**Arm Compute Library (ACL)**: the most performant library available for ML on Arm. It’s deployed on billions of devices worldwide – from servers to flagship smartphones, to smart ovens.

The Arm Compute Library is currently the most performant library for ML on Arm, and it’s deployed on billions of devices worldwide – from servers to smartphones.

The Arduino Web Editor has a limit of 200 seconds of compilation time per day.

**Hardware devices and software tools covered in each chapter**: <https://github.com/PacktPublishing/TinyML-Cookbook>

You can download the example code files for this book from GitHub at https:// github.com/PacktPublishing/TinyML-Cookbook.

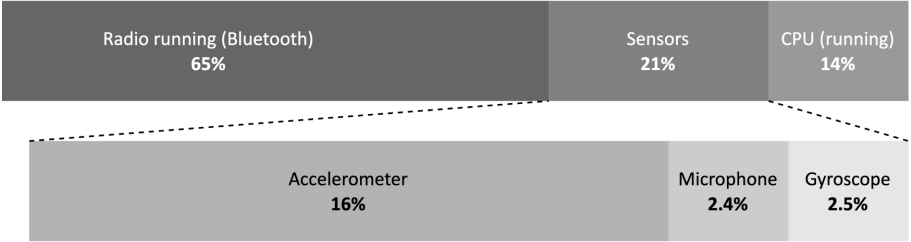
We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/ downloads/9781801814973\_ColorImages.pdf.

TinyMl is the set of technologies in ML and embedded systems to make use of smart applications on extremely low-power devices. Generally, these devices have limited memory and computational capabilities, but they can sense the physical environment through sensors and act based on the decision taken by Ml algorithms.

**Why run ML locally?**

Latency, Power consumption and Privacy.



As we can see from the power consumption breakdown, the CPU computation is more power-efficient than Bluetooth communication (14% over 65%).

**The opportunities and challenges for TinyML.**

TinyML finds its natural home wherever a power supply from the mains is impossible or complex to have, and the application must operate with a battery for as long as possible.

**Challenges.**

-Computational perspective: our devices are limited in memory and processing power.

- Deployment environment could be unfriendly. Environmental factors, such as dust and extreme weather conditions, could get in the way and influence the correct execution of our applications.

**Deployment Environments for TinyML.**

Centralized and distributed systems.

In a centralized system, the application does not necessarily require communication with other devices.

A typical example is keyword spotting. Nowadays, we interact with our smartphones, cameras, drones, and kitchen appliances seamlessly with our voices. The magic words Ok Google, Alexa, and so on that we use to wake up our smart assistants are a classic example of an ML model constantly running locally in the background. The application requires running on a low-power system without sending data to the cloud to be effective, instantly, and minimizing power consumption.

Usually, centralized TinyML applications aim to trigger more power-hungry functionalities and benefit from being

private by nature since they do not need to send any data to the cloud.

In a distributed system, the device (that is, the node or sensor node) still performs ML locally but also communicates with nearby devices or a host to achieve a common goal. (**Wireless sensor network**).

**DL**

It’s the specific class of ML that can perform complex classification tasks directly on raw images, text, or sound.

This technology makes voice-controlled virtual assistants, facial recognition systems, and autonomous driving possible, just to name a few.

**Deep neural networks.**

A deep neural network consists of several stacked layers aimed at learning patterns. Each layer contains several neurons, the fundamental compute elements for artificial neural networks (ANNs) inspired by the human brain.

However, neurons can only solve simple linear problems with linear transformations.

**Convolutional neural networks.**

Convolutional neural networks (CNNs) are specialized deep neural networks predominantly applied to visual recognition tasks.

We consider CNNs as the evolution of a regularized version of the classic fully connected neural networks with dense layers (that is, fully connected layers).

With the rise of CNNs, visual recognition tasks saw improvements thanks to convolution layers that make feature extraction part of the learning problem.

Based on the assumption that we are dealing with images, and inspired by biological processes in the animal visual cortex, the convolution layer borrows the widely adopted convolution operator from image processing to create a set of learnable features.

This approach brings two significant benefits:

* It extracts the relevant features automatically without human intervention.
* It reduces the number of input signals per neuron considerably.

When designing CNNs for visual recognition tasks, we usually place the fully connected layers at the network’s end to carry out the prediction stage. Since the output of the convolution layers is a set of images, typically, we adopt subsampling strategies to reduce the information propagated through the network and then reduce the risk of overfitting when feeding the fully connected layers.

