leonardo** microcontroller:
   - Use a **USB cable** from the shield's micro USB port to your PC.
   - Launch the Arduino IDE on your computer.
   - In the **Tools** menu, select:
     Board *Arduino Leonardo*
     Port The corresponding COM port (Windows) or `/dev/ttyACM*` (Linux/macOS).
   - Click **File → New**, then upload an empty sketch (contains only `setup()` and `loop()`) to confirm communication works.

   > **— Why do an empty sketch?.** Uploading a blank sketch is a quick test to ensure:
   > - Your PC recognizes the shield as an `Arduino Leonardo` device.
   > - The USB cable and drivers are functioning correctly.
   > - You can safely proceed to coding sensors and BLE communication without initial IDE errors.

B. BLE Code and AT Commands

4. **Understanding the Code.**
   The Arduino Leonardo core on this shield supports two serial ports:
   - `Serial` — communicates via USB to your PC (for the Serial Monitor).
   - `Serial1` — hardware UART pins that connect to the onboard BLE module (or other devices).

   Download or copy the code from GitLab into the Arduino IDE and **upload** it.

   > **— Why Serial vs. Serial1?.** • **Serial (USB)**: For debugging and sending commands via the Arduino IDE's Serial Monitor.
   > - **Serial1 (UART pins)**: Communicates with the BLE module, letting the sketch send AT commands and exchange data wirelessly.
   >
   > Knowing these distinct serial ports helps you route different commands to the correct destination.

5. **Optional `while(!Serial);` Line.**
   Inside `setup()`, you might see:

```
while(!Serial);
```

   This line *pauses* the code until a Serial connection (the USB) is opened. It's great for development (ensuring you can read all startup messages in the Serial Monitor). But if you plan to deploy the Arduino in a standalone scenario (no USB attached), **remove or comment out** this line so the program runs immediately.

   > **Tips:**
   > - If you see "*Nothing happens until I open the Serial Monitor*," that's usually due to `while(!Serial);`.
   > - For headless (production) setups, skipping this line ensures your BLE code starts right away.

6. **Open Serial Monitor.**
   After uploading, open the **Serial Monitor** (or any serial terminal at the correct baud rate). You should see something like:

```
OKOK+Set:0OK+Set:Ratata
```

   This indicates the BLE module received AT commands from `setup()` and responded. In the example code, you might see lines like:

```
Serial1.print("AT");          // Check BLE comm
Serial1.print("AT+ROLE0");    // Set BLE peripheral role
Serial1.print("AT+NAMERatata"); // Rename module to 'Ratata'
```

   Each `Serial1.print(...)` triggers a configuration command on the BLE module.

   > **— Why AT Commands?.** AT commands date back to the Hayes modem era and remain a straightforward way to configure many serial-based modules, including some BLE devices. Here, they let you set the module's role, name, baud rate, etc.

7. **Communication Settings.**
   In the Arduino IDE's Serial Monitor, set:

- **Line Ending**: *No line ending* (as shown in Figure 10.1).
- **Baud Rate**: Match the `Serial.begin(...)` in your code (often 9600 or 115200).

Type `AT` and press Enter. The module should reply with `OK`, confirming that:

- Your BLE module is listening on `Serial1`.
- You can now send further commands interactively if desired.

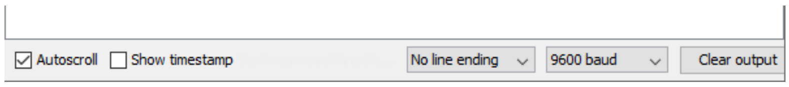


Figure 10.1: Communication Format in Serial Monitor

8. **AT Command Reference.**
BLE modules can vary widely in their AT command sets. Our module's reference is at: `https://files.seeedstudio.com/wiki/Grove-BLE_v1/res/Bluetooth4_en.pdf`. Typical commands include:
   - `AT+ROLE0` or `AT+ROLE1` to set **peripheral** vs. **central** roles.
   - `AT+NAMEYourName` to rename the module for easy discovery.
   - `AT+BAUD1,2,4` etc. to change baud rate.

   Always consult the specific datasheet or command set for your exact BLE hardware.

   Tips:
   - If a command returns `ERROR`, check for typos or confirm the module supports that command.
   - Some modules auto-save settings to flash; others require an explicit save command (e.g., `AT+SAVE`).

C. Connecting the Phone App (BLE Central)

9. **Download Serial Bluetooth Terminal.**
On your Android device, open Google Play Store and search for *Serial Bluetooth Terminal*. Install it to handle BLE (Bluetooth Low Energy) connections in a terminal-like fashion.

   — Why this App?. Unlike classic Bluetooth terminal apps, *Serial Bluetooth Terminal* includes BLE support. Many simpler terminal apps only handle "Classic" Bluetooth (Bluetooth 2.0). This app can discover, connect, and exchange data with BLE devices like your Grove BLE module.

10. **Scan for BLE Devices.**
1. Launch *Serial Bluetooth Terminal*. 2. Turn on Bluetooth if prompted, then look for a scanning menu (often labeled **Devices**). 3. Choose **Bluetooth LE** to list BLE devices specifically (not "Classic"). 4. If you uploaded code that names your module via

```
Serial1.print("AT+NAMERatata");

```

   it should appear with that name (e.g., "Ratata"). 5. If multiple BLE modules show up, rename yours to something unique (e.g., `AT+NAMEMyGroveBLE`) so you don't connect to the wrong device.

Tips:
- BLE scanning might take a few seconds. If you don't see your device, toggle Bluetooth off/on, or move the phone closer to the module.
- Ensure the BLE module's LED is still blinking slowly, indicating it's in a *peripheral, waiting-to-connect* mode.

11. **Connect and Send Data.**
After the app finishes scanning:
    - **Tap the device name** that matches your BLE module (e.g., `Ratata`).
    - The module's LED should go from *blinking* (disconnected) to *steady on* (connected).
    - In the app's terminal field, type "Hello World!" and press **send**.
    - Check your Arduino IDE **Serial Monitor**. If your code forwards incoming BLE data to `Serial`, you should see "Hello World!" appear there.

— Why do this?. This confirms full two-way communication:
- **Phone (central)** ↔ **BLE module (peripheral)** ↔ **Arduino code (Serial1)** ↔ **USB Serial Monitor**.
- Anything typed in the phone app is relayed to the Arduino via BLE, and the Arduino can respond back, which you can see on the phone's screen.

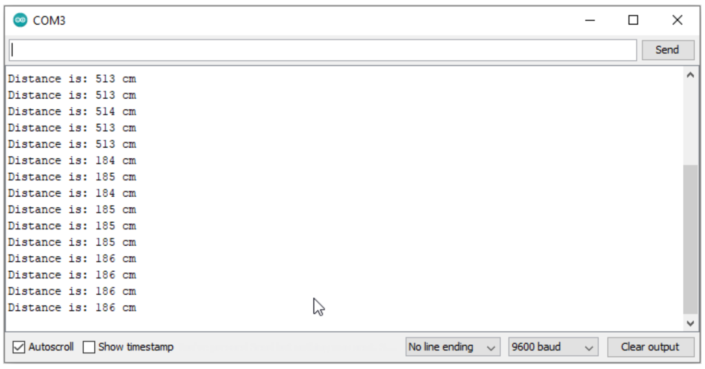
Such a setup is typical of BLE-based projects, where a phone app acts as the "central" controlling or reading data from a peripheral device.

D. Transmitting Sensor Data Over BLE

12. **Build an Alarm Example.**
Let's imagine a "storm alarm" scenario using the ultrasonic sensor data:
    - **LED On**: The storm is *very close* (ultrasonic distance is below a small threshold, e.g., < 10 cm).
    - **LED Blinks**: The storm is *coming* (distance is in a medium range, e.g., 10–20 cm).
    - **LED Off**: No sign of storm (distance above 20 cm).

    You can easily adapt these thresholds to suit different detection ranges.

13. **Upload the `bleSensor.ino`.**
1. Go to the GitLab repository and copy the `bleSensor.ino` code. 2. In your Arduino IDE, create a new sketch and **paste** the code. 3. **Verify** and **Upload** to your Arduino. 4. Open the Arduino Serial Monitor to see distance data printed (Figure 10.2).

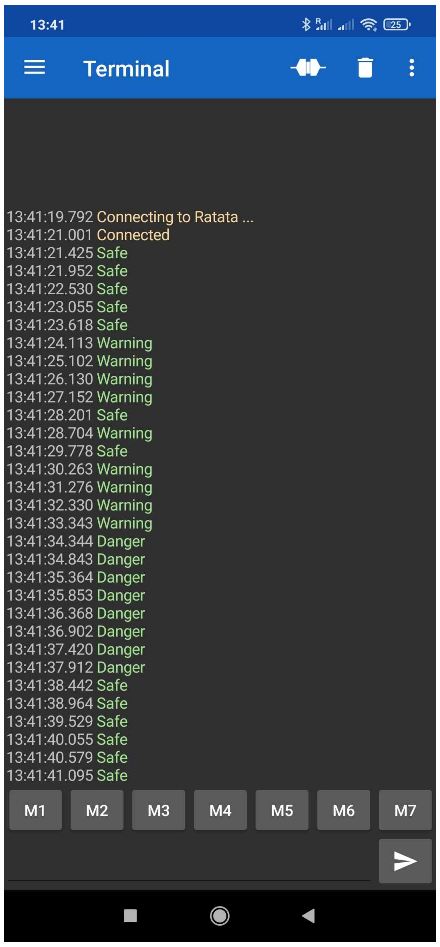


```
Distance is: 513 cm
Distance is: 513 cm
Distance is: 514 cm
Distance is: 513 cm
Distance is: 513 cm
Distance is: 184 cm
Distance is: 185 cm
Distance is: 184 cm
Distance is: 185 cm
Distance is: 185 cm
Distance is: 185 cm
Distance is: 186 cm
Distance is: 186 cm
Distance is: 186 cm
Distance is: 186 cm
```

Figure 10.2: Distance on Serial Monitor

— How the Code Works.
- **Ultrasonic Reading**: The code continuously measures distance via an ultrasonic sensor (e.g., on D4) and calculates the distance in centimeters.
- **BLE Transmission**: Each reading is sent over `Serial1` to the BLE module, which broadcasts it to any connected central device (e.g., your phone).
- **LED Control**: Depending on the distance, the code may turn the LED fully on, blink it, or turn it off—simulating a storm alarm.

14. **Observe the Data in Phone App.**
    1. Reopen your *Serial Bluetooth Terminal* (or similar BLE app) on your phone. 2. Reconnect to the BLE module (e.g., `Ratata`). 3. Wave your hand or an object in front of the ultrasonic sensor.
    - You should see updates (distance values, alarm states) in the phone app console, as shown in Figure 10.3.
    - The Arduino is effectively broadcasting sensor data to the phone over BLE.



```
13:41:19.792 Connecting to Ratata ...
13:41:21.001 Connected
13:41:21.425 Safe
13:41:21.952 Safe
13:41:22.530 Safe
13:41:23.055 Safe
13:41:23.618 Safe
13:41:24.113 Warning
13:41:25.102 Warning
13:41:26.130 Warning
13:41:27.152 Warning
13:41:28.201 Safe
13:41:28.704 Warning
13:41:29.778 Safe
13:41:30.263 Warning
13:41:31.276 Warning
13:41:32.330 Warning
13:41:33.343 Warning
13:41:34.344 Danger
13:41:34.843 Danger
13:41:35.364 Danger
13:41:35.853 Danger
13:41:36.368 Danger
13:41:36.902 Danger
13:41:37.420 Danger
13:41:37.912 Danger
13:41:38.442 Safe
13:41:38.964 Safe
13:41:39.529 Safe
13:41:40.055 Safe
13:41:40.579 Safe
13:41:41.095 Safe
```

Figure 10.3: Reading Data via App

Tips:
- **Adjust Thresholds**: If you find the LED switching states too frequently, tweak the near/medium/far boundaries in code.

- **Buzzer or Other Actuators**: You can easily replace or supplement the LED with a buzzer or servo to create more elaborate warnings (e.g., sound alarm or physically move a barrier).
- **Bi-Directional Communication**: Expand the sketch to receive phone commands (e.g., “Silence alarm”), so the phone can send “stop” messages to the Arduino.

Measuring Success

- **BLE Peripheral Up & Running**: The Arduino code prints “OK” or similar acknowledgments from the BLE module. On your phone, the module’s name (e.g., “Ratata”) appears in the BLE scanning list, confirming proper advertising and peripheral configuration.
- **Data Exchange**: When you send a message like “Hello World!” from the *Serial Bluetooth Terminal* app, the Arduino’s Serial Monitor shows it arriving through `Serial1`. Conversely, any Arduino data sent to `Serial1` is visible in the phone’s BLE terminal.
- **Sensor Data Over BLE**: If your code transmits ultrasonic distance or loudness values over `Serial1`, these readings appear in real time on the phone app. Moving your hand near the ultrasonic sensor or clapping near the loudness sensor yields immediate changes.
- **LED or Alarm Logic**: The LED toggles or blinks based on sensor thresholds (e.g., when ultrasonic distance is below a certain limit). Watching the LED react physically while seeing distance changes in the phone app confirms end-to-end logic (sensor $\rightarrow$ Arduino $\rightarrow$ BLE $\rightarrow$ phone, plus local actuator responses).

Use Case Scenario (Optional): “Advanced Storm Warning”
Envision a more detailed application where the **ultrasonic sensor** monitors an approaching “storm,” and the system communicates this status via LED blinking and BLE messages to your phone:

- **LED Blinking Rate**: The closer the “storm” (i.e., the smaller the ultrasonic distance), the faster the LED blinks:
  - **Distance > 80 cm**: LED off or slow blink every 2 s.
  - **Distance 50–80 cm**: Medium blink rate (e.g., 1 Hz).
  - **Distance 20–50 cm**: Faster blink (e.g., 2 Hz).
  - **Distance < 20 cm**: LED stays on (no blinking) or uses a rapid strobe to indicate imminent “storm.”
- **Phone Receives Distance Over BLE**:
  - **Send sensor readings**: In your Arduino code, transmit the ultrasonic distance to the BLE module every 500 ms (or a suitable interval).
  - **Print messages**: On your phone’s *Serial Bluetooth Terminal* app, you see lines like:

    ```
    Storm is 80 cm away!
    Storm is 65 cm away!
    Storm is 23 cm away!
    ```

  - **Optional thresholds**: The Arduino code can categorize the distance into 5 distinct messages (e.g., “Storm > 80 cm: safe,” “50–80 cm: caution,” “20–50 cm: alert,” etc.).
- **Implementation Hints**:
  1. **Ultrasonic Reading**: Continuously poll the sensor (e.g., `distance = readUltrasonic();`).
  2. **LED Blink Logic**: Use `millis()` or a small delay to handle variable blinking speeds:

     ```
     if (distance < 20) {
        ledOn();
     } else if (distance < 50) {
        blinkLed(2); // 2 Hz
     ```

```
} else if (distance < 80) {
   blinkLed(1); // 1 Hz
} else {
   blinkLed(0.5); // or turn off
}
```

3. **BLE Transmission**: Convert the numeric distance to a human-readable message, then `Serial1.print("Storm is " + distance + " cm away!`
`n");`.
4. **Phone Display**: The phone app will show these lines, giving real-time feedback on the storm's proximity.

By combining conditional LED blinking and BLE data broadcasts, you create a comprehensive "storm warning" system:

- **Visual Indicator**: LED light signals urgency level based on distance.
- **Data Logging / Monitoring**: The phone (central device) displays detailed distance messages, possibly logging them or forwarding them to another service.

This approach demonstrates how Arduino-based BLE peripherals can offer both local physical feedback and remote observation in a single integrated solution.

— Further Reading

- Learn more about BLE:
  `https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch01.html`
- Learn more about Hayes command set:
  `https://en.wikipedia.org/wiki/Hayes_command_set`

Theory Deep Dive: Underlying Principles and Concepts

In this section, we delve into the key principles behind Bluetooth Low Energy (BLE), how AT commands configure BLE modules, and how data is exchanged between a peripheral (the Arduino-like device) and a central device (the phone). By understanding these underlying mechanisms, you will gain deeper insight into building more robust and flexible BLE-based applications.

A. What Is Bluetooth Low Energy?

Bluetooth Low Energy (BLE) is a wireless communication technology designed for low-power operations and short-range data transfer:

- **Low Energy Consumption:** Ideal for battery-powered IoT devices (sensors, wearables, etc.).
- **Short Range:** Typically up to 10–50 meters under most conditions.
- **Profiles/GATT Services:** BLE organizes data around **services**, each containing **characteristics** that represent specific data points (e.g., temperature, heart rate, etc.).

A.1 BLE vs. Classic Bluetooth

- **BLE (Bluetooth 4.0+)**: Focuses on low power, smaller data packets, ideal for sensor readings.

- **Classic Bluetooth**: Supports larger throughput (audio streaming, file transfer) but consumes more power.
- **Device Roles**: In BLE, we commonly talk about **peripherals** (devices advertising data) and **centrals** (devices scanning and connecting, e.g., phones).

A.1.1 Why Use BLE?

- **Ubiquitous Phone Support**: Most smartphones/tablets can act as BLE central devices.
- **Low Cost/Low Power**: Great for small devices, sensor nodes, wearable tech.
- **Flexible Data Exchange**: You can create custom GATT services or use standard ones (e.g., battery, heart rate).

B. AT Commands and Module Configuration

AT commands are a set of human-readable instructions (e.g., `AT+NAME=Ratata`) that configure modems or Bluetooth modules:

- **Hayes Command Set**: Originally designed for telephone modems, widely adopted for various modules.
- **Parameter Adjustment**: Examples include changing the device name, baud rate, power level, or role (peripheral vs. central).
- **Interactive Debugging**: You can send commands over a serial connection (e.g., `Serial1`) and see text-based "OK" or error replies.

B.1 Typical Commands

- `AT`: Basic check—module should reply with `OK` if it's alive.
- `AT+ROLE0`: Switches the module to **peripheral** mode (advertises itself).
- `AT+NAMExxxx`: Sets the device name to something like "Ratata" or "MyBLEDevice."
- `AT+RESET`: Reboots the module, applying all new settings.

B.1.1 Arduino BLE Sketch Integration

- **Serial vs. Serial1**: Often, `Serial` is the USB port to your PC, and `Serial1` is the UART pins to the BLE module.
- **Automated Configuration**: Code can send AT commands in `setup()` to set the BLE name, role, or other parameters before proceeding to sensor readings.

C. BLE Peripheral vs. Central Roles

Peripheral Devices (e.g., your Arduino) advertise data or wait for connections:

- **Advertising Packets**: Basic info like device name, services offered.
- **Connection Acceptance**: Once a central initiates a connection, the peripheral can start transmitting more detailed data (e.g., sensor values).

C.1 Central Devices (Phones/Tablets)

- **Scan for BLE Devices**: Typically sees "Ratata" or whatever name the module advertises.
- **Connect & Discover Services**: The phone queries characteristics to read or write data.
- **BLE Apps**: Tools like "Serial Bluetooth Terminal" or specialized apps can parse data, show logs, or forward it to other services.

C.1.1 Data Exchange Mechanics

- **Characteristic Reads**: Central requests data from the peripheral (pull model).
- **Notifications/Indications**: Peripheral pushes data changes to the central without repeated polling (push model).
- **Write Commands**: Central can change a peripheral's parameter or send instructions (e.g., turning on an LED).

D. Reading Sensors and Sending Data Over BLE

When your Arduino code has sensor logic (ultrasonic, loudness) and a BLE "link," it can:

- **Poll Sensors**: `distance = ultrasonicRead(D4);` or `soundValue = analogRead(A2);` in a loop.
- **Format Data**: Convert sensor readings into a string (e.g., "Distance: 80 cm\n").
- **Transmit via Serial1**: Call `Serial1.print(...)` to send data to the BLE module, which relays it to the phone.

D.1 Handling Timing and Buffering

- **Loop Intervals**: Overly frequent transmissions can congest your BLE link or drain more power.
- **Batching vs. Immediate Send**: Some sketches may buffer readings and send them once per second rather than 100 times per second.

D.1.1 Bi-Directional Control

- **LED Control from Phone**: Type "LED ON" in the phone app; the Arduino sees `"LED ON"`, toggles a pin high.
- **Confirmations Back**: The Arduino can respond with "LED is now ON," giving feedback to the user on the phone.

E. BLE Security and Pairing

BLE Security can include:

- **Just Works**: No passkey required—good for quick setups, but less secure.
- **Passkey or Numeric Comparison**: The phone displays a code to confirm with the peripheral, or vice versa.
- **LE Secure Connections**: Stronger encryption methods introduced in Bluetooth 4.2 and 5.0.

E.1 Module-Specific Settings

- **AT+PASSxxxxxxxx**: Some modules allow setting a passkey. Others might rely on a default code (e.g., 0000 or 1234) or have no passkey at all.
- **Bonding/Pairing States**: Once paired, the phone might automatically reconnect in the future without re-entering commands.

F. Potential Limitations and Alternatives

- **Limited Throughput**: BLE typically transmits at tens of kilobytes/second, which is fine for sensors but not for high-bandwidth tasks (like music streaming).
- **Range Variability**: Walls and interference can reduce real-world range significantly.

- **Alternatives**: Wi-Fi modules (ESP8266/ESP32) can provide higher throughput and networking features, but at higher power consumption.

F.1 Advanced BLE Functionality

- **Custom GATT Services**: Instead of sending plain text, you can define a custom BLE characteristic to hold sensor data in a structured format (bytes, floats).
- **Notifications**: Let the phone subscribe for updates rather than continuously polling the peripheral.
- **Multiple Connections**: Some BLE modules can connect to multiple centrals or peripheral devices at once, though it's more common for a single peripheral–central link.

G. Example Use Cases and Future Directions

- **Wearable Fitness Tracker**: Reads heart rate sensor, broadcasts data to a phone app.
- **Smart Home Sensor Nodes**: Each BLE node measures temperature/humidity, sends to a phone or gateway, which logs or visualizes data.
- **Event-Driven Alarms**: A loudness sensor or PIR motion sensor triggers an immediate BLE push to the user's phone, alerting them to intrusions or anomalies.
- **Multi-Node Mesh**: BLE mesh protocols (Bluetooth 5.0+ feature) extend coverage by relaying data between nodes.

Final Note.

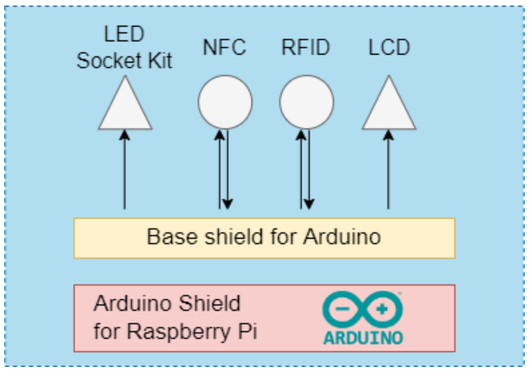
By configuring your BLE module with **AT commands** and leveraging the **peripheral–central** model, you can create flexible, low-power wireless links for sensor data, control signals, or user alerts. Whether you're toggling an LED from a phone app or streaming ultrasonic readings, BLE's low energy profile and wide smartphone compatibility make it a powerful choice for small-scale IoT projects. Experiment with custom GATT services, secure pairing, or advanced notification schemes to unlock even more robust Bluetooth solutions.