One of the most critical aspects to consider when deploying DL networks for TinyMl is the model size, generally defined as the memory required for storing the weights.

Since out tiny platforms have limited physical memory, we require the model to be compact to fit the target device.

However, the memory constraint is not he only challenge we could encounter when deploying a model on microcontrollers. For example, although the trained model commonly employs arithmetic operations in floating-point precision, CPUs on microcontrollers could not have hardware acceleration for it.

Therefore, quantization is an indispensable technique to overcome the preceding limitations.

**Quantization.**

Quantization is the process of performing neural network computations in lower bit precision. The widely adopted technique for microcontrollers applies the quantization post-training and converts the 32-bit floating-point weight to 8-bit integer values. This technique brings a 4x model size reduction and a significant latency improvement with very little or no accuracy drop.

**Learning the difference between power and energy.**

**Voltage versus current.**

Curren is what makes an electric circuit work, which is the flow of electric charges across surface A conductor in a given time.

I = Q/t

I: Amperes, Q: coulombs (C) and t: time (s)

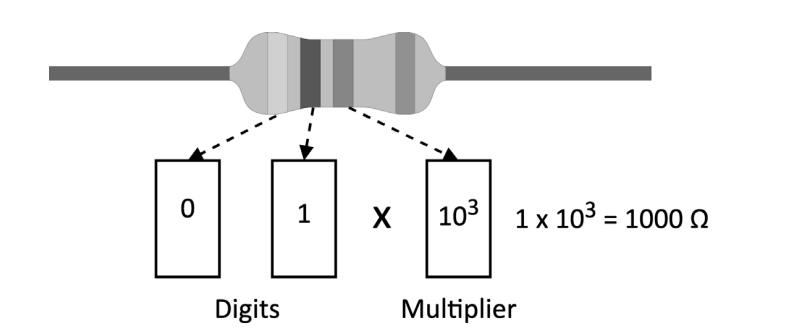
Voltage produces an electric field to allow the electric charge to flow in the circuit.

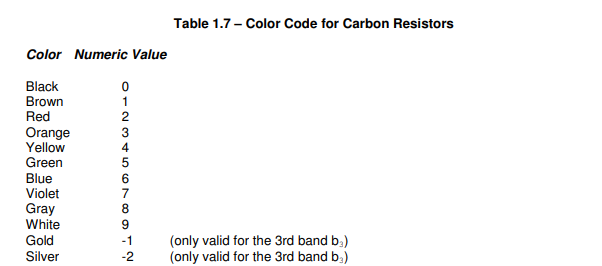
**Ohm’s law**: The current through a conductor is proportional to he voltage across a resistor.

I = V/R

A resistor is an electrical component used to reduce the current flow. This component has a resistance measured with Ohm (Ω) and identified with the letter R.

Standard resistors have four, five or six bands. The color on the bands denotes he resistance value.





To easily decode the color bands, we recommend using the online tool from Digi-Key (https://www.digikey.com/en/resources/conversion-calculators/ conversion-calculator-resistor-color-code).

**Power Versus Energy.**

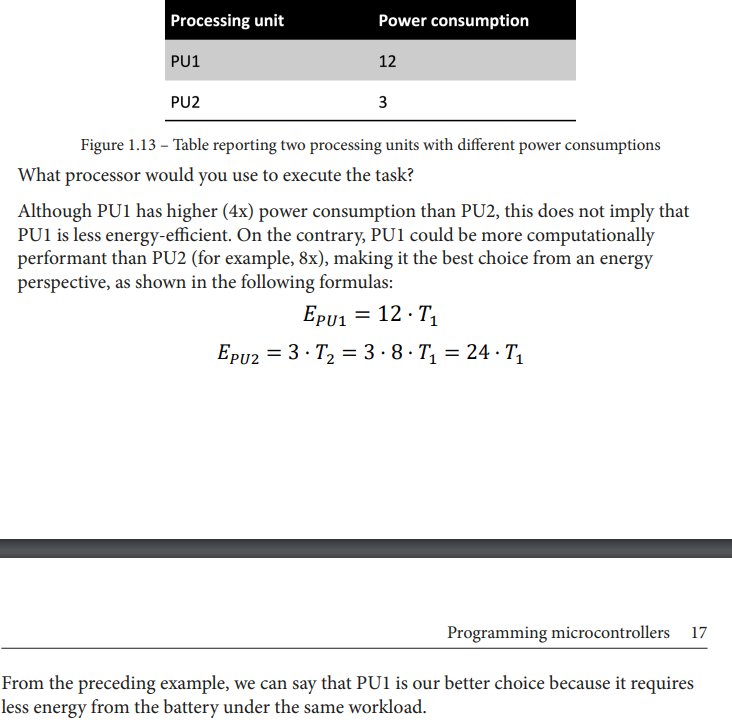
Energy is the capacity for doing work (for example, using force to move an object), while power is the rate of consuming energy.