11. RFID and NFC Based Tracking

Objectives

- Learn how to establish NFC communication
- Learn how to use NFC tags
- Learn how to use an RFID reader

Lab Plan



In this lab, we will learn how to utilize both NFC and RFID modules. We will connect them (one at a time) via the UART port on an Arduino-like board (the Arduino Shield for Raspberry Pi). We will also turn on an LED if communication is successful.

Required Hardware Components

- **Arduino Shield for Raspberry Pi** (with Arduino Leonardo chip onboard)
- **Grove Base Shield**
- **Grove-LCD RGB Backlight V4.0**
- **Grove LED Socket Kit v1.5 + LED**

- **Grove - NFC Tag v1.1**
- **Grove - 125 KHz RFID Reader**
- **NFC Tags**

A. Setting Up the Arduino Shield & Testing the LED

1. **Identify the Arduino Shield.**
   This shield includes an Arduino Leonardo chip, so we can treat it as an Arduino in the IDE.
2. **Attach Grove Base Shield & Modules.**
   - Make sure the power switch on the base shield is set to **5V**. - The LED's long leg is positive; attach it correctly to the **LED Socket Kit**.
3. **Connect the Following to the Shield:**
   - **LCD** → I2C port
   - **LED Kit** → D4 port
   - **NFC** → UART port (later replaced by RFID)
4. **Program an Empty Sketch to Verify.**
   Plug your shield into your PC via USB, open the Arduino IDE, select **Board = Arduino Leonardo** and the correct **Port**. Upload a blank sketch to ensure everything is recognized.
5. **Test the LED Kit.**
   Load this simple blink code:

```
#define LED 5   // D5 on Grove Shield
void setup() {
  pinMode(LED, OUTPUT);
}
void loop() {
  digitalWrite(LED, HIGH); delay(500);
  digitalWrite(LED, LOW);  delay(500);
}
```

6. **Check the LED Blink**
   If the LED does not blink, verify its polarity and ensure the kit is firmly connected. You can also find the code in the GitLab repository.

B. Setting Up and Testing the LCD

7. **Install the LCD Library**
   The LCD module requires a dedicated library to handle I^2C communication and basic commands (e.g., set cursor, change color).
   - Go to the GitLab repository and locate the ZIP file for the LCD library.
   - In the Arduino IDE, click **Sketch** → **Include Library** → **Add .ZIP Library...** and select the downloaded file.
   - If the library installs successfully, you should see it listed under **Contributed libraries**.

   > **— Why do this?.** Most Grove LCDs—especially "RGB Backlight" types—rely on a custom library that handles the display initialization, text printing, color adjustments, and special characters. Installing this library ensures the example sketches and your own code will compile without missing references.

8. **Run a Test Sketch.**
   Once the library is installed, verify that the LCD works properly by using the provided test

code:
- Download `labTest.ino` from the same GitLab repository.
- Open it in the Arduino IDE and **upload** it to your shield.
- The LCD should cycle through **green, red, blue** backlight colors and act as a simple timer.
- If you see the color cycling and the timer updating on the LCD, it indicates that:
  - The LCD is receiving power and I^{2}C signals correctly.
  - The library functions (e.g., `lcd.begin()`, `lcd.setRGB()`) are executing successfully.

Tips:
- If the LCD stays blank or shows random characters:
  - Double-check the I^{2}C cable connection (e.g., `I2C-1` or `I2C-2`) on your Base Shield.
  - Confirm the library matches your LCD version (some have slightly different addresses or initialization sequences).
  - Run `sudo i2cdetect -y 1` on the Raspberry Pi if using an Arduino Shield for RPi (or equivalent) to ensure the device address is found. On a standalone Arduino, just ensure the hardware wiring matches your code pin references.
- The code can be easily modified to print your own text or update the screen with sensor readings.

C. Setting Up the Grove NFC Module

9. **Download NFC Library.**
   Obtain the file "`Seeed_Arduino_NFC-master.zip`" from the GitLab repository. In the Arduino IDE, navigate to:
   - **Sketch** → **Include Library** → **Add .ZIP Library...**
   - Select the downloaded ZIP file to install it.

   This NFC library supports the Grove NFC module's PN532 chipset, providing higher-level functions to detect, read, and write NFC tags.

Tips:
- If you see compilation errors like "`No such file or directory: PN532_HSU.h`," recheck that the library installed successfully.
- Some versions of the NFC library may conflict with older PN532 libraries; ensure only one relevant NFC library is active.

10. **Load** `nfc.ino`
    You can find a reference sketch in the same GitLab repository named `nfc.ino`. To use it:
    - Download the file or open it directly in Arduino IDE using **File** → **Open**.
    - Alternatively, copy and paste the code from the GitLab page into a new Arduino IDE sketch.

11. **Examine the Code.**
    The `nfc.ino` sketch typically includes the following:

```
#include <NfcAdapter.h>
#include <PN532/PN532_HSU/PN532_HSU.h>
#include <rgb_lcd.h>
```

```

// NFC/PN532 config
PN532_HSU pn532hsu(Serial1);
NfcAdapter nfc(pn532hsu);

// LCD + LED
rgb_lcd lcd;
#define LED 4

```

Key points:

- `Serial1` is used for NFC communication, because on an Arduino Leonardo (or similar), `Serial1` refers to the hardware UART pins.
- `nfc.tagPresent()` checks if an NFC tag is in range. If found, the code sets the LED high, writes "Hello World" to the tag, and reads it back.
- The code also prints the tag's UID to both the **Serial Monitor** and the **LCD** screen.

> — **Why is `Serial1` used for NFC?.** On the Leonardo-based shield, `Serial1` is a dedicated hardware UART. The PN532 NFC module (in HSU or high-speed UART mode) communicates via these pins. Meanwhile, `Serial` remains free for USB debugging in the Serial Monitor.

12. **Upload and Test.**
1. Plug an NFC tag onto or near the Grove NFC module antenna (see Figure 11.1). 2. Compile and **upload** the `nfc.ino` sketch to the Arduino shield. 3. Open the Serial Monitor at the correct baud (e.g., 9600 or 115200). 4. Wave or place the tag in front of the module:
    - If detected, the Serial Monitor should print "`NFC TAG FOUND`," then show "`Success.`"
    - The LED (attached to pin #4) lights up, and the code writes "Hello World" to the tag's memory, then reads it back.

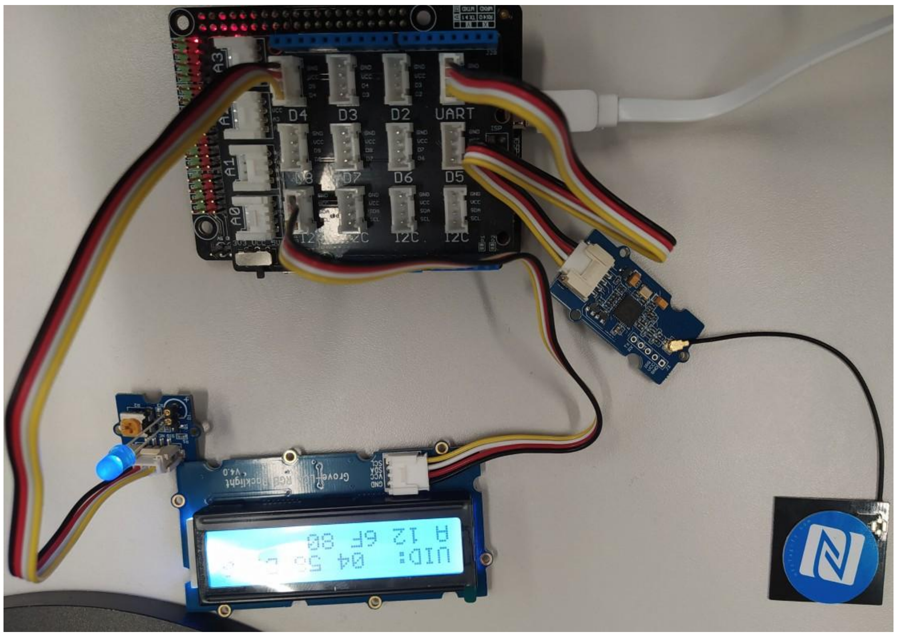
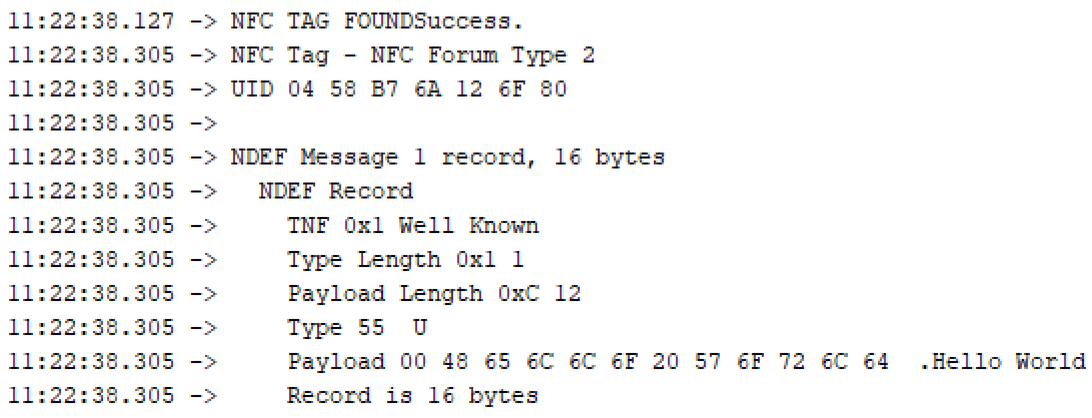


Figure 11.1: The NFC Setup

13. **Observe Serial Output.**
In the Arduino Serial Monitor, you might see:

```
NFC TAG FOUND
Success.
Write success.
Tag type: MIFARE 1k
UID: 04 A2 0B FF 1C 57 85
Reading data...
Payload: Hello World

```

The **Payload** line reveals the message ("Hello World") written to the tag, while **UID** indicates the unique identifier of the NFC tag (see Figure 11.2 for an example).



```
11:22:38.127 -> NFC TAG FOUNDSuccess.
11:22:38.305 -> NFC Tag - NFC Forum Type 2
11:22:38.305 -> UID 04 58 B7 6A 12 6F 80
11:22:38.305 ->
11:22:38.305 -> NDEF Message 1 record, 16 bytes
11:22:38.305 ->   NDEF Record
11:22:38.305 ->     TNF 0x1 Well Known
11:22:38.305 ->     Type Length 0x1 1
11:22:38.305 ->     Payload Length 0xC 12
11:22:38.305 ->     Type 55  U
11:22:38.305 ->     Payload 00 48 65 6C 6C 6F 20 57 6F 72 6C 64  .Hello World
11:22:38.305 ->     Record is 16 bytes
```

Figure 11.2: NFC Tag Output in Serial Monitor

Tips:
- The **UID** is unique to each NFC tag. Use it for identification or security checks.
- **Tag Format**: MIFARE or NTAG types may appear. Some tags allow larger payloads, while others may store only minimal data (like URLs or small text).
- **Reading distance** can be short (2–3 cm). Ensure the tag is placed over the antenna coil area on the NFC module.

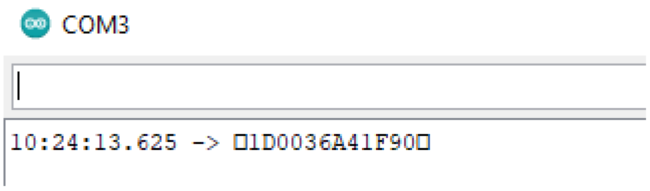
D. Setting Up the RFID Reader

14. **Swap out the NFC Module for Grove - RFID Reader.**
Unplug the NFC module from the UART port and connect the **Grove - 125 KHz RFID Reader** in its place. This module is **read-only** for 125 KHz RFID tags.

Tips:
- The Grove - RFID reader typically operates on the same UART pins as the NFC module, so you can reuse the same `Serial1` logic in your Arduino code—just be sure to load the RFID-specific sketch next.
- Note that the read range is typically a few centimeters (2–3cm).

15. **Obtain and Upload the RFID Reader Code**
Visit the GitLab repository to find the RFID reader example code. Download or copy it, then:
    - Open the Arduino IDE and paste the code into a new sketch.
    - Click **Verify** to compile and check for errors.
    - Click **Upload** to flash the code to your Arduino shield.

16. **Open Serial Monitor.**
    Once the code is running:
    - Hold an RFID tag or card **within 2.5 cm** of the reader.
    - The unique tag ID should print out in the Arduino Serial Monitor (see Figure 11.3).



```
10:24:13.625 -> □1D0036A41F90□
```

Figure 11.3: Reading an RFID Tag

17. **Try Different Tags.**
    Gather several 125 KHz RFID tokens or cards and test each one:
    - Observe that each card outputs a distinct alphanumeric ID in the Serial Monitor.
    - This is how most RFID-based access systems differentiate tokens—by their unique IDs.
    - If a card is not recognized or the range is too short, ensure the card is near the antenna coil on the Grove RFID reader.

— Why do this?. Verifying multiple tags shows that your reader reliably detects and distinguishes them by unique IDs. Real-world systems typically cross-reference these IDs in a database to grant or deny access, log activity, or track item movements.

Measuring Success

- **LED + LCD Check**:
  - The LED Kit blinks or remains steadily lit when you run your blink sketch.
  - The LCD correctly cycles colours (*green, red, blue*) or displays the time/other text in `labTest.ino`.
  - If the LCD remains blank or shows random characters, double-check the I²C cable and library installation.
- **NFC Detection/Write**:
  - When you place an NFC tag near the module, the Serial Monitor prints "`NFC TAG FOUND`," and the LED lights up.
  - A "`Success.`" message indicates the code wrote "Hello World" to the tag, and the UID is displayed on the LCD.
  - Re-scanning the tag in a phone app or another reader would reveal "Hello World" stored on that tag.
- **RFID Reads**:
  - With the 125kHz RFID module attached to the UART port, each card or fob placed within 2cm returns a unique ID in the Serial Monitor.
  - No errors appear, confirming that the correct library/code is running and the reader is functioning properly.
  - Trying multiple cards produces different IDs, demonstrating reliable detection and distinction among tags.

■

Optional Scenario: "Inventory Tracking"
Envision a simplified warehouse or stockroom where both NFC and RFID technologies aid in item management:

- **NFC Tags**:

  - Each high-value or special-handling item might include an NFC tag storing details (e.g., "Item #1234," date of manufacture, or owner info).
  - Your Arduino-based reader can *write* updated info (like "Checked Out on Feb 15, 2025") whenever items move.
  - If you have a phone or another NFC-capable device, you can *tap* the item's tag to instantly see data (e.g., "Fragile! Handle with care.").
- **RFID Cards (125 kHz)**:
  - Faster scanning is possible as employees or items pass through a doorway or gate.
  - The Arduino recognizes each card's unique ID without needing to align the tag precisely.
  - This is convenient for rapid "in/out" logging, reducing manual labor.
- **LED or LCD Feedback**:
  - **LED Indicator**: Blink or change color upon a successful read/write. In a noisy warehouse, a flashing LED can quickly signal "Item recognized" or "Scan failed."
  - **LCD Display**: After detecting a tag, display the item's ID, name, or any stored note (e.g., "Location: Aisle 3, Shelf B").
  - Optionally combine an alarm or buzzer for items that shouldn't leave the premises.

— Implementation Notes.
- **Write/Read Routines**: For NFC tags, use the `nfc.ino` approach. 125 kHz RFID is read-only, so you rely on the unique ID for tracking.
- **Database or Logging**: Expand beyond the Arduino by sending scanned IDs to a Raspberry Pi or cloud service, maintaining real-time stock levels or history logs.
- **User Prompt**: If the item is marked "out of stock" or "restricted," the LCD can display a warning, and the LED might flash red.
- **Automation Opportunities**: As employees pass items through an RFID gate, the Arduino automatically increments a "checked-out" counter. If the item's NFC tag indicates a due date, scanning can remind staff to return it on time.

This scenario shows how integrating NFC (for detailed record storage) and RFID (for quick ID scanning) can streamline an *inventory tracking system*, with immediate visual cues from LEDs or LCD for user feedback. It bridges local hardware (Arduino) with potential cloud or database solutions to provide real-time item monitoring.

Further Reading
- **NFC Forum**:
  `https://nfc-forum.org/`
- **RFID Overview**:
  `https://en.wikipedia.org/wiki/Radio-frequency_identification`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the concepts behind Radio-Frequency Identification (RFID) and Near-Field Communication (NFC), including how they encode data in tags/cards, the typical hardware setup (reader vs. tags), and practical applications like access control, inventory tracking, or simple data transfers. Understanding these principles will help you build robust projects that read, write, and process information using wireless tags.

A. RFID Basics

RFID (Radio-Frequency Identification) uses electromagnetic fields to identify and track tags attached to objects. Common frequencies and types include:

- **125 kHz (LF):** Low-frequency RFID, often read-only, short range (a few centimeters).
- **13.56 MHz (HF):** Overlaps with NFC technology (ISO 14443), allowing short-range, often read-write capability.
- **UHF (860–960 MHz):** Longer read ranges (meters), used in warehouses for bulk scanning.

A.1 How RFID Works

- **Reader (Interrogator)**: Emits a radio signal, powering or activating the tag.
- **Tag (Transponder)**: Contains a small IC and antenna. Responds by sending back a unique ID or stored data.
- **Passive vs. Active**: Passive tags are powered by the reader's field (no battery). Active tags have their own battery for higher range.

A.1.1 Example Use Cases

- **Access Control**: Swipe an RFID card to unlock a door.
- **Asset Tracking**: Tag items in a warehouse or library for quick scanning.
- **Livestock/Pets**: Implantable RFID chips help identify animals.

B. NFC Basics

NFC (Near-Field Communication) is a specialized subset of HF RFID operating at 13.56 MHz:

- **Very Short Range**: Typically up to 4 cm.
- **Peer-to-Peer Mode**: Two NFC devices (e.g., smartphones) can exchange data directly.
- **Card Emulation Mode**: The device can act like a payment or transit card.
- **Read/Write Mode**: Reads data from NFC tags or writes to them (e.g., storing small amounts of text or URLs).

B.1 Key Characteristics

- **ISO Standards (14443, 18092)**: Define communication protocols for NFC operations.
- **NDEF (NFC Data Exchange Format)**: A container format for storing text, URIs, and other data on NFC tags.
- **Smartphone Integration**: Most modern phones have NFC capabilities for tap-and-pay or quick data exchange.

B.1.1 Common NFC Applications

- **Contactless Payments (e.g., Apple Pay, Google Pay)**.
- **Smart Posters**: Tap phone on a poster's embedded NFC tag to open a website or app.
- **Pairing Devices**: Quickly set up Bluetooth or Wi-Fi connections.

C. Differences and Overlaps: RFID vs. NFC

While NFC is a type of HF RFID, there are practical distinctions:

- **Data Capacity**: Basic LF RFID tags (125 kHz) often hold just a fixed ID, while NFC tags can store text, URLs, or small data blocks.

- **Range**: NFC is limited to a few centimeters to ensure deliberate user interaction (tap), whereas other RFID systems can scan from farther away.
- **Interaction Model**: NFC typically supports more interactive read-write operations (especially for smartphone usage). Traditional RFID often focuses on simple ID reading at a distance.

C.1 Hardware Differences

- **NFC Modules** (13.56 MHz): Usually read-write, can handle NDEF data. Examples: PN532-based modules.
- **LF RFID Modules** (125 kHz): Often read-only (like the Grove 125 kHz module). Scans an ID from the tag and outputs it via serial.

D. Grove Modules and UART Configuration

Grove-based RFID/NFC modules typically communicate over:

- **UART Serial**: TX/RX lines feed into your microcontroller's serial pins (e.g., `Serial1` on an Arduino Leonardo).
- **I2C or SPI (Less Common)**: Some NFC modules (like PN532) can also operate in I2C/SPI mode. The library determines which interface is used.

D.1 AT Commands vs. Libraries

- **RFID Reader (125 kHz)**: Often does not rely on AT commands. It automatically outputs the tag ID when a card is near.
- **NFC Modules (PN532)**: Typically rely on an existing library (e.g., PN532 library) rather than raw AT commands, though some older modules use AT command sets for configuration.

E. Data Exchange Flow

- **Tag Sensing**: The reader powers or polls the tag within its field.
- **Identification**: The tag sends back its unique ID or additional data (e.g., text).
- **Microcontroller Processing**: The Arduino code parses the ID/data. In your lab, you might:
  - Display the tag ID on an LCD.
  - Light an LED upon a successful read.
  - Print data in the Serial Monitor for logging.

E.1 Writing to NFC Tags

- **NDEF Messages**: You can store short text (like "Hello World") or a URL in the tag. The library typically handles encoding into NDEF format.
- **Practical Limits**: NFC tags might hold 48 bytes to a few kilobytes, enough for basic text or short data.
- **Security**: Some tags support write protection or password-locking after data is written.

F. Real-World Applications and Use Cases

- **Door Access Systems**: RFID cards at 125 kHz are widely used for building entry; the door lock controller checks the tag ID.

- **Inventory/Asset Management**: Use RFID or NFC to track items. An LCD can confirm successful scans or show item details.
- **Personalization/IoT Integration**: Place an NFC tag near a device to configure Wi-Fi credentials or trigger a device action.
- **Payment/Loyalty Cards**: NFC cards store payment tokens or membership data; a smartphone or specialized terminal reads them.

G. Common Pitfalls and Best Practices

- **Range and Orientation**: Ensure tags are oriented correctly and within the active field. Some modules require very close contact.
- **Power Supply Stability**: RFID/NFC readers can draw surges of current when activating. A stable 5V supply is crucial.
- **Interference**: Metal surfaces or other RF sources can degrade reading performance.
- **Data Handling**: Ensure your code handles partial scans or unexpected data gracefully (e.g., empty or corrupted tag data).

G.1 Upgrading Security and Robustness

- **Encrypted Tags**: Some higher-level NFC tags support encryption or password-protected sectors (like MIFARE DESFire).
- **Failure Recovery**: If a tag read fails, prompt the user to re-scan.

H. Next Steps and Advanced Topics

- **Multi-Tag Systems**: Reading multiple tags in rapid succession or from different frequencies (LF + HF).
- **NFC P2P Mode**: Two NFC devices (e.g., phone and Arduino) exchanging data without a “tag” in between.
- **Higher-Level Protocols**: Integrating the data with a cloud or local database for advanced tracking, analytics, or automation.

Final Note.

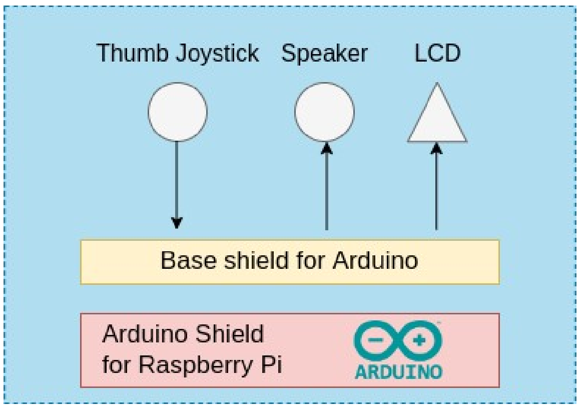
By experimenting with both **RFID (125 kHz)** and **NFC (13.56 MHz)** modules, you explore two distinct yet related wireless technologies. RFID is often used for simple identification at moderate distances, while NFC facilitates short-range, interactive read/write operations (popular with smartphones). Mastering these fundamentals empowers you to build robust systems—access controls, inventory trackers, or interactive IoT solutions—that leverage the convenience of wireless tag scanning and data exchange.

12. Multimedia Communication

Objectives

- Learn how to display data on an LCD
- Learn how to program and control a joystick
- Learn how to generate sound on a speaker
- Learn how to display (or react to) received data

Lab Plan



In this lab, we will explore how to use a **thumb joystick** and a **speaker**, as well as how to control an **LCD**'s color using joystick inputs. The joystick acts like a game controller, producing coordinate outputs you can read in your Arduino code, while the speaker can generate musical notes or beeps.

Required Hardware Components

- **Arduino Shield for Raspberry Pi** (Arduino Leonardo chip)

- **Grove Base Shield**
- **Grove - LCD RGB Backlight**
- **Grove - Thumb Joystick v1.1**
- **Grove - Speaker v1.1**

A. Setting Up the Arduino Shield

1. **Attach the Base Shield & Modules.**
   First, carefully align the Grove Base Shield on top of the Arduino Shield for Raspberry Pi so that all pins match correctly. Then, plug in your Grove modules:
   - **LCD → I2C port**
     The LCD uses I^2C communication for text and RGB backlight control.
   - **Thumb Joystick → A0 port**
     The joystick typically provides two analog axes (X & Y). The kit outputs analog voltages read by `A0` (and possibly `A1`, depending on the joystick variant).
   - **Speaker → D4 port**
     The speaker can generate tones and beeps when driven by PWM-capable digital pins.

   Ensure the voltage switch on the base shield is set to **5V**, which is standard for these modules. If you accidentally leave it at 3.3V, the LCD or speaker may not function as expected.

   — Why 5V?. Many Grove modules (especially displays, speakers, and certain sensors) are designed for 5V logic. Keeping the shield at 5V ensures full brightness, volume, or accurate readings. Always double-check the required voltage in each module's documentation.

2. **Connect to PC & Verify Board.**
   Use a USB cable to connect the shield to your computer. In the Arduino IDE:
   - Go to **Tools** → **Board** and select *Arduino Leonardo*.
   - Under **Tools** → **Port**, pick the serial port that disappears/reappears when plugging/unplugging your shield.

   If you are unsure which port to choose, try disconnecting and reconnecting the board to see which port number changes in the IDE's list.

   Tips:
   - If no port is listed, install or update the Arduino drivers for Leonardo boards.
   - Make sure you are using a data-capable USB cable (some cables are power-only).

3. **Install Required Libraries.**
   If you have not already done so, you will need the Grove LCD RGB library:
   - Grove LCD RGB library from GitHub or the GitLab repository.
   - In Arduino IDE, click **Sketch** → **Include Library** → **Add .ZIP Library...**, then select the downloaded .ZIP file.
   - Confirm it appears under **Contributed libraries**.

   This library allows you to set text, cursor position, and backlight color on the LCD via I^2C.

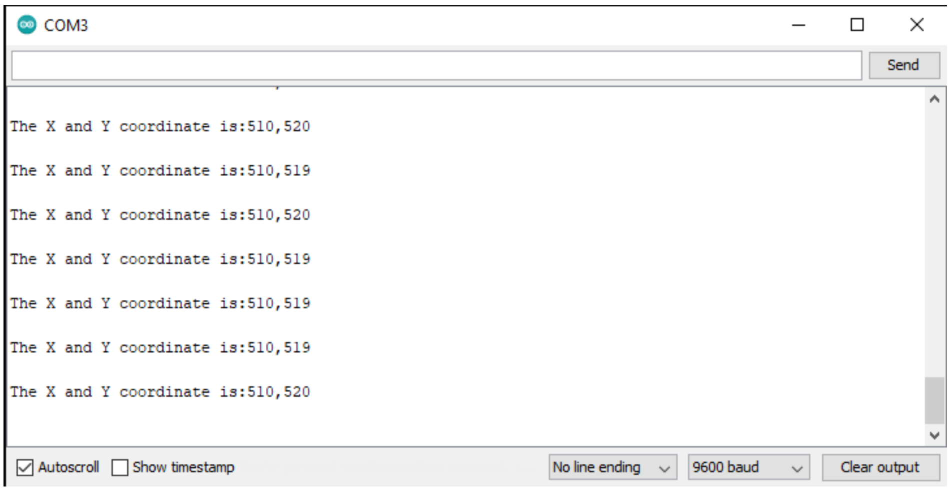
   — Why install libraries?. Each Grove module (LCD, sensor, etc.) typically needs a dedicated library to provide higher-level commands. Without these libraries, you'd have to manually handle I^2C or timing logic. Prewritten libraries simplify your code and reduce bugs.

4. **Upload an Empty Sketch.**
   To verify everything is configured properly:
   - Open the IDE, go to **File** → **New** to create a blank sketch.
   - Upload it (the default `setup()` and `loop()` do nothing).
   - Wait for "Done uploading" in the IDE status bar, confirming no driver or cable issues.

   If you see "Done uploading" and no error messages, you can proceed to write code for the LCD, joystick, or speaker.

B. Programming the Thumb Joystick

5. **Obtain the Joystick Code.**
   From GitLab, copy the content into a new Arduino sketch.
6. **Upload & Observe Coordinates.**
   Open Serial Monitor and move the joystick around. You should see **X** and **Y** values changing (Figure 12.1).



```
The X and Y coordinate is:510,520
The X and Y coordinate is:510,519
The X and Y coordinate is:510,520
The X and Y coordinate is:510,519
The X and Y coordinate is:510,519
The X and Y coordinate is:510,519
The X and Y coordinate is:510,520
```

Figure 12.1: Joystick Serial Output Example

C. Programming the Speaker

7. **Download the Speaker Code.**
   In the GitLab repository, find the example code for the speaker (often named something like `speaker.ino`). Copy it into a new Arduino IDE sketch.

   Tips:
   - Confirm you have the speaker module plugged into **D4** (or whichever pin is referenced in the code).
   - If the code uses `tone()` or `analogWrite()`, make sure the pin supports PWM or tone generation on your particular board.

8. **Upload & Test.**
   1. **Verify** the sketch, then click the **Upload** button. 2. Once uploaded, the speaker should produce tones or a short melody. 3. In the code, you might see an array like:

```
int BassTab[] = {1911, 1702, 1516, 1431, 1275, 1136, 1012};

```

which maps frequency or timing values for different musical notes (e.g., a bass scale). You can tweak these numbers to experiment with different pitches.

> **— Why adjust the array?.** Changing the array values modifies the frequency (and thus the pitch) of the notes. By increasing or decreasing each entry, you can create custom melodies or sound effects—useful for alarms, feedback tones, or simple music.

D. Combining Joystick, Speaker, and LCD

9. **Create a Combined Sketch.**
   From GitLab, copy the code into Arduino IDE. Don't upload it yet if you want to avoid immediate noise.
10. **Code Overview.**
    The sketch initializes:

```
int colorR = 123, colorG = 123, colorB = 123;
int bass = 1000;
```

    and reads joystick **X** and **Y** values. If the joystick is moved left/right, the code changes ‘`bass`‘ up/down. If moved up/down, it changes ‘`colorR, colorG, colorB`‘ for the LCD backlight.
11. **Idle Positions.**
    Joysticks might read `X=509`, `Y=510` at idle. The code checks if they're near these defaults to prevent random triggers.
12. **Upload & Watch the Effects.**
    - The **speaker** changes pitch (‘`bass`‘) based on joystick left/right. - The **LCD** color changes (‘`colorR/G/B`‘) when you move the joystick up/down. - The code prints values in the Serial Monitor, so open it to observe them in real time.

Measuring Success

- **Joystick Output**:
  - The Serial Monitor displays clear X/Y coordinate changes when you move the thumb stick.
  - Joystick values hover around the default midpoint (e.g., `509/510`) at rest, and shift noticeably when tilted in any direction.
- **Speaker Tones**:
  - Adjusting the `bass` variable via joystick left/right produces distinct pitch changes.
  - The speaker emits audible tones that vary in frequency, confirming real-time interaction between the joystick and speaker.
- **LCD Color Changes**:
  - Moving the joystick up or down updates `(colorR, colorG, colorB)`, altering the backlight hue on the LCD.
  - The transitions happen smoothly, with color changes reflecting in real time.

■

Use Case Scenario (Optional): "Sound + Color Synchronization"
Envision an interactive mini "synthesizer" or "visual instrument" where:

- **Joystick X-axis** increases or decreases the speaker's `bass` value, producing higher or lower pitches.

- **LCD Backlight** grows darker or more intense (e.g., from a light pastel to a deep hue) as the pitch climbs.
- **LCD Display** is only lit while you're actively playing a note (i.e., the speaker is sounding). When the sound level or pitch returns to idle, the LCD's backlight turns off to save power or highlight silence.

— **Implementation Hints.**

1. **Mapping Pitch to Color:**
   - Suppose your `bass` variable ranges from 500 (low pitch) to 2000 (high pitch).
   - You could map `bass` to a color intensity range (e.g., `0-255` for `colorR`, `colorG`, `colorB`).
   - Use a linear mapping function, like:

     ```
     colorValue = map(bass, 500, 2000, 0, 255);
     lcd.setRGB(colorValue, 0, 255 - colorValue);
     ```

     so at lower pitch, you get a teal shade; at higher pitch, it transitions toward magenta (or any color blend you prefer).
2. **Tying LCD Brightness to Active Sound:**
   - Whenever you *actually* produce a tone (via `tone()` or `analogWrite()` on the speaker pin), call `lcd.setRGB(r, g, b)` to illuminate the backlight.
   - Once you stop generating sound (e.g., `noTone()`), set the LCD backlight to `(0, 0, 0)` or simply call a function like `lcd.noBacklight()` if your library supports it.
3. **Joystick Controls Both:**
   - **Left/Right (X-axis):** Adjust `bass` for pitch changes.
   - **Up/Down (Y-axis):** Optionally influence brightness or a secondary color channel, so you can generate richer color variations beyond a single dimension of hue.
4. **Additional Ideas:**
   - Display the current pitch (or color intensity values) on the LCD text line, so you know precisely where you are in the range.
   - Fade the LCD color gradually rather than instantly, for smoother transitions (e.g., using a small `delay` or incremental steps).
   - Integrate button logic: a button press might trigger a special color "flash" or sound effect.

This setup turns your Arduino-based kit into a rudimentary "audio-visual" system, where **sound** and **color** blend in real time, driven by **joystick** motion. Experiment with different mappings and transitions to create unique user experiences—a mini "sound + color" art piece or an interactive learning tool for pitch and color relationships.

Theory Deep Dive: Underlying Principles and Concepts

This section explores the underlying concepts behind reading a thumb joystick (analog input), driving a speaker (audio output), and controlling a color LCD backlight. By understanding how these interfaces work, you can build more intuitive multimedia experiences in embedded systems—ranging from simple interactive menus to advanced, game-like user interfaces.

A. Joystick Fundamentals

A **thumb joystick** typically provides two independent analog axes (**X** and **Y**) each mapped to a potentiometer:

- **Potentiometers:** The joystick's motion changes the resistance on each axis, producing a variable voltage from 0–5 V (or 0–3.3 V on some boards).
- **Analog-to-Digital Conversion (ADC):** The microcontroller's ADC (e.g., `analogRead(A0)`) converts this voltage to a digital value (often 0–1023 on a 10-bit ADC).
- **Neutral/Idle Position:** Typically, the joystick center hovers around half the ADC range (e.g., 512). Slight variations are normal.
- **Push-Down Button (Optional):** Some joysticks have a built-in switch when you press down on the stick, read as a separate digital input.

A.1 Mapping Joystick Values

- **Thresholds:** Code often checks if the X/Y readings deviate significantly from the center (e.g., ± 50) before reacting.
- **Directional Control:** Left/Right might correspond to negative/positive changes in `X`, up/down to negative/positive changes in `Y`.
- **Scaling:** `map(value, 0, 1023, 0, 255)` can convert full-range ADC reads into an 8-bit scale for color or speaker frequency.

A.1.1 Use Cases

- **UI Navigation:** Move a cursor on an LCD menu.
- **Game-like Interaction:** Control a servo or sprite's position based on joystick movements.
- **Parameter Tuning:** Adjust volume, pitch, or brightness by tilting the joystick.

B. Speaker and Sound Generation

A **speaker**, particularly a small piezo or electromagnetic buzzer, converts electrical signals into audible vibrations:

- **Digital Tones:** A microcontroller toggles a pin at a certain frequency to generate square waves heard as tones (beeps).
- **PWM (Pulse Width Modulation):** Sometimes used to create different pitches/volumes. For basic buzzers, just toggling `HIGH/LOW` at a set frequency suffices.
- **Tone Libraries**: Arduino offers `tone()` or similar functions to simplify playing notes by specifying frequency and duration.

B.1 Musical Notes and Frequencies

- **Pitch Tables**: A4 is typically 440 Hz, C4 is 261.63 Hz, etc. Code may store these frequencies in arrays (e.g., `int noteFreq[] = {262, 294, 330, ...};`).
- **Duration Control**: `delay()` or timers define how long a note plays. Sequencing multiple notes yields a simple melody.
- **Volume**: Basic buzzers have limited volume control, typically on/off. More advanced modules or amplifiers can vary loudness via PWM amplitude.

B.1.1 Joystick-Driven Pitch

- **Real-Time Frequency Adjustments**: For dynamic sound effects, read joystick X/Y, map values to 100 Hz–2 kHz, then update tone output.
- **Creative Feedback**: Changing pitch can signal different states or intensities (e.g., a game object's "energy" level).

C. LCD Control and RGB Backlight

A **Grove - LCD RGB Backlight** module typically features:

- **16x2 or 20x4 Character Display**: Standard characters or custom symbols.
- **I2C Interface**: The microcontroller sends commands over SDA/SCL lines, freeing multiple digital pins for other uses.
- **RGB Backlight**: Individual red, green, and blue LEDs behind the LCD can be mixed to produce a wide color range (`setRGB(r,g,b)`).

C.1 Adjusting Backlight Color

- **Setting Each Channel**: R, G, B typically range 0–255. For example, `(255, 0, 0)` is bright red, `(0, 255, 0)` green, etc.
- **Smooth Transitions**: By incrementing color channels gradually, you can fade from one color to another.
- **Mapping Joystick Values**: `colorR = map(xVal, 0, 1023, 0, 255);` transforms joystick input into a red intensity.

C.1.1 Displaying Text

- `setCursor(col, row)`: Position the text cursor before printing messages.
- **Scrolling / Clearing**: Some libraries let you scroll text or clear the display with a single function call.

D. Integrating Inputs and Outputs for Interaction

When combining joystick, speaker, and LCD, consider:

- **Event Loop**: Poll the joystick each loop, update speaker frequency if *xVal* changes, and update LCD color if *yVal* changes.
- **Debouncing/Thresholds**: Joysticks can jitter slightly near center. Add small "dead zones" to avoid constantly changing pitch or color.
- **User Feedback**: Printing current `(bass)` or `(colorR, colorG, colorB)` in **Serial Monitor** or on the LCD itself helps users see real-time status.

D.1 Combining Joystick Directions

- **Left/Right for Sound**: Map `xVal` to frequency or beep patterns.
- **Up/Down for Color**: Map `yVal` to `R, G, B` values on the LCD backlight.
- **Click (if available)**: Some joysticks have a center push button. That could trigger a reset or toggle mode.

E. Potential Extensions and Real-World Examples

- **Mini Synthesizer**: Use the joystick to select notes or chords, the speaker to play them, and the LCD to show note names.
- **Gaming Interface**: Create a basic interface for a Pong or Snake game using the joystick for control, speaker for sound effects, and LCD for score or statuses.
- **Color-Themed Alarms**: Map environment data (e.g., temperature) to color and pitch. Hotter = redder and higher pitch.

E.1 Additional Devices

- **LED Bars or Neopixel Strips**: Expand your color control beyond the LCD backlight.

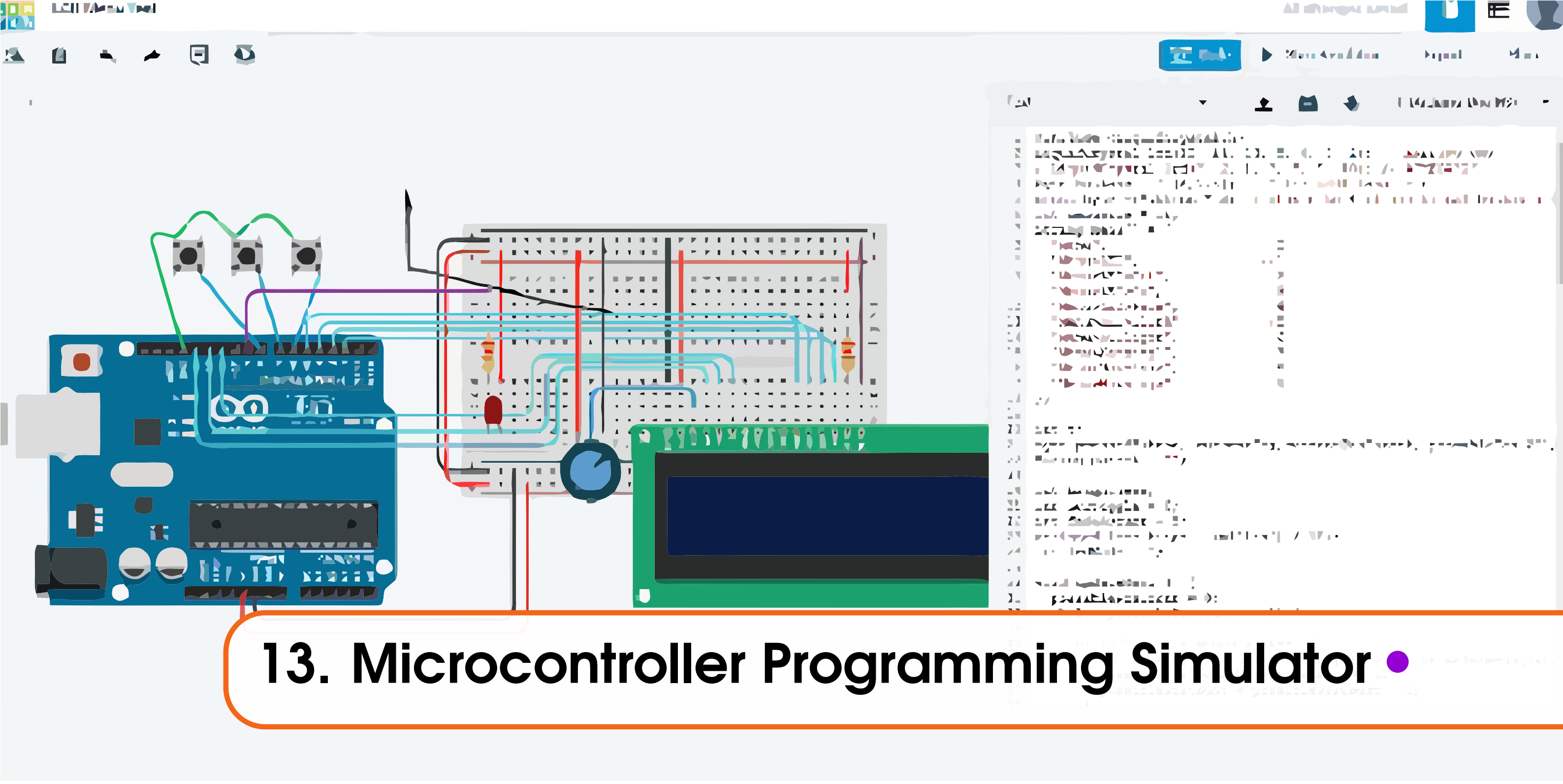
- **Pressure or Touch Sensors**: Combine multiple inputs to create richer interactive experiences.

F. Best Practices and Common Pitfalls

- **Power Management**: Speakers can draw noticeable current when generating louder or higher-frequency sounds. Ensure your power supply is stable.
- **Noise/Distortion**: Piezo speakers at high frequencies can sound harsh. Adjust duty cycle or use a small amplifier for better audio quality.
- **User Calibration**: Some joysticks might read slightly off-center. Consider calibration or allow the user to set a neutral point.
- **Resource Conflicts**: Ensure the pins used for joystick analog reads and speaker PWM do not clash with other libraries or hardware.

G. Conclusion and Future Directions

By integrating a **joystick** for analog input, a **speaker** for audio output, and an **LCD** for visual feedback, you can build interactive, multimedia-rich projects. Experimenting with different pitch mappings, color schemes, and joystick interactions can inspire creative designs—from simple musical demos to mini-games or interactive dashboards. As you expand your skillset, consider adding sensors like accelerometers or ultrasonic rangers, controlling LED arrays, or even hooking the system to network services for remote control or data logging. The core principles of **sensor reading**, **signal generation**, and **visual feedback** remain the cornerstone of engaging IoT and embedded applications.



13. Microcontroller Programming Simulator

Objectives

- Learn how to use a simulator (Tinkercad Circuits)
- Learn how to design a basic circuit
- Get familiar with the Tinkercad circuits interface

Required Hardware Components

- **PC with internet connection**

A. Setting Up Tinkercad Circuits

1. **Go to** `www.tinkercad.com`**.**
   Navigate to Tinkercad in your browser, then click on **Start Tinkering**. If you're not already logged in, you'll be prompted to **log in** or **create an account**. Tinkercad uses Autodesk accounts, so you can sign in with Google, Apple, or standard credentials.

   > **— Why Tinkercad?.** Tinkercad offers a free, web-based platform to simulate basic electronic circuits and Arduino projects. You can prototype your connections, run code, and visualize component interactions—all without physical hardware.

2. **Open the Circuits Dashboard.**
   After signing in:
   - Look for the left-hand menu (sometimes under a hamburger icon).
   - Select **Circuits** to view or create virtual electronics projects.
   - Click the **Create new Circuit** button (Figure 13.1) to begin a fresh simulation workspace.
3. **Select "All" Components.**
   In the panel on the right side, switch from "Basic" to "All" to reveal the full range of Tinkercad's components (Figure 13.2). This gives you access to Arduino boards, sensors, LEDs, resistors, and more.

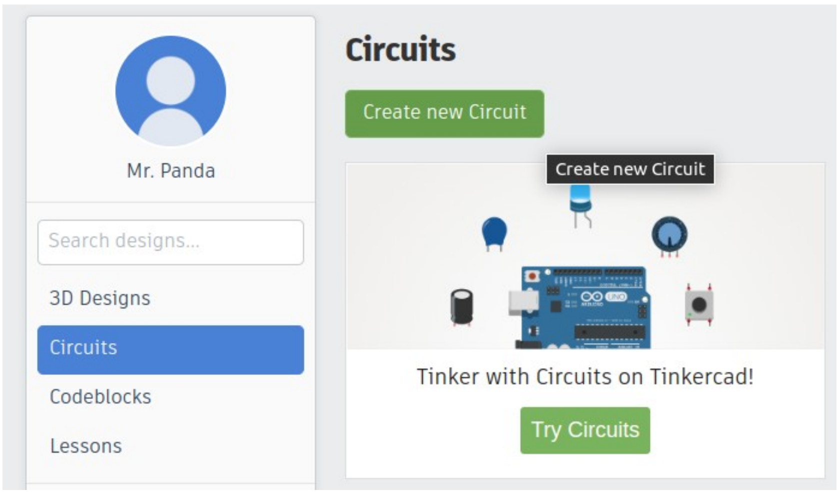


Figure 13.1: Create New Circuit

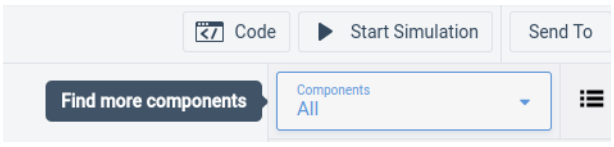


Figure 13.2: Component Selection

Tips:

- If you don't see the component you need, ensure you're in "**All**" mode rather than "**Basic**" or "**Starters**."
- You can search for a specific item (e.g., "Arduino," "Servo," or "LED") using the magnifying glass icon in Tinkercad's component panel.
- Rename your circuit from the default "Untitled" to something more descriptive (e.g., "MyFirstArduinoCircuit") by clicking the name in the top-left corner.