In practical terms, power tells us how fast we drain the battery, so high power implies a faster battery discharge.

Power and energy are related to voltage and current through the following formulas:

P = V . I

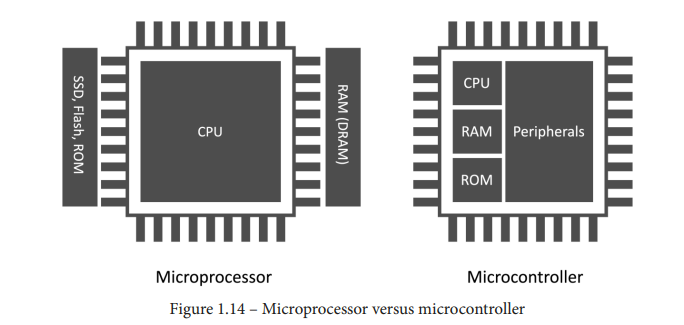
E = P . T



**Programming Microcontrollers.**

Unlike a computer, a microcontroller fits entirely on an integrated chip, and it has incredibly low power and low price.

**Microprocessor vs Microcontroller.**

****

Microprocessor tackles scenarios where the tasks are usually as follows:

1. Dynamic.
2. General-purpose.
3. Compute intensive.

A microcontroller addresses completely different scenarios, and in the following list, we shall highlight some of the critical ones:

* The tasks are single-purpose and repetitive:

Microcontrollers do not require strict re-programmability. Typically, the applications are less computationally intensive than the microprocessor ones and do not have frequent interactions with the user. However, they can interact with the environment or other devices.

As an example, you could consider a thermostat. The device only requires monitoring the temperature at regular intervals and communicating with the heating system.

* We could have time frame constrains:

Certain tasks must complete execution within a specific time frame. This requirement is the characteristic for real-time application (RTAs), where the violation of the time constraint may affect the quality of service (soft real time) or be hazardous (hard real time).

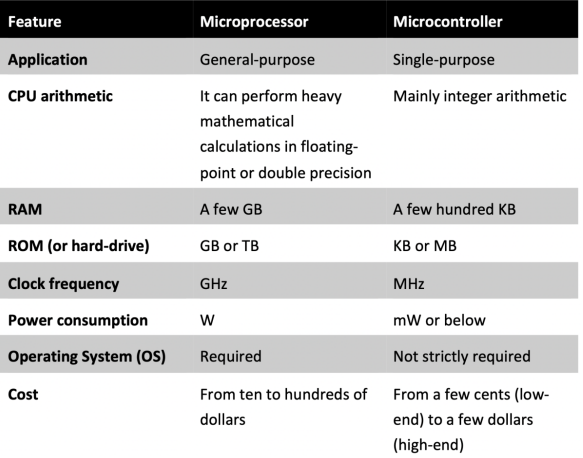
* Low Power constraints.

Applications could live in a battery-powered environment, so the microcontroller must be low-power to extend their lifetime.

NB: All the off-chip components generally reduce power efficiency as a rule of thumb. That is the main reason why microcontrollers integrate both the RAM and a kind of hard drive (ROM) within the chip.

* Physical size constrains:

The device could live in products that are small in size. Since the microcontroller is a computer within a chip, it is perfect for these scenarios. The package size for a microcontroller can vary but typically is in the range of a few square millimeters.



**Memory architecture.**

* **Program memory (ROM)**:

This is a non-volatile read-only memory reserved for the program to execute. Although its primary goal is to contain the program, it can also store constant data. Thus, program memory is similar to our everyday computer’s hard drive.

* **Data memory (RAM):**

This is volatile memory reserved to store/read temporary data. Since it is RAM, we lose its content when switching off the system.