B. Creating an Initial Circuit

4. **Drag Arduino Uno, Breadboard, and an LED.**
   From the **All** components panel:
   - Place an **Arduino Uno** board onto the Tinkercad workplane.
   - Add a **small breadboard**.
   - Drag an **LED** and drop it near the breadboard (see Figure 13.3).

   Connect them so that the LED is in line with one of the Arduino's digital pins and ground. Use **red** wires for positive connections, **black** for negative (ground).

Tips:

- If you just place the LED directly between a pin and GND without a resistor, Tinkercad will warn you about excessive current.
- By default, Tinkercad simulates the `Blink` sketch on the Arduino, toggling `Pin 13` on/off every second. You can change this later by editing the code.

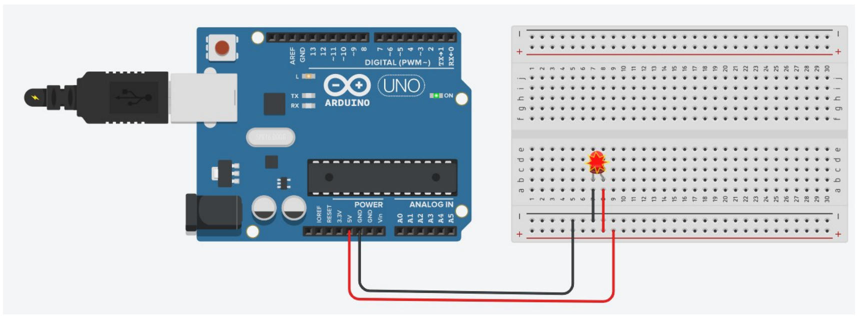


Figure 13.3: Initial Setup

5. **Start Simulation & Check for Warnings.**
   In the top-right corner, click **Start Simulation**. Hover over the LED or look at the bottom warnings:
   - Tinkercad likely flags the LED as drawing too much current. In reality, this setup would burn out the LED or damage the Arduino pin.
   - Stopping the simulation or modifying the circuit is necessary to fix the error.

6. **Add a 1 kΩ Resistor.**
   To protect the LED, place a **1 kΩ** resistor in series:
   - Disconnect the LED anode (long leg) from the Arduino pin.
   - Place the resistor so one end connects to the Arduino pin, and the other end to the LED anode on the breadboard (Figure 13.4).
   - The LED cathode (short leg) should go to `GND`.

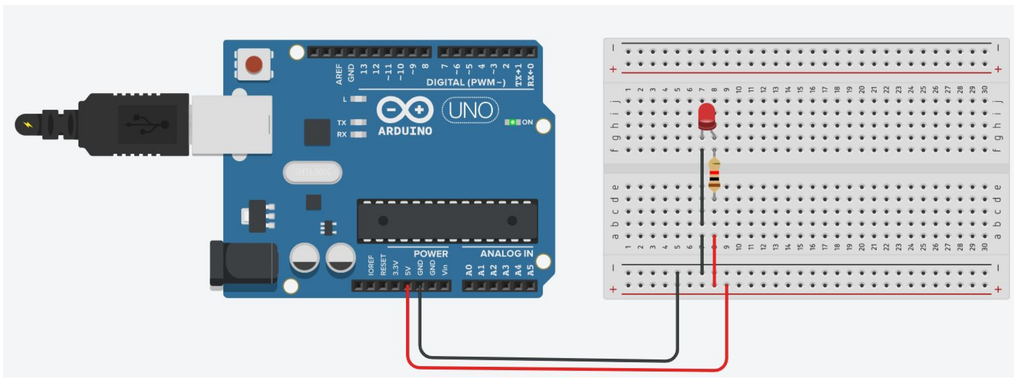


Figure 13.4: Adding a Resistor

7. **Run Simulation Again.**

   - Click **Start Simulation** once more.
   - Now the LED should blink at a safe current level, respecting the Arduino's default blink sketch (on `Pin 13`)—or whichever pin you've wired the LED to, if you change the code later.
   - If there are no warnings and the LED flashes periodically, you have a healthy, functioning circuit.

C. Using the Code Editor

8. **Open the Code Editor.**
   In your Tinkercad Circuits workspace, locate the **Code** button near the **Start Simulation** control.
   - Clicking this opens Tinkercad's built-in code interface, which often defaults to a **block-based** editor.
   - You'll see something similar to Figure 13.5, showing "Blocks" (for drag-and-drop coding) or "Blocks + Text."

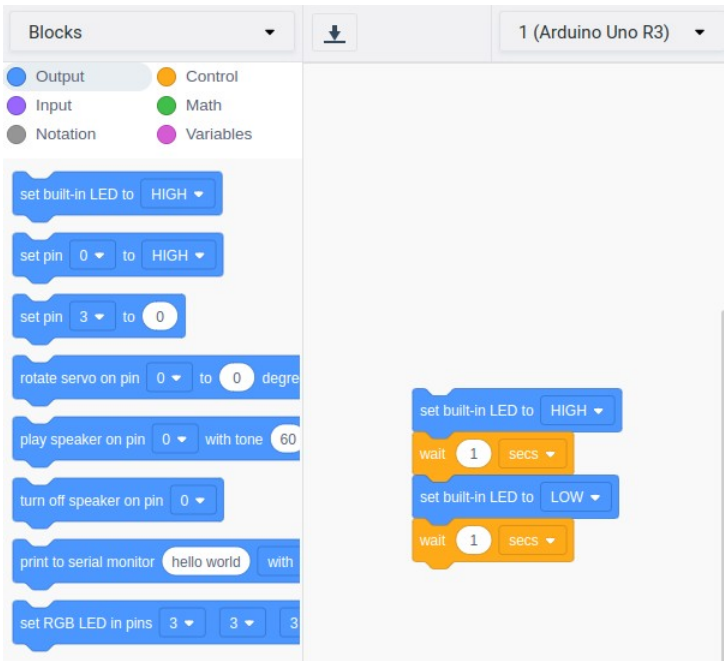


Figure 13.5: Tinkercad Code Editor

9. **Adjust the Delay or Switch to Text.**
   By default, Tinkercad provides a sample *Blink* program that toggles the built-in LED (digital pin 13) at a certain interval. You can:
   - Modify the **delay** values in the Blocks editor to make the LED blink faster or slower.
   - Click "**Blocks**" and choose "**Text**" or "**Blocks + Text**" mode to see or edit the underlying C++ `.ino` code.
   - In Text mode, you can rewrite the sketch completely—changing pins, adding more sensors or devices, etc.

   Tips:
   - Switching from **Blocks** to **Text** can help you learn how the block commands translate into Arduino C++ code.
   - If you prefer a purely textual approach, choose "**Text**" mode from the start.

10. **Experiment and Observe.**
    - Start the simulation again after making changes. Notice how different `delay(...)` values affect the blink rate.
    - If you have multiple components (e.g., a second LED or a servo), you can expand the

code to drive those as well, verifying how each behaves under simulation.
- Tinkercad automatically compiles the code and uploads it to the virtual Arduino when you click **Start Simulation**.

D. Printing Temperature to an LCD

11. **Add an LCD and Temperature Sensor (TMP36).**
In Tinkercad Circuits, you can expand your circuit to include a basic **16x2 LCD** and a **TMP36** temperature sensor, as shown in Figure 13.6. For the LCD pins, you typically wire them to Arduino digital pins (e.g., 12, 11, 5, 4, 3, 2) and use a 220 Ω resistor for the LCD's backlight or contrast. The TMP36 sensor connects its **signal** pin to **A0** for analogue input, while **V$_{\text{in}}$** and **GND** go to the Arduino's 5V and GND pins, respectively.

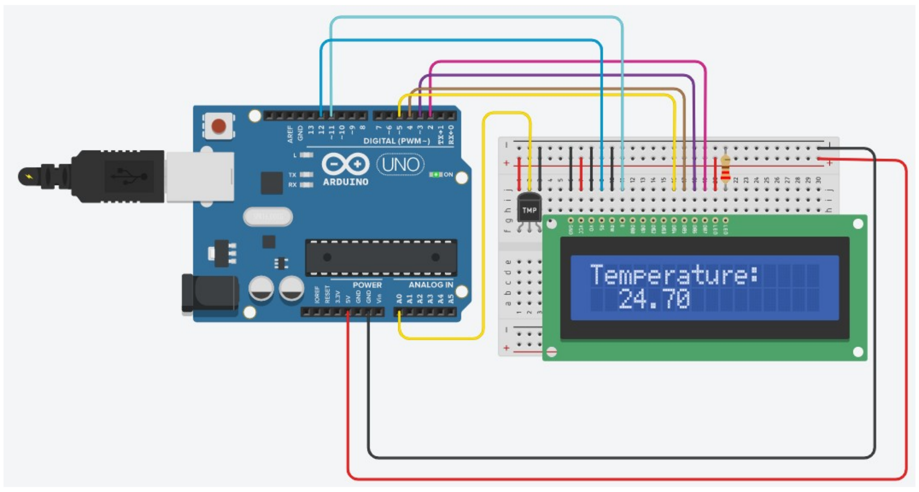


Figure 13.6: Printing Temperature to LCD

Tips:
- If you hover your mouse over the TMP36 in Tinkercad, you can see its pin labels (e.g., `GND`, `VCC`, `SIGNAL`). Match those to your Arduino.
- Sometimes Tinkercad defaults to a 16x2 LCD with pins labeled `RS, E, D4, D5, D6, D7`. These map to your Arduino pins (12, 11, 5, 4, 3, 2 in the example code).
- The 220 Ω resistor can be placed in series with the LCD's backlight pin or used as a contrast resistor, depending on your chosen wiring.

12. **Switch Code to Text.**
Open the Code Editor in Tinkercad and select "**Text**" mode from the dropdown near the top (instead of "Blocks"). Delete any existing blocks code. Then **paste** the following example code into the text editor:

```
// Example for Arduino Uno + 16x2 LCD + TMP36
#include <LiquidCrystal.h>

// Initialize LCD with digital pins (RS=12, E=11, D4=5, D5=4, D6=3, D7=2)
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup()
{
  // Start the LCD, specifying the 16x2 format
  lcd.begin(16, 2);
}

void loop()
{
  // Read the analog value from the TMP36
  int reading = analogRead(A0);

  // Convert the raw reading to voltage
  float voltage = reading * 5.0;
  voltage /= 1024.0;

  // TMP36 outputs 750mV at 25_C, with a scale of 10mV/_C.
  // Another common formula: temperatureC = (voltage - 0.5) * 100
  float temperatureC = (voltage - 0.5) * 100;

  // Print on top row
  lcd.setCursor(0, 0);
  lcd.print("Temperature:");

  // Print the numeric value on the second row
  lcd.setCursor(2, 1);
  lcd.print(temperatureC);
  lcd.print(" C");

  // Slow down updates for readability
  delay(1000);
}

```

— **Why This Formula?.** The TMP36 outputs:
- **0.5 V** at 0°C
- **750 mV** at 25°C
- each 10 mV = 1°C increment

So `temperatureC = (voltage - 0.5) * 100` converts the measured voltage to °C. For instance, `voltage=0.75` V → 25°C.

13. **Run Simulation.**

- Click **Start Simulation** to upload the code to the virtual Arduino.
- The LCD should initialize, and you'll see a **temperature** reading displayed.
- Tinkercad simulates the TMP36's behaviour; if you click on the sensor, you can sometimes adjust its properties (e.g., ambient temperature or offset) to see the readout change in real time.

Measuring Success

- **Circuit Simulation**:
  – The LED blinks without error messages, indicating correct resistor placement and proper wiring in Tinkercad.
  – No warnings about excessive current: Tinkercad's LED or resistor usage matches realistic safe limits.
- **Code Editor Usage**:
  – You comfortably switch between the "Blocks" and "Text" modes.

    - Adjusting `delay` values or pin assignments in the text code updates the simulation as expected.
    - You understand the default `Blink` sketch and how to modify it for other functionalities.
- **LCD + Sensor Integration**:
    - The TMP36 sensor's analogue reading is correctly converted to a temperature value in Celsius.
    - The LCD displays this temperature in real time.
    - Changing the TMP36 properties in Tinkercad (if available) alters the reading, confirming a responsive and dynamic simulation.

■

Use Case Scenario (Optional): "Greenhouse Monitor"
If you'd like to simulate a small greenhouse environment in Tinkercad:

- **Add a Humidity Sensor or Second Sensor:** While Tinkercad may not have a direct DHT11/22 sensor, you can mimic humidity input using another analog sensor (like a potentiometer) to emulate changing moisture levels. This second sensor's output is read by a different analog pin (e.g., `A1`).
- **Display Temperature & Humidity on LCD:**
    - Extend the code you used for the temperature sensor.
    - Create a `float humidity` variable and read the "sensor" value from `A1` (or wherever you place your potentiometer).
    - Convert that reading into a simulated "humidity" percentage, e.g.:

    ```
    humidity = map(analogValue, 0, 1023, 0, 100);
    ```

    - Print both `temperatureC` and `humidity` on separate lines of the LCD (e.g., "Temp: XX.XC / Hum: YY%").
- **Blink an LED for Thresholds:**
    - For instance, if temperature exceeds 30 °C **or** humidity passes 80%, you can blink a red LED to warn you.
    - In code, check:

    ```
    if (temperatureC > 30 || humidity > 80) {
        // blink LED
    } else {
        // LED off
    }
    ```

    - Experiment with different thresholds or color LEDs to show status (green for "safe," red for "alert").
- **Take Advantage of Tinkercad's Simulation:**
    - You can rotate the potentiometer in the simulator to simulate changing humidity levels.
    - Notice how the LCD updates in real time, and how the LED reacts when values pass your set thresholds.
    - This approach helps you verify logic before building a physical greenhouse monitor.

This setup demonstrates how you might combine multiple sensors (temperature, humidity) and actuators (LCD, LED) into a single Tinkercad circuit. By simulating dynamic inputs, you gain confidence that your "greenhouse monitor" logic is sound, all without needing the real hardware on hand.

— Further Reading

- **TMP36 Temperature Sensor**:
  `https://learn.adafruit.com/tmp36-temperature-sensor/using-a-temp-sensor`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the principles behind simulation-based microcontroller development, focusing on Tinkercad Circuits as a tool. Understanding how virtual environments model real hardware behaviors will help you prototype and debug circuits before building them physically.

A. Why Use a Simulator?

Microcontroller and circuit simulators (e.g., Tinkercad Circuits, Proteus, Fritzing) let you:

- **Experiment Without Hardware**: Test different component arrangements and code changes without risking physical damage.
- **Save Time & Resources**: Identify wiring mistakes or logic bugs quickly, eliminating the need for repeated breadboard rewiring.
- **Educate at Scale**: In a classroom or remote setting, students can all access the same virtual boards—no hardware bottlenecks or missing cables.

A.1 Limitations of Simulation

- **Real-World Physics Simplifications**: Tinkercad models basic currents and voltages, but doesn't always simulate noise, parasitic effects, or advanced IC behaviors.
- **Limited Component Library**: Some advanced or less common sensors might not exist in the simulator.
- **Performance Constraints**: Very large circuits can become slow to simulate or might not be fully supported.

B. Tinkercad Circuits Overview

Tinkercad Circuits is a web-based platform that combines:

- **Virtual Components**: Arduinos, LEDs, resistors, sensors, etc.
- **Breadboard Layout**: Wires and pins arranged like a real breadboard for quick prototyping.
- **Code Editor**: Allows you to write or generate `.ino` sketches (either block-based or text-based).

B.1 Simulation Workflow

1. **Place Components**: Drag an Arduino board, LED, sensors, etc., onto the workspace.
2. **Wire Connections**: Link pins and breadboard rails with color-coded wires.
3. **Add Code**: Use the built-in code editor (blocks or text) for the Arduino program.
4. **Start Simulation**: The simulator runs your code as if on a real microcontroller. Observations (like LED on/off, sensor readings) appear visually.

B.1.1 Switching Between Blocks and Text

- **Blocks Mode**: Great for beginners, easy to see logic flow.
- **Text Mode**: Full Arduino sketch view, enabling advanced coding, libraries, and more intricate logic.

C. Modeling Common Components in a Simulator

LED and Resistors:
- Tinkercad detects if an LED is driven without a current-limiting resistor (danger in real life).
- The simulator warns about potential overcurrent, helping you learn correct resistor placement.

Sensors (e.g., TMP36, LDR, PIR):
- Provide adjustable "environment" values. For a temperature sensor, Tinkercad might let you "turn a knob" to emulate changing temperature.
- The simulator calculates approximate voltage outputs or digital states based on the chosen parameter.

LCD Screens:
- Pin mappings replicate real-world wiring (like using pins 12, 11, 5, 4, 3, 2 for a 16x2 LCD).
- The simulator updates the displayed text as your Arduino code prints to the LCD library.

D. Debugging in a Simulated Environment

When **something goes wrong** in your circuit:
- **Check Wiring Color & Connections**: Is the resistor or sensor connected to the correct pin? Is ground (GND) established?
- **Error Messages**: Tinkercad might highlight short circuits or excessive current paths in red.
- **Serial Monitor**: Use 'Serial.print()' statements in code to track variable values or logic flow, just like on a physical Arduino.

D.1 Typical Issues

- **Pin Mismatch**: The code references `A0`, but the sensor is physically on `A1`.
- **Wrong Polarity**: The LED's anode/cathode reversed, or sensor pins swapped.
- **Missing or Incorrect Resistor**: Causing the simulator to show a meltdown symbol or a short circuit warning.

E. Advantages for Classroom and Remote Learning

- **Cost-Effective**: No need to buy multiple Arduinos, sensors, or boards.
- **No Risk of Hardware Damage**: Perfect for absolute beginners who might accidentally short pins or apply reverse polarity.
- **Collaboration**: Students can share links to their Tinkercad circuits, allowing instructors or peers to review and provide feedback.

E.1 Bridging to Real Hardware

- **Validation Step**: Once a circuit works in Tinkercad, replicate it physically to confirm real-world operation.
- **Physical Constraints**: In reality, layout, wire length, and power supply stability can matter more. The simulator only partially accounts for these.

F. Example Projects and Extensions

- **Blink an LED**: Easiest entry-level example.
- **Multiple LEDs in Sequence**: Show advanced timing or patterns.
- **Sensor Data Logging**: Use a simulated temperature sensor, print the values to Serial Monitor or LCD.
- **Button Debounce**: Code a stable input read from a pushbutton, learning about software debouncing in a safe environment.

F.1 Simulating More Complex Logic

- **Libraries Integration**: Some external libraries (like `Servo.h`, `Adafruit_Sensor.h`) may not be fully supported, but you can attempt partial testing.
- **State Machines or Finite Automata**: Prototyping logic flows before final deployment.

G. Best Practices for Effective Simulation

- **Label Your Components**: Provide descriptive names in Tinkercad so you recall which pin is used for what.
- **Use Distinct Wire Colors**: Keep a consistent color scheme (e.g., red for VCC, black for GND, green for signals) to reduce confusion.
- **Save Iterations**: Tinkercad auto-saves, but rename crucial versions (e.g., "Circuit_Rev2_with_LCD") for reference.
- **Cross-Check with Real Data**: Even though it's a simulator, approximate typical resistor values and sensor ranges. Don't rely solely on the simulator's placeholder logic if planning real deployment.

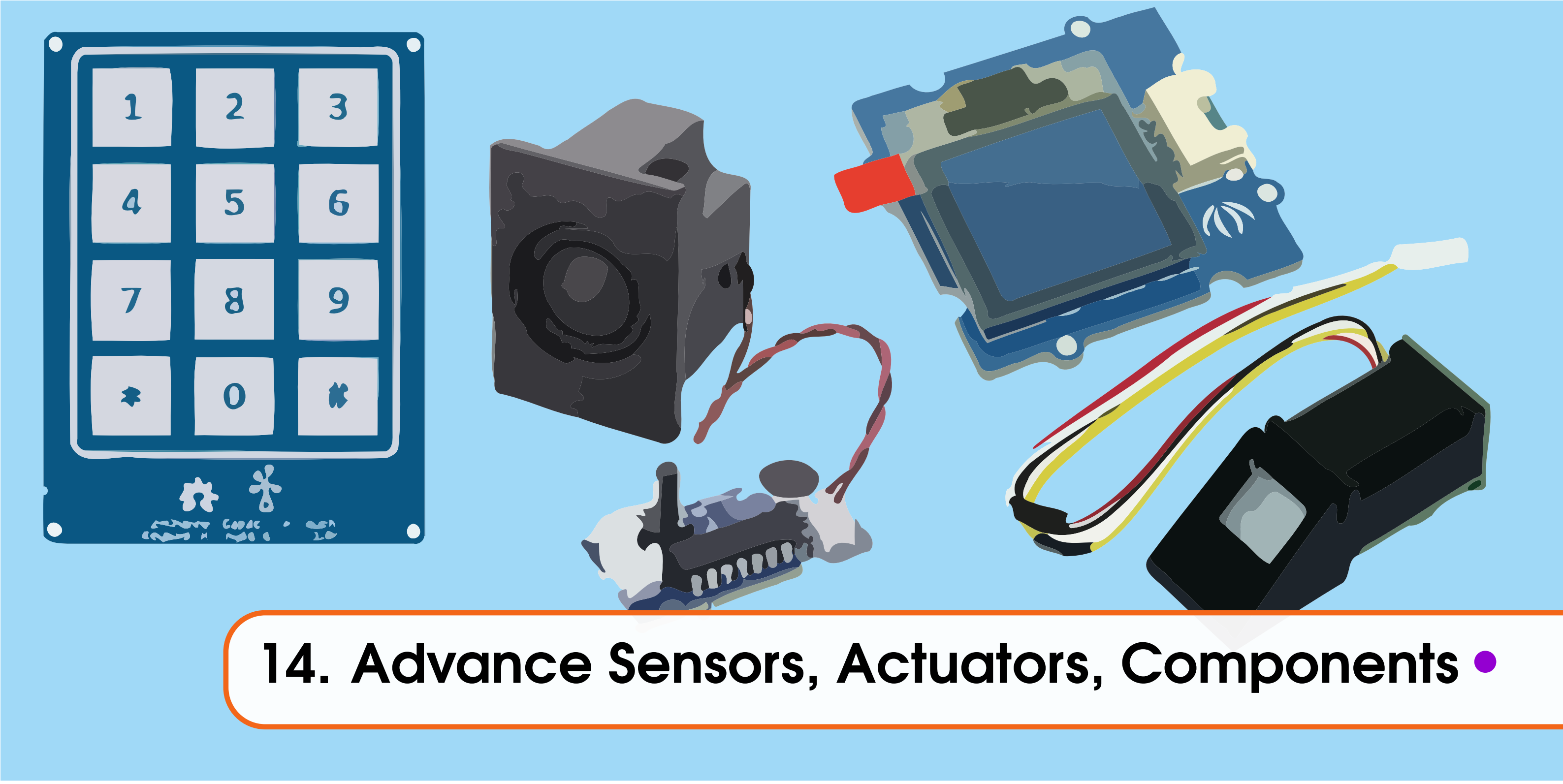
H. Conclusion and Future Explorations

Tinkercad Circuits and similar simulators provide a powerful environment for learning electronics and microcontroller programming:

- **Hands-On Virtual Experience**: Safe and quick iteration on circuit ideas, code logic, and sensor interactions.
- **Foundational Step**: Great for beginners or rapid prototyping. Eventually, real-hardware testing is vital for verifying actual performance.
- **Scalability**: Once comfortable, you can move to more advanced environments (like Proteus or real breadboards) and incorporate complex libraries or custom PCBs.

Final Note.

By building and testing circuits in a simulator, you gain confidence and refine your design before implementing it physically. Whether it's blinking an LED, reading sensor data, or driving an LCD display, Tinkercad Circuits offers a low-risk, high-reward way to understand electronics fundamentals—paving the way for more sophisticated prototyping and real-world IoT or embedded systems development.

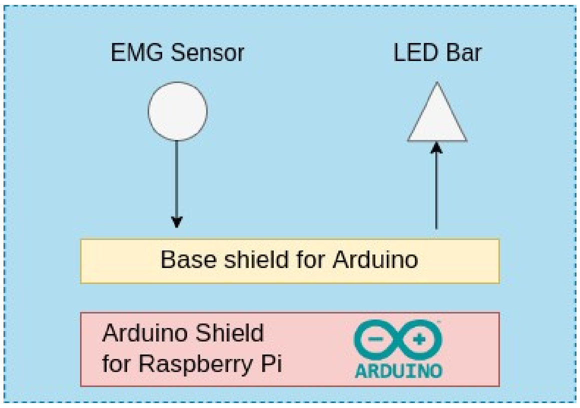


14. Advance Sensors, Actuators, Components

Objectives

- Learn how to map one sensor to another
- Learn how to use advanced sensors
- Get familiar with EMG Sensor

Lab Plan



In this lab, you will learn how to utilize more advanced sensors (like the Electromyography (EMG) sensor). You will connect an EMG sensor to Arduino and visualize muscle signals on a Grove LED bar.

Required Hardware Components

- **PC with internet connection**
- **Arduino Shield for Raspberry Pi** (with an onboard Arduino Leonardo chip)
- **Grove Base Shield**

- **Grove LED Bar v2.1**
- **Grove EMG Sensor v1.1**

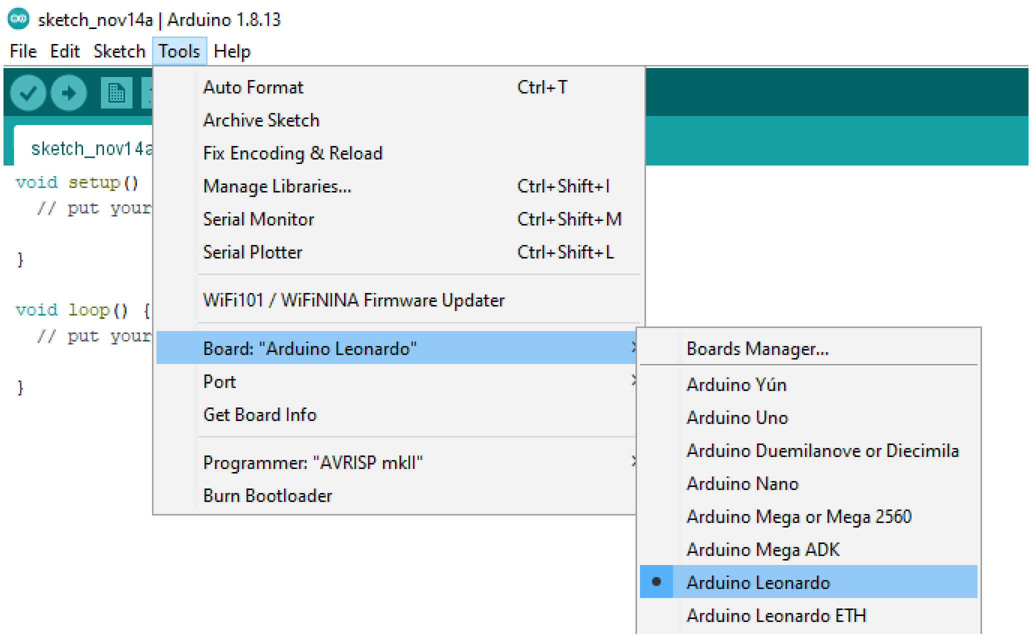
A. Setting Up the Arduino Environment

1. **Open Arduino IDE.**
   If you are using a lab machine, the Arduino IDE is likely preinstalled. Look for the Arduino icon on the desktop or in your applications menu. If you need to install it on your own computer, refer to the official Arduino documentation at `https://www.arduino.cc/en/Main/Software` for step-by-step instructions.

   > **— Why do this?.** The Arduino IDE allows you to write, compile, and upload sketches (programs) to the Arduino board. Ensuring it's properly installed and launched is the first step in programming.

2. **Connect your Arduino Shield to the computer via USB.**
   1. Plug the USB cable from the Arduino Shield (Leonardo-based) to your computer. 2. Wait a few seconds for your operating system to detect the device. 3. In the Arduino IDE, select:
   - **Tools** → **Board** → *Arduino Leonardo*
   - **Tools** → **Port** → pick the COM port (on Windows) or `/dev/ttyACM*` (on Linux/macOS) that appears/disappears when plugging/unplugging the board.

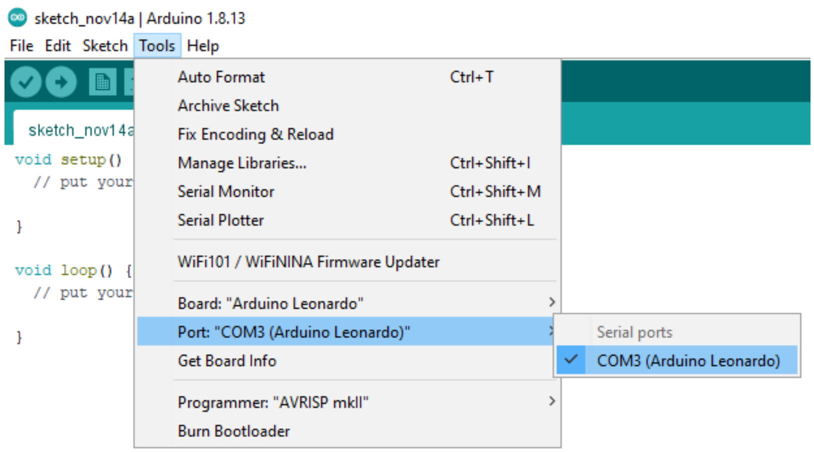
   See Figures 14.1 and 14.2 for examples.


sketch_nov14a | Arduino 1.8.13
File Edit Sketch Tools Help
sketch_nov14a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino Leonardo"
Port
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Boards Manager...
Arduino Yún
Arduino Uno
Arduino Duemilanove or Diecimila
Arduino Nano
Arduino Mega or Mega 2560
Arduino Mega ADK
● Arduino Leonardo
Arduino Leonardo ETH

```
void setup()
  // put your

}

void loop() {
  // put your

}
```

Figure 14.1: Configuring Arduino IDE Board

> **Tips:**
> - If you're unsure which port to select, unplug the board and see which port disappears from the list. Then plug it back in, and select that newly re-appeared port.
> - If no ports are listed, you may need to install Arduino drivers or check your USB cable (some are power-only and lack data lines).


sketch_nov14a | Arduino 1.8.13
File Edit Sketch Tools Help
sketch_nov14a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
Board: "Arduino Leonardo"
Port: "COM3 (Arduino Leonardo)"
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Serial ports
COM3 (Arduino Leonardo)

```
void setup()
  // put your
}

void loop() {
  // put your

}
```

Figure 14.2: Setting The Port

3. **Attach the Grove Base Shield.**
   Carefully align the base shield pins with the headers on your Arduino Leonardo board. Press down gently until it's seated firmly.

   > **— Why the Base Shield?.** The Grove Base Shield provides easy plug-and-play ports for sensors and actuators, reducing wiring complexity and eliminating the need for soldering or breadboards. Each Grove port is labeled for digital (D), analog (A), UART, or I^{2}C.

4. **Connect Sensors/Actuators.**
   Using Grove cables:
   - **LED Bar** → **D8** port (Digital pin 8 for controlling the LED bar level or patterns)
   - **EMG Detector** → **A0** port (Analog pin 0 to read muscle electrical activity signals)

   Confirm each cable is fully inserted in both the sensor/actuator port and the corresponding Grove port on the shield.

5. **Upload an Empty Sketch.**
   In the Arduino IDE, open a new sketch (with just `setup()` and `loop()`). Click the top-left upload icon (➜) to send this blank program to your board. If the IDE reports "Done uploading" with no errors, your board is recognized correctly. You can now proceed to programming the LED bar or EMG sensor as needed.

B. Setting Up the LED Bar

6. **Download LED Bar Library.**
   In order to drive the LED Bar easily, you need a dedicated library. Go to the GitLab repository and look for something like `Grove_LED_Bar-master.zip`:
   - Save the ZIP file to your computer.
   - In Arduino IDE, click **Sketch** → **Include Library** → **Add .ZIP Library...**.
   - Browse to the downloaded file, select it, and click **Open**.

   After successful installation, you should see the LED Bar library under **Contributed Libraries** in your IDE.

   > **Tips:**
   > - If you encounter an error like "library invalid" or "already exists," be sure to remove

any conflicting older versions from your Arduino libraries folder.
- Different Grove LED Bars sometimes have slightly different versions of the library. Confirm you have the one matching your hardware model.

7. **Open Example "Bounce."**
Once the library is installed, open a ready-made example:
   - In Arduino IDE: **File** → **Examples** → `(library name you just added)` → **Bounce**.
   - A new sketch window appears with example code that cycles the LED Bar in different patterns.

   Within the code, locate a line similar to:

```
Grove_LED_Bar bar(9, 8, 0, LED_BAR_10);
```

   This line specifies the pins used to control the LED Bar and its mode (`LED_BAR_10` often denotes 10 segments).
   Since you plugged the LED Bar into **D8** on your Grove Base Shield:
   - You might need to adjust this constructor so it references the correct digital pins used by that port (commonly `D8` and `D9`).
   - Check your library's documentation or any comments in the example to see how to map the shield port to actual Arduino pins. Often for a single Grove port, you have two pins: one for `CLK`, one for `DATA`.

   — Why specify pins 9, 8?. The `Grove_LED_Bar` library typically requires two digital pins for **data** and **clock**. If your shield maps the D8 port to these two lines (for instance, `CLK` on pin 9, `DATA` on pin 8), you must match that in `Grove_LED_Bar bar(9, 8, ...)`. Always verify the pin references in the shield's documentation or the library readme to ensure correct wiring.

8. **Observe the LED Bar Behavior.**
Finally, **verify** and **upload** the example. The LED bar should begin lighting in a "bounce" pattern:
   - One or more segments light in succession, moving back and forth along the bar.
   - If the code includes brightness or speed settings, you can experiment by modifying those variables.
   - Watch the Arduino IDE's **Serial Monitor** if the example prints diagnostic messages, or just observe the LED Bar physically.

   Tips:
   - If no segments light up, check that you selected the correct `DATA` and `CLK` pins in the code. Confirm the Grove cable is securely connected to D8 on the shield.
   - Try other examples in the library to learn how to set individual LED segments, display bar levels based on sensor inputs, etc.
   - Integrate with additional hardware: for example, read a sensor value (like a rotary angle sensor or an EMG sensor) and map it to the LED Bar level.

C. Setting Up the EMG Sensor

9. **Understand EMG Basics.**
An EMG (Electromyography) sensor detects the electrical potentials generated by muscle

cells when they contract. The sensor amplifies and processes these minute signals, providing an analogue output. Key points:

- **Amplitude Changes**:
  - When muscles are relaxed, the measured voltage is relatively low.
  - During contraction or flexing, the voltage spikes higher, reflecting stronger EMG signals.
- **Placement Matters**:
  - For consistent readings, place electrodes in line with the muscle fibers you want to measure (often the forearm or bicep).
  - Avoid areas with thick hair or bones directly beneath the skin, as this can interfere with electrode contact or produce weaker signals.
- **Skin Preparation**:
  - If in a real scenario, you might clean the skin with alcohol wipes to reduce noise.
  - In a lab environment or demonstration, ensure the electrode pads stick firmly to maintain good contact.

10. **Download the EMG Code.**
    Go to the GitLab repository and locate the Arduino sketch for the EMG sensor (e.g., `emg.ino`):
    - Download or copy the file.
    - Open it in your Arduino IDE.
    - Verify you have the correct board (*Arduino Leonardo*) and port selected.
    - **Upload** the code to your board.

Tips:
- Some EMG sensors require an external power supply or reference. Confirm your Grove EMG sensor's documentation for proper voltage (5V vs. 3.3V).
- Ensure the sensor is plugged into the correct port (e.g., **A0**) if the code references `analogRead(A0)`.

11. **Attach Electrodes.**
    The EMG sensor typically comes with disposable electrode pads:
    - **Wait to Remove Covers**: Keep the adhesive covers on until you're ready to use them, so they don't dry out.
    - **Placement** (Figure 14.3):
      - One (signal electrode) near the center of the muscle you want to measure (e.g., forearm flexors).
      - Another electrode can be placed near the same muscle group but a few centimeters away.
      - A reference electrode is often placed on a bony area (or less active muscle) to serve as a baseline, depending on your sensor's instructions.
    - Plug each electrode cable lead into the sensor board as indicated (e.g., `IN+`, `IN-`, `REF`).

— Why Electrode Placement Matters?. EMG signals are faint and easily affected by noise. Properly positioning electrodes over the muscle belly (where electrical activity is strongest) helps capture clear signals. A poor or incorrect placement may yield low or inconsistent readings.

12. **Relax, then Move Muscles.**

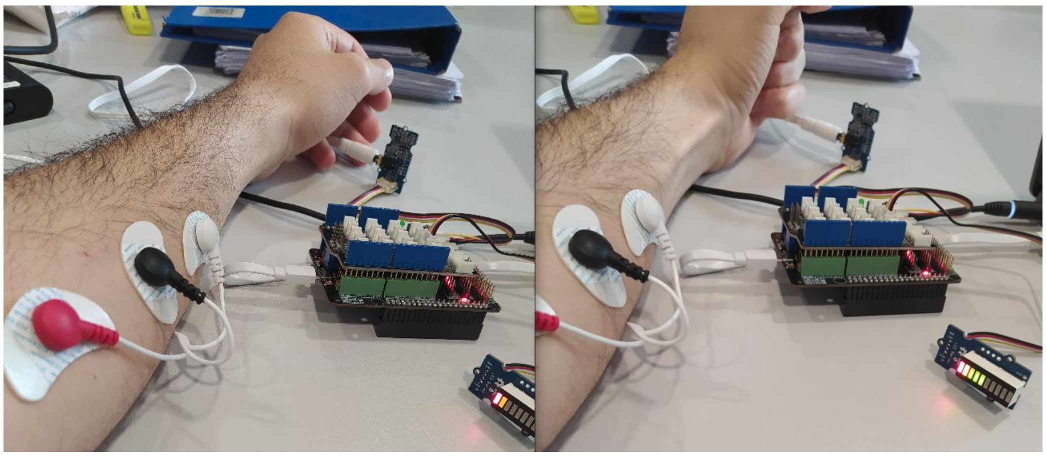


Figure 14.3: Example Experiment with EMG Sensor

Run the uploaded sketch:

- **Observe the LED Bar** (if the code maps EMG to LED levels):
  - In a relaxed state, only a few segments (if any) light up.
  - When you flex the targeted muscle, the EMG sensor output increases, driving more segments on the LED bar.
- **Check Serial Monitor** (if the code prints raw or processed EMG values):
  - Notice the baseline reading at rest and how it spikes during flexing.
  - If the code includes thresholds, verify how the program transitions between "low EMG" and "high EMG" states.
- **Reposition if needed**:
  - If signals are too weak or unstable, adjust electrode placement or ensure the sensor cables are firmly connected.
  - Keep your arm still during rest to see a stable baseline, then contract only the muscle under test.

Tips:

- Keep wire movement minimal, as jostling cables can introduce noise or spikes.
- If the sensor includes a gain adjustment pot, you can tweak it to amplify or reduce signal strength.
- If you want to read EMG data for further analysis, you might log the analog values over time or use an additional data-plotting tool.

Measuring Success

- **LED Bar Operation**: You have successfully uploaded the "Bounce" example (or any other LED Bar demo), and the bar lights in a cycling or responsive pattern. Modifying the code's timing or level logic yields the expected changes on the LED segments.
- **EMG Sensor Readings**: With your EMG electrodes placed correctly, the LED Bar reflects muscle activity. When your arm is relaxed, only a few LEDs (or none) illuminate; flexing the muscle causes more LEDs to light up, indicating higher EMG amplitude.
- **No Errors in Upload/Serial Output**: Uploading sketches in the Arduino IDE completes without warnings. If the EMG code prints data to the Serial Monitor, no unexpected errors appear, and the readings (voltages or raw EMG values) change as you contract or relax

your muscles.

Use Case Scenario (Optional): "Real-Time EMG Display and Feedback"

Envision enhancing your EMG project by showing the muscle activation level on both a Grove LCD and an LED bar in real time:

- **Live EMG Values on LCD**:
  - Add a Grove LCD (e.g., RGB Backlight 16x2) connected to an I^2C port on your Arduino shield.
  - In your EMG code, after reading the EMG sensor's analogue value (e.g., `int emgValue = analogRead(A0);`), convert it to a human-friendly metric (e.g., `float muscleLevel = map(emgValue, 0, 1023, 0, 100);`).
  - Print `muscleLevel` on the LCD in **loop()**:

    ```
    lcd.setCursor(0,0);
    lcd.print("EMG: ");
    lcd.print(muscleLevel);
    lcd.print(" %");
    ```

  - Update every few hundred milliseconds, so you see the muscle activity changing as you flex.
- **Matching LED Bar Color/Intensity**:
  - Already have an LED bar connected at D8 (digital).
  - Either use your existing code that lights segments based on `emgValue` (low = fewer LEDs, high = more), or incorporate color changes if it's an RGB LED bar.
  - You could define thresholds, e.g. "`< 300` means only the first 2 segments, `300-700` lights half the bar, `> 700` lights all segments."
- **Color Change on LCD**:
  - If you have an RGB Backlight LCD, you can modify its color via `lcd.setRGB(r, g, b);`:

    ```
    if (emgValue < 300) {
      lcd.setRGB(0, 0, 255);  // blue for calm
    } else if (emgValue < 700) {
      lcd.setRGB(255, 165, 0); // orange for medium
    } else {
      lcd.setRGB(255, 0, 0);   // red for high EMG
    }
    ```

  - This gives an immediate visual indication of the muscle activation level.
- **Optional Buzzer Alert**:
  - Connect a buzzer to another digital pin (e.g. D6).
  - If the muscle activity is above a certain threshold, beep the buzzer:

    ```
    if (emgValue > 700) {
      tone(6, 1000, 200); // beep at 1kHz for 200ms
    }
    ```

  - This can act as an alarm or feedback for training scenarios (e.g. muscle rehab).
- **Integration Notes**:
  - Sampling Rate: Continuously reading EMG in **loop()** might produce rapid changes. A `delay(100)` or so can smooth updates; or implement averaging if the signal is noisy.
  - Serial Monitor Debug: Keep printing raw values to the Serial Monitor if you want to see detailed data logs. The LCD can show a simpler, user-friendly version.

  - User Comfort: Ensure electrode cables are stable; continuous motion or stretching can introduce spikes or dropouts.

This scenario fuses visual feedback (LCD text + LED bar color/intensity) with muscle activity from the EMG sensor, providing a more intuitive interface. It's especially useful for training, therapy, or even creative applications where you want the body's movements to drive sound, light, or data displays.

— Further Reading
- Learn more about the EMG detector:
  `https://wiki.seeedstudio.com/Grove-EMG_Detector/`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the underlying principles of electromyography (EMG) sensors and other advanced components—like LED bars—for capturing, visualizing, and responding to biological and environmental signals. Understanding how these sensors work will help you integrate them effectively in real-world applications, from biomedical monitoring to interactive installations.

A. Electromyography (EMG) Fundamentals

Electromyography (EMG) is the technique of measuring the electrical signals generated by muscle fibers during contraction:
- **Muscle Electrical Activity:** When you flex or tense a muscle, the body's motor neurons send electrical impulses that can be detected on the skin's surface.
- **Surface Electrodes:** By placing electrodes on the skin above a target muscle, you can measure minute voltage differences (on the order of microvolts to millivolts).
- **Signal Amplification & Filtering:** An EMG sensor module typically includes an amplifier and filter circuitry to boost and clean the raw muscle signal so a microcontroller can interpret it.

A.1 Key EMG Sensor Concepts

- **Placement Matters:** Electrodes positioned over the muscle belly yield stronger signals than near tendons or bony landmarks.
- **Baseline (Noise) vs. Activation:** A relaxed muscle may show a small baseline, while a flexed muscle can significantly increase the EMG amplitude.
- **Skin Preparation:** For more accurate readings, some EMG setups require cleaning the skin (removing oils) or using conductive gel/paste. Basic modules often do not, but still benefit from stable electrode contact.

A.1.1 Safety and Usage Disclaimer

- **Low-Voltage Signals:** EMG sensors used in hobbyist/educational contexts typically operate at safe voltage levels. However, always ensure equipment is in good condition (no damaged wires, no exposure to mains power).
- **Skin Irritation:** Some individuals may be sensitive to adhesive electrodes. Discontinue or reposition if irritation occurs.
- **Intended Educational Use Only:** Hobby-grade EMG sensors are not medical de-

vices; data are for demonstration or R&D, not for diagnosing health conditions.

B. Mapping EMG Signals to Outputs

LED bars, buzzers, or servo motors can visually or physically represent EMG magnitude:

- **Analog-to-Digital Conversion (ADC):** The EMG sensor's output is typically an analog voltage. The Arduino reads it on an analog pin (e.g., A0) with a range of 0–1023 (for a 10-bit ADC).
- **Signal Thresholding or Scaling:** In code, you might map raw EMG values (e.g., 0–500) to 0–10 segments on an LED bar to show how "strong" the muscle contraction is.
- **Real-Time Feedback:** This direct mapping encourages interactive "biofeedback." As you flex, you see an immediate response on the LED bar.

B.1 LED Bar Considerations

- **Serial-to-Parallel Driver**: Many LED bars have built-in driver circuitry controlled via two digital pins (data, clock). Libraries abstract the shifting logic.
- **Brightness Levels**: Some advanced bars allow partial brightness steps; simpler ones are on/off for each segment.
- **Power Requirements**: Although each segment is typically a small LED, driving many simultaneously can draw noticeable current. Ensure a stable 5V supply.

B.1.1 Using PWM for Smooth Control

- **Dimming or Smooth Transitions**: If hardware supports it, you can drive segments at different duty cycles for a smoother gradient effect.
- **Software or Library Approach**: Some LED bar libraries handle partial brightness automatically. Others only toggle each LED fully on or off.

C. Advanced Sensor Types and Integration

Beyond EMG, there are many other advanced sensors to explore:

- **EEG (Electroencephalography):** Brainwave detection. Typically more delicate and specialized than EMG.
- **ECG (Electrocardiography):** Heart electrical signals (cardiac cycle). Similar electrode principles to EMG but different signal ranges.
- **GSR (Galvanic Skin Response):** Measures skin conductivity changes due to sweat (stress or emotional states).
- **Environmental Sensors (CO2, Gas, Particulate Matter):** Useful for advanced IoT projects, requiring specialized calibration.