Since program and data memory are functionally opposite, we usually employ different semiconductor technologies. In particular, we can find Flash technologies for the program memory and static-random-access memory (SRAM) for the data memory.

Flash memories are non-volatile and offer low power consumption but are generally slower than SRAM. However, given the cost advantage over SRAM, we can find larger program memory than data memory.

Now that we know the difference between program and data memory, where can we store the weights for our deep neural network model?

The answer to this question depends on whether the model has constant weights. If the weights are constant, so do not change during inference, it is more efficient to store them in program memory for the following reasons:

* Program memory has more capacity than SRAM.
* It reduces memory pressure on the SRAM since other functions require storing variables or chunks of memory at runtime.

**Peripherals.**

Peripherals are essential and can interface with sensors or other external components.

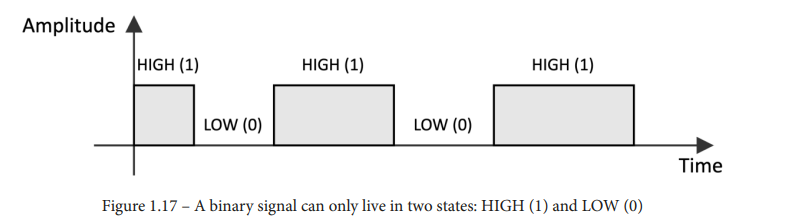
Each peripheral has a dedicated functionality, and it is assigned to a metal leg (pin) of the integrated circuit.

Hardware vendors typically number the pins anti-clockwise, starting from the top-left corner of the chip, marked with a dot for easy reference, as shown below:

Since peripherals can be various types, we can group them into four main categories for simplicity:

1. **General-purpose input/output (GPIO or IO)**

GPIOs do not have a predefined and fixed purpose. Their primary function is to provide or read binary signals that, by nature, can only live in two well-defined states: HIGH (1) or LOW (0). The following figure shows an example of a binary signal:



Typical GPIO usages are as follows:

* Turning on and off an LED.
* Detecting whether a button is pressed.
* Implementing complex digital interfaces/protocols such as VGA.

**Analog/digital converters.**

In TinyML, our applications will likely be dealing with time-varying physical electrical quantities, such as images, audio, and temperature.

Whatever these quantities are, the sensor transforms them into a continuous electrical signal interpretable by the microcontrollers. This electrical signal, which can be either a voltage or current, is commonly called an analog signal.

The microcontroller, in turn, needs to convert the analog and digital worlds.

An analog-to-digital converter (ADC) samples the analog signal at fixed intervals times and converts the electrical signal into a digital format.

A digital-to-analog converter (DAC) performs the opposite functionality: converting the internal digital format into an analog signal.

**Serial communication.**

Communication peripherals integrate standard communication protocols to control external components. Typical serial communication peripherals available in microcontrollers are I2C, SPI, UART, and USB.

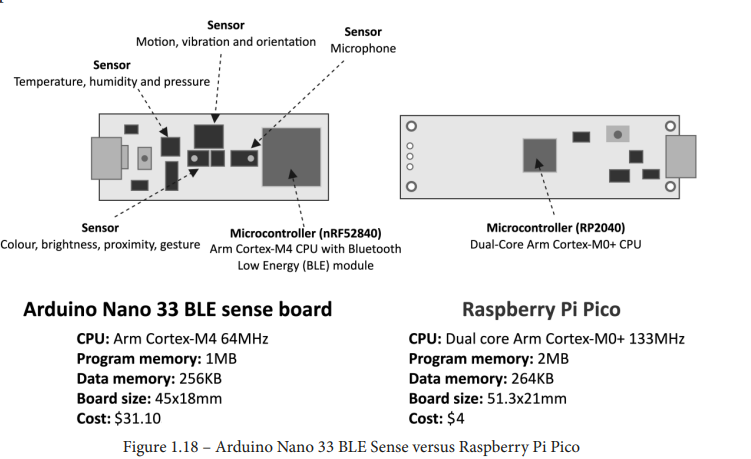
**Timers.**

Timers do not interface with external components since they are used to trigger or synchronize events.

**Presenting Arduino 33 BLE Sense and Raspberry Pi Pico.**

**Arduino Nano**; is a board that combines a microcontroller (**nRF52840**) powered by an Arm Cortex-M4 processor with several sensors and Bluetooth radio for an easy TinyML development experience. We will require just a few additional external components when developing on Arduino Nano since most are already available on-board.