C.1 Interfacing Strategies

- **Analog vs. Digital Output**: Some advanced sensors (like an EMG board) produce analog voltage. Others communicate via I2C/SPI.
- **Power and Noise Isolation**: High-gain sensors can be susceptible to interference; consider separate grounds or shielded cables to reduce noise.
- **Data Filtering or Smoothing**: In software, a moving average or low-pass filter can remove spurious spikes, improving readability.

D. Real-World Applications and Project Ideas

- **Muscle-Controlled Prosthetics**: EMG signals can drive servo motors or exoskeletons, providing partial hand or arm movement for assistive devices.
- **Gesture Control Interfaces**: A user might flex different muscle groups to trigger distinct system actions (e.g., navigating a menu, controlling a robot).
- **Biofeedback & Rehabilitation**: Visual or auditory cues (LED bars, tones) help patients learn to control muscle activation or practice therapy exercises.
- **Art Installations**: EMG signals transformed into dynamic lighting, color, or music, creating interactive experiences that respond to muscle movement.

D.1 Safety and Reliability for Real Implementation

- **Signal Calibration**: Each individual's EMG baseline or max amplitude can vary. Consider a calibration phase to tailor thresholds.
- **Ambient Electrical Interference**: Fluorescent lights, power cables, or other electronics can distort the sensor reading. Keep wires short or twisted, separate sensor cables from high-power lines.

E. Best Practices for Working with EMG & Advanced Sensors

- **Start with Low Gains**: Some EMG modules have adjustable gains. Begin at moderate settings to avoid saturation or excessive noise.
- **Use Shielded Electrodes**: If you need more stable readings, or are in an electrically noisy environment, shielded electrode cables help reduce interference.
- **Ensure Proper Skin Contact**: Loose electrodes lead to artifacts. Use consistent placement for repeatable results.
- **Understand Library Docs**: For the LED bar or specialized sensors, read documentation for pin connections, library calls, and example sketches to avoid hidden pitfalls.

E.1 Code Organization Tips

- **Modular Functions**: Keep sensor reading, data processing, and output control in separate functions for clarity.
- **Serial Debugging**: Print raw EMG values occasionally to verify signals. Optionally store them in arrays for offline analysis or plotting.

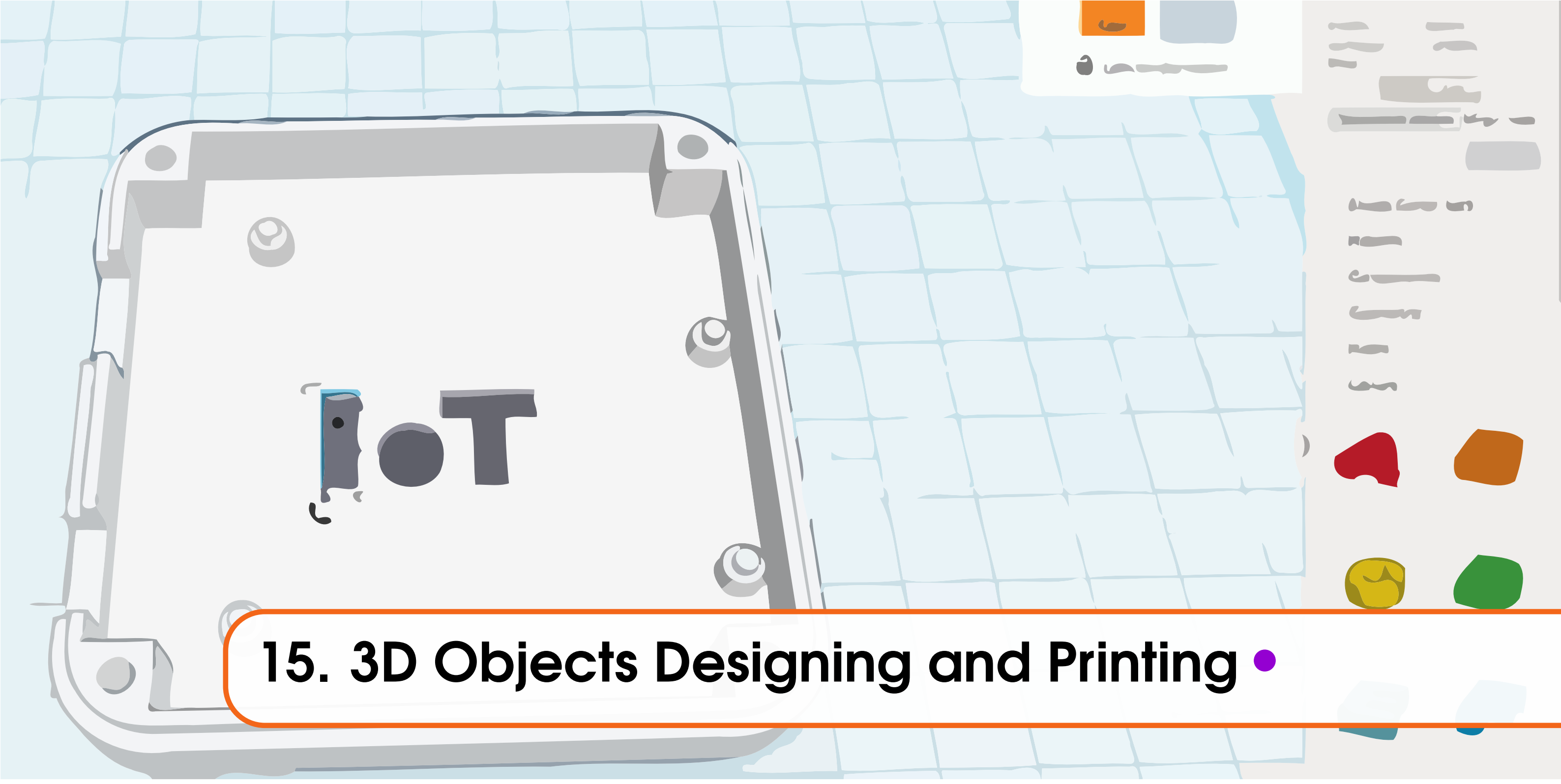
F. Future Extensions and Exploration

By combining an EMG sensor with other modules, you can create richer systems:

- **Machine Learning on Microcontrollers**: Classify different muscle movements or gestures in real time with TinyML frameworks.
- **Wireless Data Transmission**: Send EMG data via Bluetooth or Wi-Fi to remote dashboards or mobile apps for telemetry or logging.
- **Closed-Loop Control**: Drive motors or solenoids based on muscle activity. E.g., a pneumatic clamp that only engages when the user flexes a specific muscle.

Final Note.

Exploring advanced sensors like **EMG** fosters a deeper appreciation for bio-signal detection, signal conditioning, and real-time responsiveness in electronics. Coupling these signals with a visually engaging output device (e.g., an LED bar) provides immediate, tangible feedback and opens doors to creative, interactive applications—from assistive robotics to artistic biofeedback installations. Always keep in mind safety, calibration, and environmental noise considerations when working with high-gain or biologically derived signals. Enjoy pushing the boundaries of what your microcontroller can sense and do with advanced sensor technology.



15. 3D Objects Designing and Printing

Objectives

- Learn how to design 3D models in Tinkercad
- Learn how to manipulate shapes and generate `.STL` files
- Learn how to prepare models for 3D printing

Required Hardware Components

- **PC with internet connection**

A. Getting Started with Tinkercad

1. **Visit `www.tinkercad.com`.**

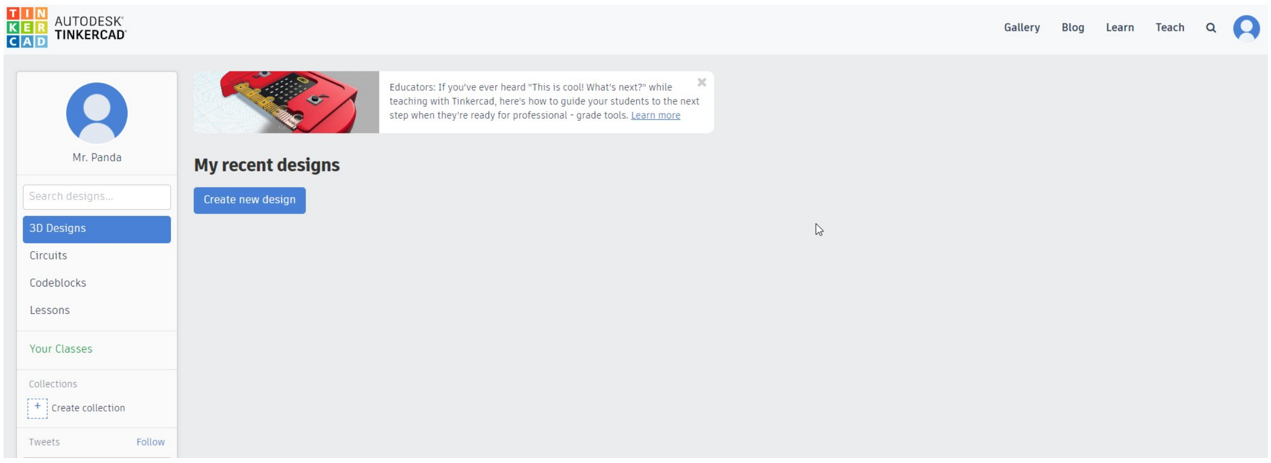
    - Click the **Start Tinkering** button on the homepage.
    - If prompted, sign in with your Google, Apple, or Autodesk account (or create a new one if you don't have one).

    Tinkercad provides a free, web-based platform for 3D design, electronics (circuits), and code simulation.

    > **— Why Tinkercad?.** Tinkercad is beginner-friendly and doesn't require local installations or drivers. You can build 3D objects, simulate Arduino circuits, or create code blocks entirely in your browser. See the official documentation at `https://www.tinkercad.com/learn` for tutorials and detailed guides.

2. **Explore the Workspace.**
    After logging in, you'll arrive at your Tinkercad dashboard (Figure 15.1). Look for the section labeled **My recent designs**:
    - Click **Create new design** under the 3D Designs tab to open a blank 3D workplane (Figure 15.2).

- Alternatively, choose **Circuits** if you plan to simulate Arduino projects or other electronics.



Educators: If you've ever heard "This is cool! What's next?" while teaching with Tinkercad, here's how to guide your students to the next step when they're ready for professional - grade tools. Learn more

Figure 15.1: Tinkercad Workspace

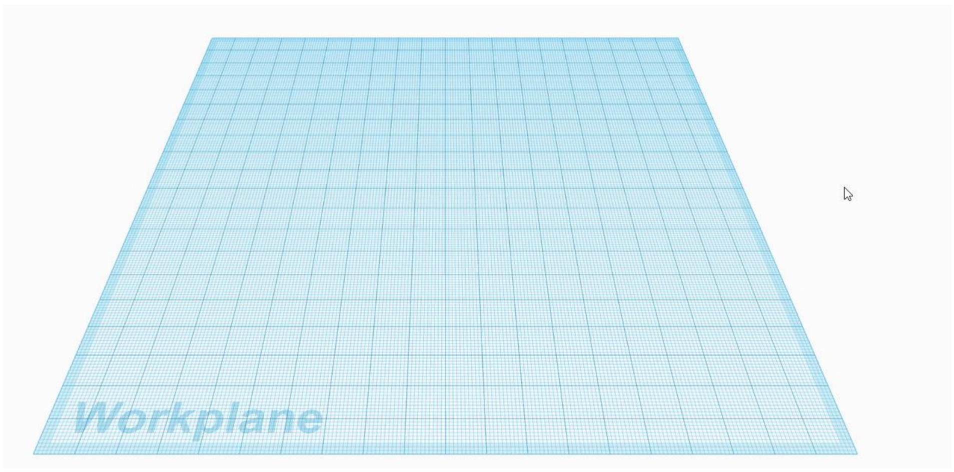


Figure 15.2: Tinkercad Workplane

Tips:

- **Rename** your project by clicking the random default name at the top-left (e.g., "Glorious Llama"). Give it something descriptive, like "MyTestDesign."
- Switch tabs (e.g., from 3D **Designs** to **Circuits** or **Codeblocks**) depending on what you want to create.

3. **Navigation.**
   In the 3D design environment:
   - Use your mouse wheel to **zoom in** or **out**.
   - **Right-click + drag** (or hold `Shift` + right-click) to orbit around the workplane.
   - Middle-click + drag (or hold `Ctrl` + right-click) to pan the camera view.

   Experiment with these controls to position your design. If you ever get lost, click the **Home View** button (a small house icon) to reset your camera.

B. Creating Basic Objects

4. **Drag & Drop Shapes.**
   In the right-hand **Shapes** panel (within Tinkercad's 3D workspace), locate a **box** and drag it onto the workplane.
   - **Height**: 80 mm
   - **Length**: 80 mm
   - **Width**: 200 mm

   You'll see a large rectangular prism, as shown in Figure 15.3. The `height` setting changes the box's vertical dimension, while `length` and `width` affect the horizontal footprint.

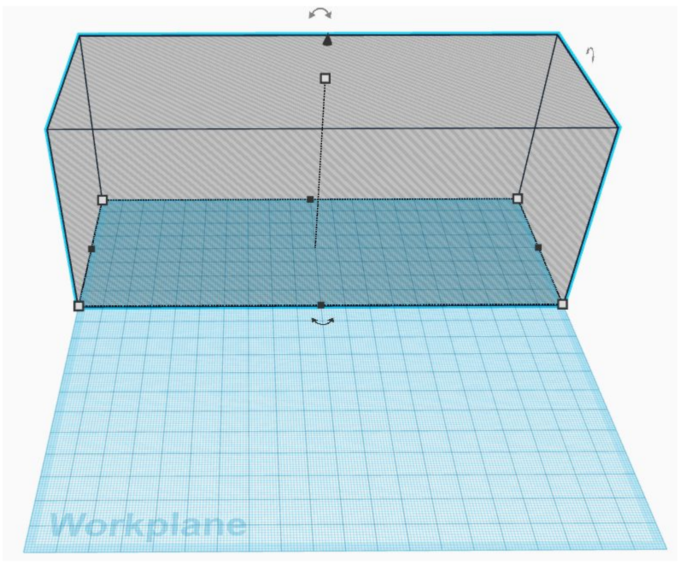


Figure 15.3: Tinkercad Box

— Why define these dimensions?. Assigning specific measurements helps you practice precise 3D modeling instead of just eyeballing shapes. When 3D printing or exporting models, correct sizing is essential.

5. **Adding a Second Box.**
   Create another **box** with slightly smaller or different dimensions:
   - **Height**: 75 mm
   - **Length**: 75 mm
   - **Width**: 120 mm

   Move it *inside* the first box by carefully selecting and dragging it. For precise movements:
   - Press **Right Arrow** 5 times to shift it 5 mm on the X-axis.
   - Press **Down Arrow** 2 times to move it 2 mm along the Y-axis.
   - Press **Ctrl + Up Arrow** 5 times to raise it 5 mm above the workplane.

   Viewed from the left, you should see something like Figure 15.4.

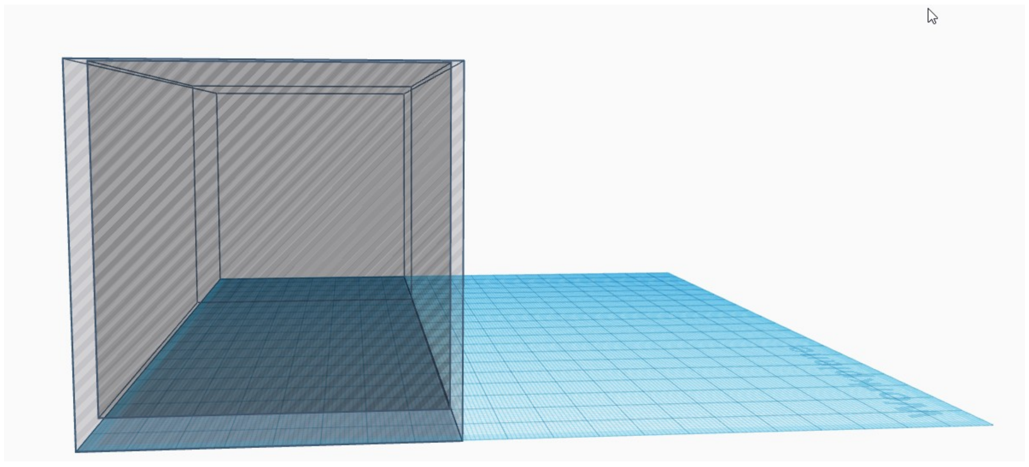


Figure 15.4: Box Placement

Tips:

- Adjusting the "snap grid" settings at the bottom-right corner of Tinkercad can give you finer (e.g., 0.1 mm) or coarser (e.g., 1 mm) movement increments.
- Use the **align** tool (shortcut key `L`) if you want to center or align shapes easily.

6. **Adding Cylinders.**
   Now drag a **cylinder** shape onto the workplane:
   - **Width**: 55 mm
   - **Length**: 55 mm
   - **Height**: 75 mm

   Position it so it's partially above the plane (raise it 5 mm, for instance). Make the outer box **solid** if you haven't already. The arrangement might resemble Figure 15.5.

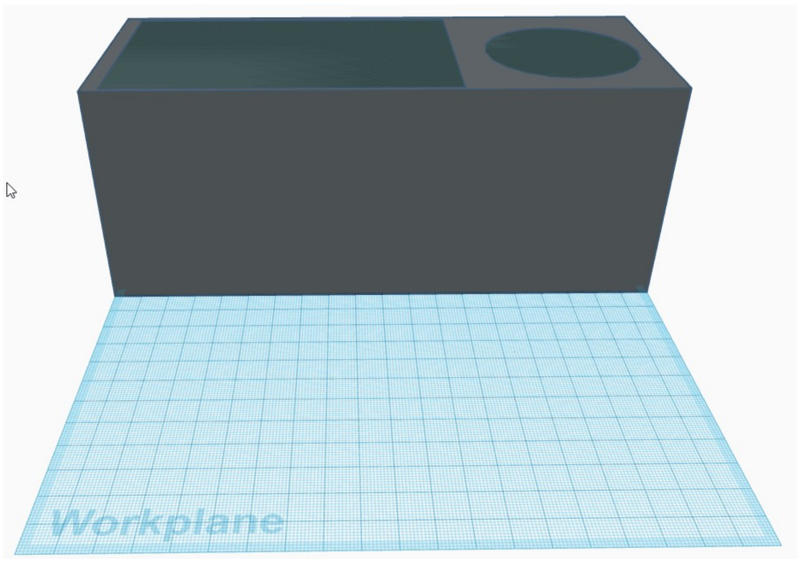


Figure 15.5: Cylinder Placement

— **Why add a cylinder?.** By mixing boxes and cylinders, you can create hollow compartments or design features (like round posts or pillars). Tinkercad allows grouping and "hole" operations to subtract shapes—useful if you're constructing something like a container with cylindrical cut-outs.

C. Finalizing Your Design

7. **Combine Shapes.**
   At this point, you can group multiple objects (boxes, cylinders, etc.) to form a cohesive "IoT box" or any other container design. For instance:
   - **Large rectangular box** for housing a Raspberry Pi, Arduino, or other electronics.
   - **Smaller inner boxes** to store sensors, Grove cables, or other small components.
   - **Cylinders or holes** for pens, pencils, or even cable pass-throughs.

   Figure 15.6 shows an example final setup with multiple compartments and cylindrical cutouts.

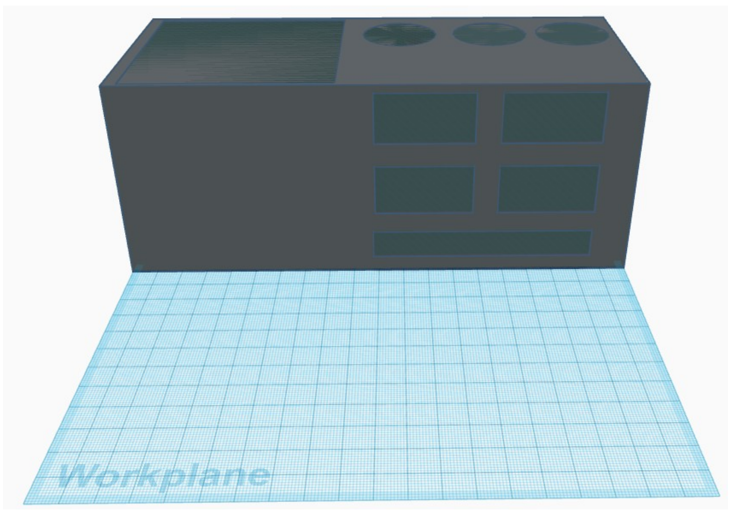
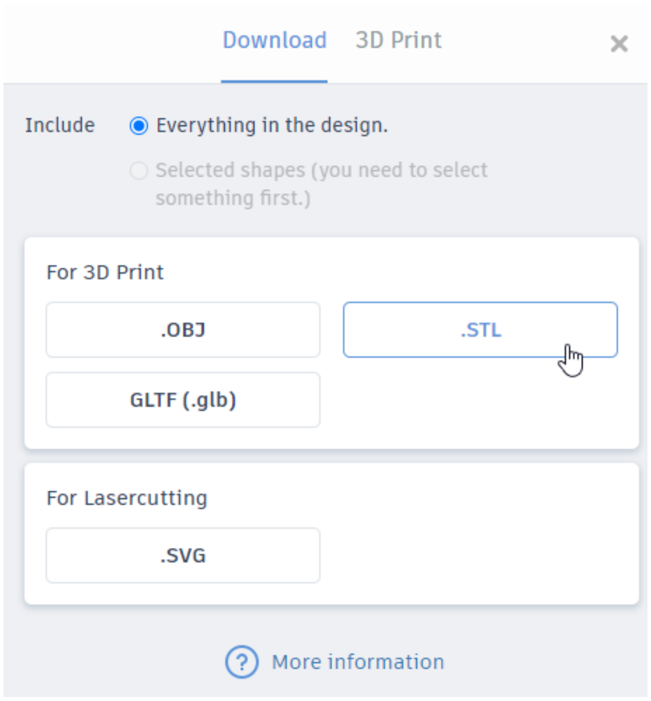


Figure 15.6: Tinkercad Final Setup

Tips:
- **Group vs. Hole**: Use the "*Group*" tool (Ctrl+G or the grouping icon) to merge shapes into one. Use the "*Hole*" property on a shape if you want to subtract that shape from another (like cutting out a cylinder in the box).
- **Snap Grid Adjustments**: If you need finer placement than 1 mm steps, look for the "Snap Grid" setting at the lower right of the Tinkercad screen and set it to 0.5 mm or 0.1 mm for precision.

8. **Exporting as `.STL`.**
   When you've finalized your design and want to 3D print or share it:
   - Click the **Export** button at the top-right corner of Tinkercad.
   - Choose **.STL** as the file format (see Figure 15.7).
   - Tinkercad will download an `.stl` file, which you can import into a slicing program (e.g., Cura, PrusaSlicer) to generate G-code for your 3D printer.


Download 3D Print

Include ● Everything in the design.
○ Selected shapes (you need to select something first.)

For 3D Print
.OBJ .STL
GLTF (.glb)

For Lasercutting
.SVG

More information

Figure 15.7: Exporting as STL

> **— Why `.STL`?.** `.STL` (Standard Triangle Language) is one of the most common 3D file formats for additive manufacturing. It breaks the surface of your 3D model into a mesh of triangles, which most slicing/printing software understands. Alternatives like `.OBJ` or `.GLTF` can also be used, but `.STL` remains a universal standard in 3D printing.

Measuring Success

- **Basic Shapes Mastery**:
  - You can comfortably drag and drop boxes or cylinders onto the Tinkercad plane.
  - Resizing and precise movement (via arrow keys or snap grid adjustments) are clear to you.
  - Rotating the camera or panning around the workplane feels intuitive.
- **Combining Objects**:
  - You've used the "Group" command (or "Hole" + "Solid" operations) to sculpt your 3D model into compartments, cylindrical cutouts, or other custom forms.
  - You understand how to align and adjust multiple shapes to create a cohesive design—like an "IoT storage box" with dedicated sections for boards, sensors, and cables.
  - No unexpected gaps, overlaps, or misalignments persist in your final arrangement.
- **Exporting an STL**:
  - You clicked **Export** in Tinkercad and saved your model as a `.stl` file.
  - The file successfully downloads without errors, confirming that Tinkercad recognizes your model as a valid 3D object.
  - You're prepared to load this `.stl` into a slicer (e.g., Cura, PrusaSlicer) for 3D printing, or share it with collaborators who can print or further modify your design.

Use Case Scenario (Optional): "IoT Enclosure"

You can expand your Tinkercad skills by designing a more specialized enclosure for your sensors and boards:

- **Create compartments**:
  - If you have multiple sensors (e.g., ultrasonic, temperature, humidity), design separate internal "rooms" or "bays" in your 3D model.
  - Consider adding vertical walls or partial dividers. This helps keep components in place without sliding around.
- **Holes or Slots for Cables/Air Vents**:
  - Draw "cylinders" in Tinkercad, mark them as *hole*, and align them with your main enclosure wall. Group them to "cut out" cable pass-through openings.
  - For ventilation, arrange a grid or pattern of small circular holes on the enclosure's side or top to help dissipate heat from a running Raspberry Pi or Arduino.
  - If you have an LED inside for status, cut a hole aligned with where the LED will be, so its light is visible from outside the enclosure.
- **Customize Dimensions**:
  - Measure the length, width, and height of your hardware (e.g., **Raspberry Pi** is approximately 85mm × 56mm × 17mm, while an **Arduino Uno** is about 68.6mm × 53.4mm).
  - Add extra **tolerance** (1–2mm) around these dimensions to ensure a snug but not overly tight fit.
  - If you want to secure boards with screws, plan small pillars or standoffs inside your model (again using cylinders as "holes" to create screw mounts).
- **Final Assembly Thoughts**:
  - Consider splitting the enclosure into two main pieces (a **base** and a **lid**). Each can be printed separately. The lid might include cutouts for sensor "eyes" (like an ultrasonic sensor).
  - Add text labels or embossed logos by dragging the "*Text*" shape, adjusting its height to a minimal extrusion, and grouping it with your base object.
  - If you expect frequent cable changes, ensure your cable holes are large enough, or add removable "lid" sections.

By combining these ideas, you create a functional, aesthetically pleasing enclosure to protect and organize your IoT hardware, all simulated and tested virtually in Tinkercad before committing to a 3D print or final fabrication.

— Further Reading

- Learn more about Tinkercad design:
  `https://www.tinkercad.com/learn/designs`

Theory Deep Dive: Underlying Principles and Concepts

This section explores the core principles of creating 3D models (using Tinkercad or similar CAD tools) and preparing them for 3D printing. By understanding these concepts, you can design more functional, robust, and aesthetically pleasing prototypes that suit your project requirements.

A. Basics of 3D Modeling

3D modeling involves creating or manipulating shapes in a digital environment to form complex objects:

- **Primitives:** Basic shapes like cubes, cylinders, spheres serve as the building blocks.
- **Transformations:** Translate (move), rotate, scale, or combine these shapes using Boolean operations (union, subtract, intersect).
- **Precision vs. Creativity:** You can make highly accurate mechanical parts (requiring precise dimensions) or more artistic/organic designs.

A.1 Tinkercad's Role

- **Beginner-Friendly Interface:** Drag-and-drop shapes, easy set of numeric inputs for sizing and positioning.
- **Cloud-Based:** No local installation required. Projects are stored online.
- **Rapid Iteration:** Quickly visualize changes, clone objects, and experiment with layout ideas.

A.1.1 Other CAD Tools

- **Fusion 360, SolidWorks, Blender:** More advanced software with parametric modeling, complex surfacing, animation, or rendering features.
- **OpenSCAD**: A script-based approach for parametric designs, favored by programmers.

B. From Model to 3D Print: The STL Format

When you finish designing in Tinkercad, you typically **export** to an `.STL` (Stereolithography) file:

- **Triangular Mesh Representation:** The 3D surface is approximated by many small triangles (facets).
- **Universal Acceptance:** Most 3D slicing software (Cura, PrusaSlicer) supports STL files for FDM/FFF printers.
- **Alternatives (OBJ, STEP, etc.):** OBJ can store color/texture, STEP is parametric. STL is the simplest universal "triangle mesh" standard.

B.1 Slicing Software

- **Cura, Simplify3D, PrusaSlicer**: Convert `.STL` into machine instructions (G-code) for your 3D printer.
- **Layer Heights, Speeds, Infills**: Adjust printing parameters (like 0.2 mm layer height, 20% infill, 50 mm/s speed) to balance quality vs. speed.
- **Support Structures**: Overhangs beyond 45° may need support material. The slicer can automatically generate these.

B.1.1 Post-Processing

- **Removing Supports**: Break or dissolve (for water-soluble filaments) after printing.
- **Sanding, Painting**: Depending on the desired finish or functional requirements.

C. Key Design Considerations for 3D Printing

FDM/FFF Printers extrude molten plastic (usually PLA or ABS). This imposes some constraints on design:

- **Overhangs and Bridges**: Parts of the model that extend too far horizontally without support can sag.
- **Wall Thickness**: Thin walls (< 0.8 mm) may be weak or unprintable.
- **Tolerance/Fit**: Allow clearance if you plan to insert or assemble multiple parts (e.g., 0.2–0.4 mm gap).

C.1 Material Choice

- **PLA (Polylactic Acid)**: Easiest to print, low warping, biodegradable, good for prototypes or aesthetic parts.
- **ABS (Acrylonitrile Butadiene Styrene)**: More heat-resistant, stronger, but warps more and can emit fumes.
- **PETG, Nylon, TPU**: Other options for flexibility, chemical resistance, or advanced mechanical properties.

D. Combining Shapes and Boolean Operations

In Tinkercad (and many CAD tools), you often use:

- **Union (Add)**: Merge multiple solids into one shape.
- **Subtract (Hole in Tinkercad)**: A shape set as a "Hole" can cut out from another shape, forming slots or negative space.
- **Intersect**: Only the overlapping region is kept. (Not always exposed in Tinkercad's basic interface, but conceptually similar.)

D.1 Structuring Complex Designs

- **Group and Rename**: Label your shapes ("Main box," "Sensor slot," "Cable hole"). Tinkercad allows grouping them into one entity while maintaining sub-object names for clarity.
- **Iterative Approach**: Start with a large bounding shape, subtract holes for sensors, add compartments for boards, etc.

E. Practical Tips for Tinkercad Efficiency

- **Use Align Tool**: Perfectly center or align shapes by selecting multiple objects and clicking "Align."
- **Custom Grids**: Adjust snap grid increments (e.g., 0.1 mm) for fine movements, or keep it at 1 mm for quick large shifts.
- **Duplicate and Repeat**: If you need multiple identical holes or features spaced evenly, Tinkercad can replicate them with consistent offsets.

E.1 File Management

- **Naming Conventions**: Keep version numbers (e.g., "EnclosureV2") to track changes.
- **Backup/Export**: Download the Tinkercad .STL or the .OBJ frequently, especially before major design changes.

F. Example Use Cases

- **IoT Sensor Enclosures**: Custom housings with holes for cables, displays, or vents.
- **Robotic Parts**: Wheels, brackets, sensor mounts, or gear assemblies.

- **Artistic Models**: Decorative shapes, personalized keychains, or figurines.
- **Functional Prototypes**: Phone stands, Raspberry Pi cases, or mechanical jigs.

F.1 Scaling Up and Collaboration

- **Parametric Modeling**: In advanced CAD, a single parameter (like board width) can auto-update the entire design's dimensions.
- **Team Projects**: Tinkercad's share link allows multiple users to access or copy a design, though not simultaneously co-edit in real time.

G. Best Practices for Printing Success

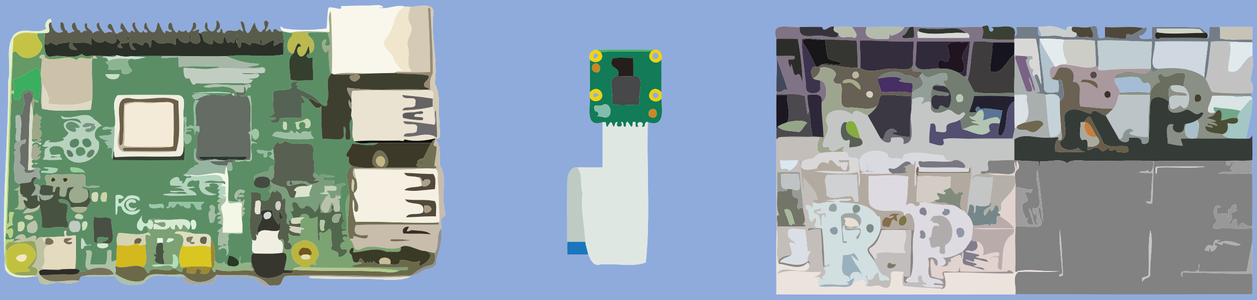
- **Design for Minimal Supports**: Orient your model so less support material is needed; consider bridging strategies.
- **Rounded Corners**: Sharp internal corners can stress crack; small fillets or chamfers help.
- **Draft Angles**: Slight angles can help with part removal from the print bed, especially for large flat surfaces.
- **Monitor First Layers**: The first layer is crucial for bed adhesion—smooth lines indicate a correct Z offset and temperature setting.

H. Conclusion and Future Explorations

Tinkercad provides a straightforward introduction to 3D object design:

- **Concept to Reality**: Quickly prototype an idea visually, then export to `.STL` for printing.
- **Iterative Refinements**: Inspect your 3D-printed part, measure real-world fit, and revise design as needed.
- **Path to Advanced CAD**: For complex assemblies or parametric constraints, consider learning tools like Fusion 360 or FreeCAD.

Final Note.
Mastering the basics of 3D modeling in Tinkercad opens up a world of possibilities—from simple project enclosures to creative art pieces and engineering prototypes. By combining shape manipulation, Boolean operations, and an understanding of slicer settings, you can produce functional, precise, and visually appealing 3D prints, bringing your digital designs into tangible, real-world objects.



16. Getting Started with Raspberry Pi Camera

Objectives

- Learn how to connect a camera to the Raspberry Pi
- Learn how to view camera output using Python
- Learn how to save images and videos from the command line
- Learn how to capture images and videos in Python
- Learn how to apply basic camera effects (brightness, contrast, image_effect, etc.)

Lab Plan

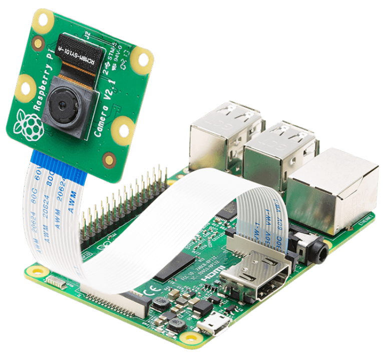


Figure 16.1: Raspberry Pi Camera connected to a Raspberry Pi

In this lab, you will attach the Raspberry Pi Camera Module, enable it in the Pi's configuration, then practice capturing still images and videos both from the command line (`raspistill` and `raspivid`) and from Python using the `picamera` library. Finally, you'll experiment with basic camera configurations and image effects.

Required Hardware Components

- **SD Card** with Raspberry Pi OS
- **Raspberry Pi Camera v2**
- **Display and HDMI cable**
- **Keyboard and mouse**
- **Power supply**

A. Connecting & Enabling the Camera

Attach the Camera Module

1. **Power Off the Pi.**
   Before attaching the camera, *shut down and disconnect* the Raspberry Pi from power. Locate the **Camera Module** port (often labeled "CAMERA" or "CSI") on the Pi board (Figure 16.2).
2. **Lift the Port's Plastic Clip.**
   Gently pull up on the port's clip to unlock it. Insert the camera ribbon cable with the metal contacts *facing* the metal contacts of the Pi port. Push the clip back down to secure it.
3. **Ensure Proper Orientation.**
   On a standard Raspberry Pi (not the Zero series), the ribbon cable usually faces toward the HDMI ports, so the blue backing of the ribbon faces toward the Ethernet/USB ports.

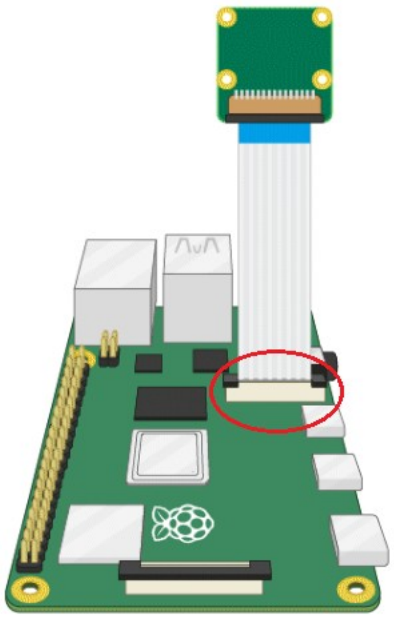


Figure 16.2: Raspberry Pi Camera Connection

Enable the Camera in Configuration

4. **Boot the Pi & Open Raspberry Pi Configuration.**
   After you've reconnected the Pi's power and it has finished booting to the desktop:
   - Click the **Raspberry** icon (top-left).
   - Go to **Preferences** → **Raspberry Pi Configuration**, as shown in Figure 16.3.
5. **Interfaces Tab.**
   In the configuration window:

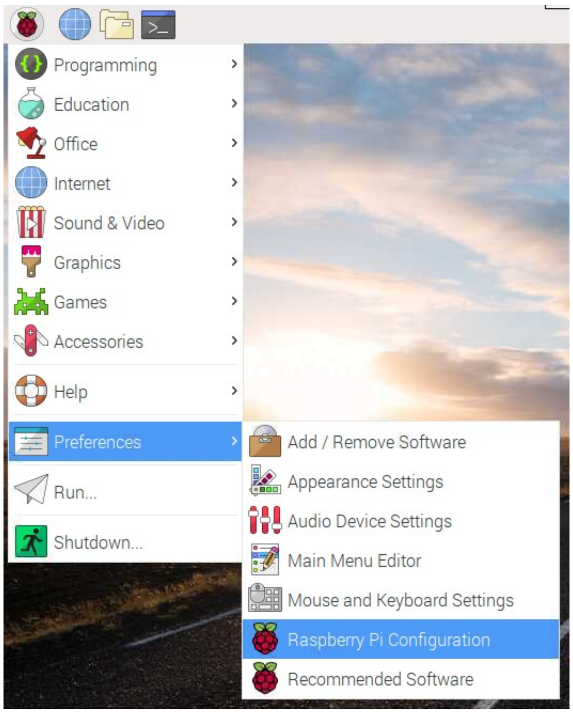
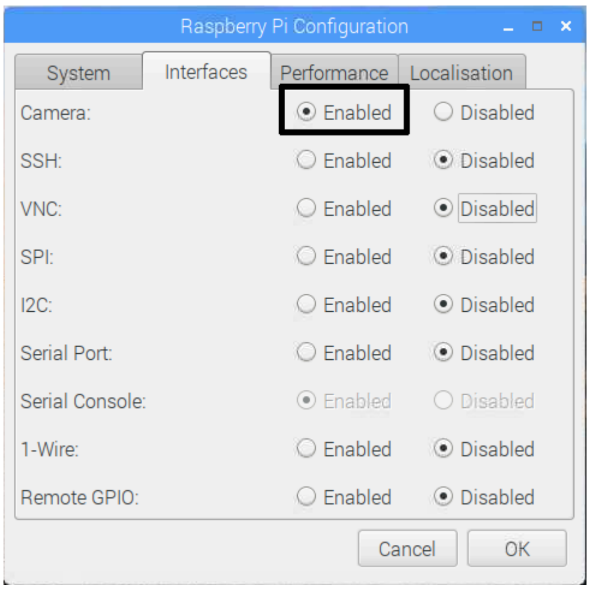


Figure 16.3: Raspberry Pi Configuration Menu

- Click the **Interfaces** tab.
- Set "**Camera**" to **Enabled**, as shown in Figure 16.4.
- **Reboot** if prompted.



Raspberry Pi Configuration

System | Interfaces | Performance | Localisation

Camera: [✓ Enabled] Disabled
SSH: Enabled [✓ Disabled]
VNC: Enabled [✓ Disabled]
SPI: Enabled [✓ Disabled]
I2C: Enabled [✓ Disabled]
Serial Port: Enabled [✓ Disabled]
Serial Console: [✓ Enabled] Disabled
1-Wire: Enabled [✓ Disabled]
Remote GPIO: Enabled [✓ Disabled]

Cancel OK

Figure 16.4: Raspberry Pi Interface Configuration

— Why Enable the Camera Interface?. By default, the Pi's camera interface is disabled for security and performance reasons. Turning it on allows the system to load the drivers needed to communicate with the CSI port and the camera module. Without this step, camera commands (like `libcamera` or older `raspistill/raspivid`) won't detect the module.

B. Capturing Images & Video (Command Line)

1. `raspistill` **for Photos.**
   Open a terminal on your Raspberry Pi and type:

```
raspistill -o Desktop/image.jpg
```

   This launches a 5-second camera preview (on your Pi-connected monitor) before capturing a still image to `Desktop/image.jpg`. The default resolution is the camera's maximum still resolution.

   Tips:
   - **Change preview time**: Add `-t <milliseconds>` to extend or shorten the preview duration, e.g., `raspistill -t 2000 -o pic.jpg` for a 2-second preview.
   - **Suppress preview**: Use `-nopreview` if you want to take an immediate photo without the live window.

2. **Custom Resolutions.**
   You can specify **width** (`-w`) and **height** (`-h`) to capture smaller or specific image sizes:

```
raspistill -o Desktop/image-small.jpg -w 640 -h 480
```

   This captures a 640×480 resolution image. Additional parameters (brightness, contrast, rotation, etc.) can be explored by typing `raspistill -help` or reading the manual page (`man raspistill`).

3. `raspivid` **for Video.**
   Recording a video uses a similar command:

```
raspivid -o Desktop/video.h264
```

   By default, this records until you press `Ctrl + C`. You'll see a live camera preview on the Pi's monitor for the duration. To limit recording time, add:

```
raspivid -o Desktop/video.h264 -t 10000
```

   which captures 10 seconds (10,000 ms).

   — Why `.h264`**?.** The `raspivid` command encodes raw camera data into H.264 video format. You can later convert or re-wrap it to `.mp4` if needed, but most modern media players (like VLC) handle `.h264` files directly.

4. **Playback.**
   After recording finishes:
   - Double-click the resulting `.h264` file in the Pi's file manager.
   - It should open in **VLC Media Player** (if installed) or another capable media player.
   - If you prefer a more common container, use tools like `ffmpeg` or `MP4Box` to convert:

```
MP4Box -add video.h264 video.mp4
```

   - Now `video.mp4` is playable in virtually any standard media player.

C. Capturing with Python (`picamera`)

5. **Install/Check `picamera`.**
   The `picamera` library typically comes pre-installed on Raspberry Pi OS (for models and OS images predating the new camera stack). If you need to install or update it:

```
sudo apt-get update
sudo apt-get install python3-picamera

```

   - If you're using the newer **libcamera**-based stack (e.g., on Raspberry Pi OS Bullseye or later), you may need to switch to libraries such as `picamera2`.
   - For older setups (Buster or prior), `python3-picamera` works seamlessly with the legacy camera firmware.

   Tips:
   - Type `python3` in a terminal, then `import picamera` to check if it's already installed.
   - If `import` fails, use the above commands or see documentation on transitioning to `picamera2` if your OS uses the newer camera stack.

6. **Simple Preview.**
   Launch **Thonny** or any Python 3 editor. Create a new file called `camera.py` and paste:

```
from picamera import PiCamera
from time import sleep

camera = PiCamera()
camera.start_preview()
sleep(5)
camera.stop_preview()

```

   Run the script:
   - You should see a 5-second camera preview on the Pi's connected monitor (not visible over SSH or VNC by default).
   - This confirms the camera is detected and the `picamera` library can control it.

7. **Capturing an Image.**
   Next, update the code to actually take a picture:

```
camera.start_preview()
sleep(5)
camera.capture('/home/pi/Desktop/image.jpg')
camera.stop_preview()

```

   - The `sleep(5)` allows the camera to auto-adjust for brightness/white balance.
   - After 5 seconds, it saves `image.jpg` to the Desktop.
   - Check your Desktop folder to see the captured image.

8. **Multiple Images in a Loop.**
   You can capture a sequence of images by looping:

```
camera.start_preview()
for i in range(5):
    sleep(5)
    camera.capture(f'/home/pi/Desktop/image{i}.jpg')
camera.stop_preview()

```

- This takes `image0.jpg`, `image1.jpg`, up to `image4.jpg`, each 5 seconds apart.
- Adjust the loop count or sleep duration to suit your needs.

Recording Video with Python

9. **Video Example.**
You can also record video via Python:

```
camera.start_preview()
camera.start_recording('/home/pi/Desktop/video.h264')
sleep(5)
camera.stop_recording()
camera.stop_preview()

```

- The camera will display a preview for 5 seconds while it records to `video.h264`.
- Play back the video using VLC or convert it to `.mp4` with a tool like `MP4Box -add video.h264 video.mp4`.

Tips:
- Add additional parameters (like `camera.resolution`, `camera.framerate`, or `camera.annotate_text`) to customize your recordings.
- If the preview doesn't appear while running headless (SSH/VNC), either connect an HDMI monitor or skip the preview steps.

D. Adjusting Camera Settings & Image Effects

Note: Some settings only affect the *preview*, others affect *captured images*, and some affect both. Experiment to see how each behaves in your environment.

Rotation and Transparency

- **Rotation**:

```
camera.rotation = 180

```

Valid values for rotation are `0`, `90`, `180`, or `270`. Use this if you mounted the camera upside-down or need another orientation.

- **Alpha Transparency for Preview**:

```
camera.start_preview(alpha=200)

```

The `alpha` parameter can be from `0` (fully transparent) to `255` (fully opaque). A lower alpha value makes the preview see-through, useful if you want to overlay camera feed onto the Pi's desktop or GUI.

Brightness and Contrast

- **Brightness** ranges from `0` (dark) to `100` (bright). Default is `50`:

```
camera.start_preview()
camera.brightness = 70
sleep(5)
camera.stop_preview()

```

- **Contrast** also ranges from 0 (lowest contrast) to 100 (highest). Default is 50:

```
for i in range(100):
    camera.annotate_text = "Contrast: %s" % i
    camera.contrast = i
    sleep(0.1)

```

  Cycling through contrast values in a loop lets you see how the preview changes in real time.

Annotating Text

- **Adding text**:

```
camera.annotate_text = "Hello world!"

```

  This overlay text appears on both the *preview* and the *captured image*.

- **Text Size** (from 6 to 160):

```
camera.annotate_text_size = 50

```

  Adjust to make text smaller or larger. Default is 32.

- **Colors**:

```
from picamera import PiCamera, Color

camera.annotate_background = Color('blue')
camera.annotate_foreground = Color('yellow')

```

  Here, you set the text color (`foreground`) and background behind the text. Common color names include `'blue'`, `'red'`, `'green'`, or you can specify RGBA like `Color(r=255,g=255,b=0,a=128)`.

Image Effects

- `camera.image_effect` can be:

```
    none, negative, solarize, sketch, denoise, emboss, oilpaint,
    hatch, gpen, pastel, watercolor, film, blur, saturation,
    colorswap, washedout, posterise, colorpoint, colorbalance,
    cartoon, deinterlace1, deinterlace2
```

- Example:

```
camera.image_effect = 'colorswap'

```

  This inverts or swaps color channels, creating a fun or surreal look.

- **Loop over all effects**:

```
for effect in camera.IMAGE_EFFECTS:
    camera.image_effect = effect
    camera.annotate_text = "Effect: %s" % effect
    sleep(5)

```

  Each effect applies for five seconds, letting you preview them one by one.

Exposure Mode & White Balance

- **Exposure Mode** (`camera.exposure_mode`):

  ```
  off, auto, night, nightpreview, backlight, spotlight,
  sports, snow, beach, verylong, fixedfps, antishake, fireworks
  ```

  The default is `auto`.
- **White Balance** (`camera.awb_mode`):

  ```
  off, auto, sunlight, cloudy, shade, tungsten,
  fluorescent, incandescent, flash, horizon
  ```

  The default is `auto`.
- Example:

```
camera.awb_mode = 'sunlight'
camera.exposure_mode = 'beach'

```

These settings can improve image clarity under specific lighting or scenic conditions (e.g., tungsten lighting, sunny day, or dimly lit environments).

— Experimenting with Settings. Through trial and error, you can discover which combinations of `brightness`, `contrast`, `exposure_mode`, and `image_effect` best suit your project's look and environment. For example:
- In low light, using `camera.exposure_mode = 'night'` might help gather more brightness.
- For a stylized effect, combine `camera.image_effect = 'cartoon'` with `camera.annotate_text_size = 60` for a fun, dramatic preview.

Always remember that some settings only apply to the preview or the captured result, so check the official `picamera` docs for details.

Measuring Success

- **Camera Attached & Enabled**: Your Raspberry Pi recognizes the camera module; using `raspistill` and `raspivid` commands works without errors or "camera not detected" messages.
- **Basic Captures from Command Line**: You can run:

  ```
  raspistill -o image.jpg
  raspivid -o video.h264

  ```

  and obtain still images (`.jpg`) and video clips (`.h264`). Opening them in VLC or another player confirms the capture was successful.
- **Python Preview/Capture**:
  - The `picamera` library correctly initializes the camera; a 5-second preview appears on the Pi's monitor when you run a Python script.
  - Calling `camera.capture()` or `camera.start_recording()` generates valid files, demonstrating that the Python environment, drivers, and camera interface all work together.
- **Simple Effects Tested**: You've attempted at least one effect (e.g., `colorswap`, `oilpaint`, or `negative`), or adjusted brightness/contrast to see how it impacts the preview or final

capture. Observing the preview helps confirm that the chosen settings are taking effect. ■

Optional Scenario (Photo Booth)
Transform your Raspberry Pi + camera into a mini *photo booth* experience:

- **Timed Captures with Effects:**
  - Write a Python script that loops every 10 seconds (or your preferred interval) to capture an image.
  - On each iteration, set `camera.image_effect` to a different style (like "`negative`," "`colorswap`," or "`cartoon`").
  - You'll end up with a series of photos, each sporting a unique effect.
- **Fun Messages and Overlays:**
  - Use `camera.annotate_text` to display silly phrases, timestamps, or event names over each snapshot.
  - Optional: Adjust `camera.annotate_text_size` for bigger, bolder lettering.
  - If you have an RGB backlight LCD or LED strip, you could also flash colorful lights in sync with captures.
- **Interactive Trigger (Button or Web Interface):**
  - Connect a physical button to the Pi's GPIO pin. When pressed, the script captures an image or records a short video.
  - Or, run a lightweight web server (using `Flask` or `django`), so visitors tap a webpage on a phone/tablet to trigger the capture.
  - Provide real-time feedback: blink an LED or show a "*Say Cheese!*" message on the Pi's monitor before taking the shot.
- **Experiment with Exposure/White Balance:**
  - If your booth area is brightly lit, set `camera.exposure_mode = 'sports'` for faster captures with less motion blur.
  - For a dimly lit environment, `camera.exposure_mode = 'night'` or `camera.awb_mode = 'tungsten'` might help compensate.
  - Check each setting until you find a fun or artistic style that suits your booth theme.

By blending timed captures, annotated overlays, and an interactive trigger, you can create a lively *photo booth* that's both entertaining and easy to operate. Whether at a party, an event, or a simple home project, the Raspberry Pi camera's flexibility allows for endless creative possibilities.

Theory Deep Dive: Underlying Principles and Concepts

This section explores the fundamental concepts behind the Raspberry Pi Camera—how it interfaces with the Pi, how camera modules capture and process images, and how advanced features like image effects and exposure modes can be leveraged to create more sophisticated applications. Understanding these foundations will help you optimize camera usage in projects ranging from simple photo booths to advanced computer vision.

A. Raspberry Pi Camera Module Overview

Raspberry Pi Camera Modules typically connect via a specialized ribbon cable to the camera port on the Pi:

- **CSI (Camera Serial Interface)**: A high-speed interface for image data transfer directly from the camera to the Pi's GPU/ISP (Image Signal Processor).

- **Fixed-Focus Lens**: Most Pi cameras come with a small, fixed-focus lens. Some advanced versions offer adjustable or higher-resolution sensors.
- **Infrared Variants (NoIR)**: Versions without an IR filter, suitable for low-light or night-vision scenarios (often used with IR illuminators).

A.1 Hardware Accelerated Image Processing

The Pi's SoC (System on Chip) includes a GPU-based ISP that handles:

- **Debayering**: Converting raw sensor data (Bayer pattern) into a color image.
- **Lens Shading Correction**: Compensating for brightness falloff toward the edges of the frame.
- **Noise Reduction, Sharpening, Denoising**: Enhancing captured images in real time.

This hardware acceleration allows the Pi to record HD video or capture stills without overburdening the CPU.

B. Capturing Images and Video

Two main approaches exist for capturing images/videos on the Pi:

- **Command-Line Tools**: `raspistill` (photos) and `raspivid` (videos). These directly call the GPU's camera functions.
- **Python Scripts (picamera)**: A Python library that offers a flexible API, enabling preview windows, annotation, effects, and dynamic adjustments at runtime.

B.1 The Preview Window Nuance

- **Direct HDMI Display**: If you have a monitor connected to the Pi's HDMI, you'll see a native camera preview overlay (hardware accelerated).
- **SSH/VNC Limitations**: Remote sessions typically do not display the camera preview window or require special configurations (like X11 forwarding or using the *legacy camera stack*).

C. Camera Parameters and Effects

The Pi camera's internal processing can be tuned via command-line flags or Python properties. Common parameters include:

- **Resolution, FPS, Bitrate**: For videos or still images. E.g., 640x480 vs. 1920x1080.
- **Brightness, Contrast, Saturation**: Adjust image aesthetics.
- **Exposure Modes**: `auto, night, backlight`, etc., for different lighting scenarios.
- **White Balance (AWB) Modes**: `auto, sunlight, cloudy, tungsten`, etc. Helps correct color casts.
- **Image Effects**: `negative, watercolor, colorpoint`, etc. Interesting visual transformations.

C.1 Use Cases for Effects

- **Colorpoint** or **Colorswap**: Creating artistic or stylized images for a photo booth.
- **Negative**: Inverting colors for a unique, high-contrast feed.
- **Cartoon or Sketch**: Fun illusions or quick previews for a kid-friendly interface.

D. Image Data Handling and Storage

When capturing media:

- **JPEG vs. RAW**: By default, still images are stored as `.jpg`. Some advanced usage can output RAW sensor data alongside JPEG if needed.
- **H.264 Videos**: The `.h264` container can be re-wrapped in MP4 for broader compatibility, e.g.:

```
MP4Box -add video.h264 video.mp4

```

- **File Sizes**: High-resolution images or extended video recordings can consume significant disk space on the SD card. Monitor free space carefully.

D.1 Performance Considerations

- **Thermal Throttling**: Long video captures or high ISO in low light can generate heat. The Pi may throttle CPU/GPU if overheated.
- **Simultaneous Processes**: Running CPU-intensive tasks (like image processing in Python) while recording video could impact frame rate or cause dropped frames.

E. Additional Tools and Libraries

Beyond `raspistill`, `raspivid`, and `picamera`, other noteworthy tools include:

- **libcamera**: A newer camera stack (in progress) that replaces the older `raspivid/still` commands on some Pi OS releases.
- **OpenCV**: Powerful computer vision library. Integrates with `picamera` or `libcamera` for real-time image analysis (object detection, face recognition).
- **FFmpeg**: If installed, can transcode or record from the camera for advanced streaming or compression workflows.

E.1 Real-time Processing & AI

- **Edge AI**: The Pi can run small neural networks for object detection (e.g., YOLO tiny) on camera frames in real time, though performance varies.
- **GPU Acceleration**: Some ML frameworks can offload certain tasks to the Pi GPU or use specialized hardware like the Google Coral USB accelerator.

F. Example Projects

- **Time-Lapse Camera**: Use Python loops to capture images at set intervals, then compile them into a time-lapse video.
- **Surveillance System**: Automate `raspivid` with motion detection (via external PIR sensor or software) to record only when movement is detected.
- **Interactive Art Installations**: Real-time color effect changes when certain sensors (light, proximity) trigger events.
- **Remote Monitoring**: Stream camera feed to a website or local network using `MJPG-streamer` or custom Python scripts.

F.1 Recording or Streaming Over Networks

- **Use Cases**: Broadcasting Pi camera feed to an IP address for remote viewing. Great for security cams or robotics.

- **Bandwidth Constraints**: HD streaming may require a reliable network. Wireless connections might cause lower frame rates.

G. Best Practices and Troubleshooting

- **Lens Care**: Keep the camera lens clean and free from dust or fingerprints.
- **Check for Cable Issues**: If the Pi can't detect the camera, ensure the ribbon cable is seated properly, metallic contacts facing the right direction.
- **Lighting Conditions**: For best results, ensure adequate light or increase exposure/ISO carefully to avoid grainy images.
- **Avoid Overheating**: The camera module and Pi can get warm during continuous video capture. Consider small heatsinks or ventilation if doing extended recordings.

G.1 Debug Steps

- **vcgencmd get_camera**: Check if the system detects the camera (output: `supported=1 detected=1`).
- **Device Tree Settings**: On newer Pi OS versions, ensure `dtoverlay=vc4-kms-v3d` or `dtoverlay=vc4-fkms-v3d` is configured properly in `/boot/config.txt`.

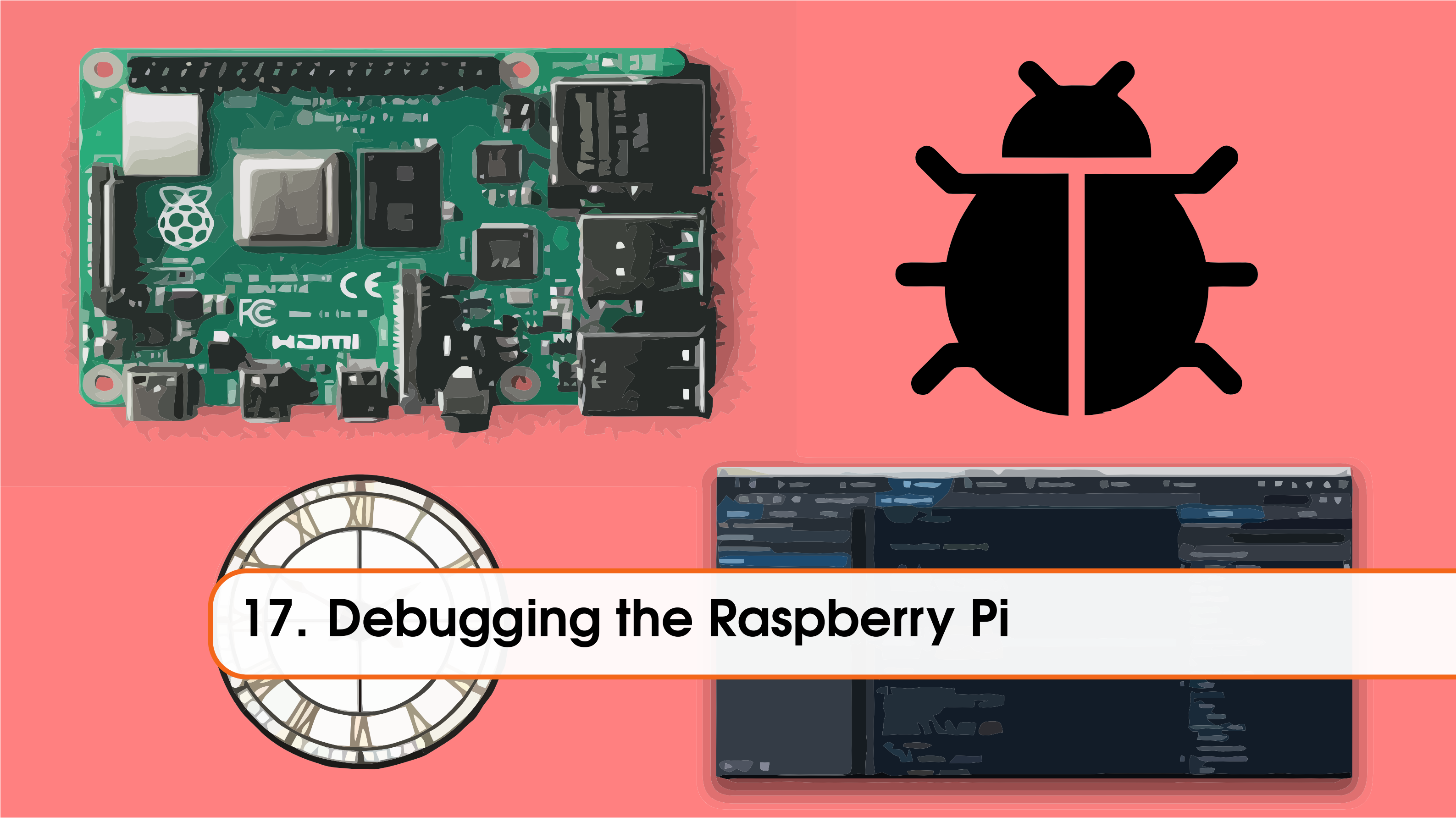
H. Conclusion and Future Explorations

The **Raspberry Pi Camera Module** is a versatile tool for capturing images and videos in embedded, educational, and hobbyist contexts:

- **Hardware-Accelerated** capturing ensures smooth HD video even on a modest Pi.
- **Flexible Software Ecosystem**: Command-line tools (`raspistill`, `raspivid`) and Python libraries (`picamera`, `libcamera`) cover simple snapshots to complex real-time processing.
- **Broad Applications**: Time-lapse photography, surveillance, computer vision, photo-booths, interactive media.

Final Note.

Once comfortable with the fundamentals—connecting the camera, taking stills and videos, adjusting brightness or effects—you can expand into advanced areas: streaming footage over networks, integrating the Pi camera feed with **OpenCV** for machine vision, or creating sophisticated GUI-based camera control interfaces. The Raspberry Pi camera stack continues to evolve (especially with `libcamera`), ensuring a rich environment for creative experimentation and real-world imaging solutions.



17. Debugging the Raspberry Pi

Objectives

- Learn how to set the correct time and date on a Raspberry Pi (which lacks a built-in RTC).
- Learn how to keep packages up to date to avoid errors caused by outdated software.

Lab Plan

In this lab/tutorial, we will:

1. Verify or configure the system time and date (both manually and using `raspi-config`).
2. Check or change the time zone.
3. Update the Raspberry Pi's packages to fix common dependency or compatibility problems.

Required Hardware Components

- **Raspberry Pi** (any model, e.g., Raspberry Pi 4)
- **SD Card with Raspberry Pi OS** (preferably up to date)
- **Display and HDMI cable** (for visual feedback)
- **Keyboard and mouse** (USB)
- **Power supply** (5V, 2.5–3A recommended)

Practical Steps

1. **Check If Your Time/Date Are Correct**
   Many tasks on the Raspberry Pi (e.g., SSL certificate checks, log timestamps) rely on accurate system time. By default, a Pi with internet access uses NTP (Network Time Protocol). If your Pi is showing the wrong date/time or you're offline, you'll need to configure it manually.
2. **Verify Network Time Synchronization (Optional)**
   If your Pi has internet access, it typically auto-syncs the clock. Confirm by:

```
sudo systemctl status systemd-timesyncd
```

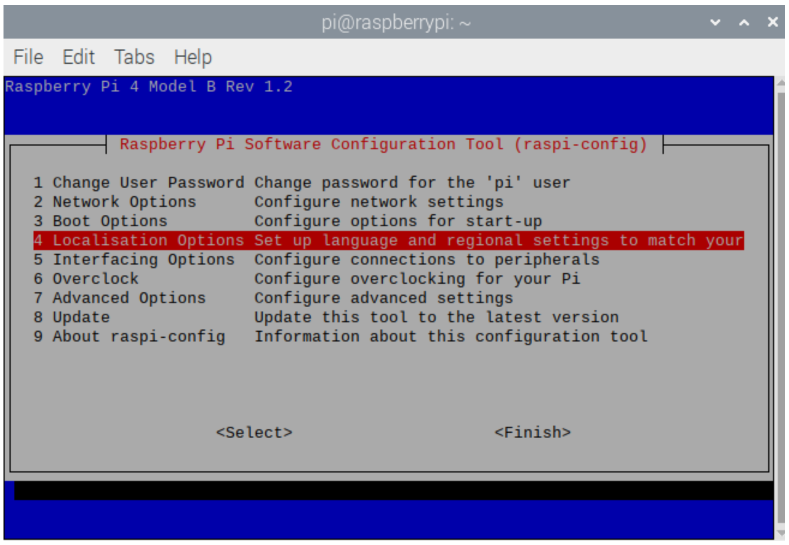
If the service is active (running), your Pi's clock should update automatically via NTP.

3. **Manually Set the Time/Date**
If you're offline or the time is far off, run:

```
sudo date -s "19/09/2020 11:00"
```

Replace the quoted string with your current date/time. This is particularly useful if the Pi has been powered off without an RTC module.

4. **Configure Time Zone via `raspi-config`**
Even if the date is correct, the time zone might be wrong. Fix it by:
   (a) `sudo raspi-config`
   (b) Select **Localization Options** → **Change Time Zone**.
   (c) Pick your region and city.



```
Raspberry Pi 4 Model B Rev 1.2

        Raspberry Pi Software Configuration Tool (raspi-config)

 1 Change User Password Change password for the 'pi' user
 2 Network Options      Configure network settings
 3 Boot Options         Configure options for start-up
 4 Localisation Options Set up language and regional settings to match your
 5 Interfacing Options  Configure connections to peripherals
 6 Overclock            Configure overclocking for your Pi
 7 Advanced Options     Configure advanced settings
 8 Update               Update this tool to the latest version
 9 About raspi-config   Information about this configuration tool

                    <Select>                     <Finish>
```

Figure 17.1: Selecting a time zone from `raspi-config`.

5. **Refresh Your System's Packages**
Outdated packages are a leading cause of Pi errors. Updating applies security patches, bug fixes, and newer versions. However, if your project depends on a specific package version, take caution with a full upgrade.

6. **Run `apt update`**

```
sudo apt update
```

This fetches new package indexes from repositories, telling the Pi what updates are available. It doesn't install anything yet.

7. **Upgrade Packages with `apt full-upgrade`**

```
sudo apt full-upgrade
```

`full-upgrade` not only installs new versions but also removes packages that create conflicts. If you want to avoid removing anything, try:

```
sudo apt upgrade
```

But some updates may fail if they require removing old dependencies.

8. **Reboot After Upgrades**
   Kernel or system-level updates require a restart:

```
sudo reboot
```

   This ensures all changes take effect.

9. **(Optional) Remove Unused Packages**

```
sudo apt autoremove
```

   Cleans up leftover packages. This can free storage space on smaller SD cards.

   Warning: Close all running applications **before** upgrading. Interrupting an `apt full-upgrade` could leave your system with partially installed packages and cause serious issues.

   Know the Difference:
   - `apt update`: Refreshes the list of available packages (metadata).
   - `apt upgrade`: Upgrades currently installed packages without removing anything.
   - `apt full-upgrade`: Can install *and remove* packages to resolve version conflicts or dependencies.

Measuring Success

- **Accurate System Clock**: Run `date` and confirm correct time and date. The time zone matches your region, so log timestamps and SSL checks function properly.
- **Update Commands Run Smoothly**: `sudo apt update` and `sudo apt full-upgrade` complete without broken dependencies or error messages. No warnings appear on reboot.
- **Optional RTC Module**: If you've added an external hardware RTC, the Pi retains correct time even without internet. No major drift or resets on power cycles.
- **Stable Operation**: Post-upgrade, the Pi reboots cleanly, with no "partial upgrade" issues. You can confirm new software versions with `apt show <package>` or `dpkg -l`.

Theory Deep Dive: Underlying Principles and Concepts

This optional section dives deeper into the reasoning behind the Pi's timekeeping approach and how package managers like `apt` handle upgrades.

A. Real-Time Clock (RTC) vs. NTP

A typical PC has a battery-backed RTC chip on the motherboard. The Pi omits this for cost and space reasons, relying on:

- **NTP** for automatic clock sync if online.
- **Manual date/time input** or an external RTC if offline.

If your project demands an accurate clock offline (e.g., data logging in remote areas), consider an add-on RTC module.

B. Understanding apt

APT (Advanced Package Tool) is the main package manager on Debian-based systems (including Raspberry Pi OS):

- **apt update** fetches the newest package "lists" or metadata from repositories.
- **apt upgrade** updates existing packages if possible without removing anything.
- **apt full-upgrade** resolves conflicts by removing or installing additional packages as needed.

This system ensures you get official packages compiled for the Pi's ARM architecture.

C. Minimizing Upgrade Risks

- **Backup critical files** before major upgrades, especially if you run custom software.
- **Keep a stable environment** if a specialized project depends on older versions (you might freeze or pin certain packages).
- **Test after each big upgrade** to confirm everything still works.

D. Why the Raspberry Pi Lacks a Built-In RTC

Unlike many PCs, the Pi does not have an on-board Real-Time Clock (RTC) chip with a battery. Thus:

- It reverts to a default time on each reboot if it can't reach an NTP server.
- If offline, manual setting is the only way to maintain a correct clock, unless you attach an external RTC module.

E. How Updates Fix Package Errors

- **Bug Fixes & Security Patches**: Newer versions patch vulnerabilities and performance issues.
- **Dependency Resolution**: Some libraries require updated core packages; failing to update can lead to "unmet dependencies."
- **Recommended Approach**: For most users, regularly updating ensures a stable, secure system. But specialized projects might pin certain package versions.

F. Additional Tips

- **Network Connectivity**: Confirm you have a reliable internet connection before `apt` operations.
- **Check Storage**: Use `df -h` to ensure you have enough free space. Large updates can fill smaller SD cards quickly.
- **Further Docs**: For advanced debugging or configuration topics, see the official Raspberry Pi documentation.