**Raspberry Pi Pico:**  it does not provide sensors and the Bluetooth module on-board. It has a microcontroller (**RP2040**) powered by a dual-core Arm Cortext-M0 + processor for unique and powerful TinyMl applications.



**Setting up Arduino Web Editor, TensorFlow, and Edge Impulse.**

**TensorFlow:** is an end-to-end free and open source software platform developed by Google for ML. We will be using this software to develop and train our ML models using Python in Google Colaboratory.

TensorFlow does not require setting up because Colab comes with it.

In Colab, we recommend enabling the GPU acceleration on Runtime tab to speed up the computation on TensorFlow.

TensorFlow is not the only tool from Google that we will use. In fact, once we have produced the ML model, we will need to run it on the microcontroller. For this, Google developed TensorFlow Lite for Microcontrollers (TFLu).

TFLu is a key software library to unlock ML application on low-power microcontrollers. Written in C/C++ the library does not require an operating system and dynamic memory allocation.

TFLu does nor need setting up because it is included in Arduino Web Editro.

**Prototyping with Microcontrollers.**

The source code and additional material are available in the Chapter02 folder on the GitHub repository (<https://github.com/PacktPublishing/TinyMLCookbook/tree/main/Chapter02>).

**Code debugging 101.**

All programs are prone to bugs, and print debugging is a basic process that prints statements on the output terminal to give insight into the program execution.

In contrast to the standard C library printf function, the Serial.Print () function requires initialization before transmitting data. Therefore, we initialize the peripheral with the Arduino Serial.begin () function, which only requires the baud rate as an input argument.

The baud rate is the data transmission rate in bits per seconds, and it is set to 9600 bps.

However, we can’t use the peripheral immediately after the initialization because we should wait until it is ready to transmit. So, we use while (!Serial) to wait until the serial communication is open.

**NB**: Print debugging is a simple debugging approach, but it has significant disadvantage with the increase of software complexity, such as the following:

* Needing to re-compile and flash the board every time we add or move Serial.print().
* Serial.print () costs in terms of program memeory footprint.
* We could make mistakes reporting the information (for example, using print to report an usigned int variable that is actually signed).

Explore **serial wire debug (SWD)** debuggers to make this process less painful.

SWD is an Arm debug protocol for almost all Arm Cortex processors that you can use to flash the microcontroller, step through the code, add breakpoints, and so on with only two wires.

**Implementing an LED status indicator on the breadboard.**

When connecting external components to the microcontroller, we mean physically joining two or more metal connectors together. Although we could solder these connectors, it is not usual for prototyping because it is not quick and straightforward. Therefore, this Getting ready section aims to present a solderless alternative to connect our components effortlessly. Making contacts directly with the microcontroller's pins can be extremely hard for the tiny space between each pin. For example, considering the RP2040 microcontroller, the pin space is roughly 0.5 mm since the chip size is 7x7 mm. Therefore, it would be practically impossible to connect any of our components safely since most terminals have a wire diameter of ~1 mm. For this reason, our platforms provide alternative points of contact with wider spacing on the board. These contact points on the Arduino Nano and Raspberry Pi Pico are the two rows of pre-drilled holes located at the platform's edge. The simplest way to know the correspondence between these contacts and the microcontroller pins is to refer to the datasheet of the microcontroller boards. Hardware vendors usually provide the pinout diagram to note the pins' arrangement and functionality. For example, the following list reports the links to the Arduino Nano and Raspberry Pi Pico pinout diagrams: • Arduino Nano: https://content.arduino.cc/assets/Pinout-NANOsense\_latest. pdf • Rasberry Pi Pico: <https://datasheets.raspberrypi.org/pico/Pico-R3-A4-Pi>

**Why use a Resistor with Led.**

LEds have fized voltage drop when they emit light(vf), the resistor limits the current in the acceptable range means that the vf does not fall out of the expected operating range.

**Introducing the GPIO peripheral.**

General-purpose input/output (GPIO) is the most common and versatile peripheral on microcontroller.

Its primary function is to provide (output) or read (input) digital signals (1 or 0 ) through external pins, commonly called either GPIO, IO, or GP.

The LED blinking is a typical example of configuring the GPIO peripheral in output mode to supply either 3.3V (1) or 0 V (0) programmatically.

**Turning an Led on and off with a push button.**

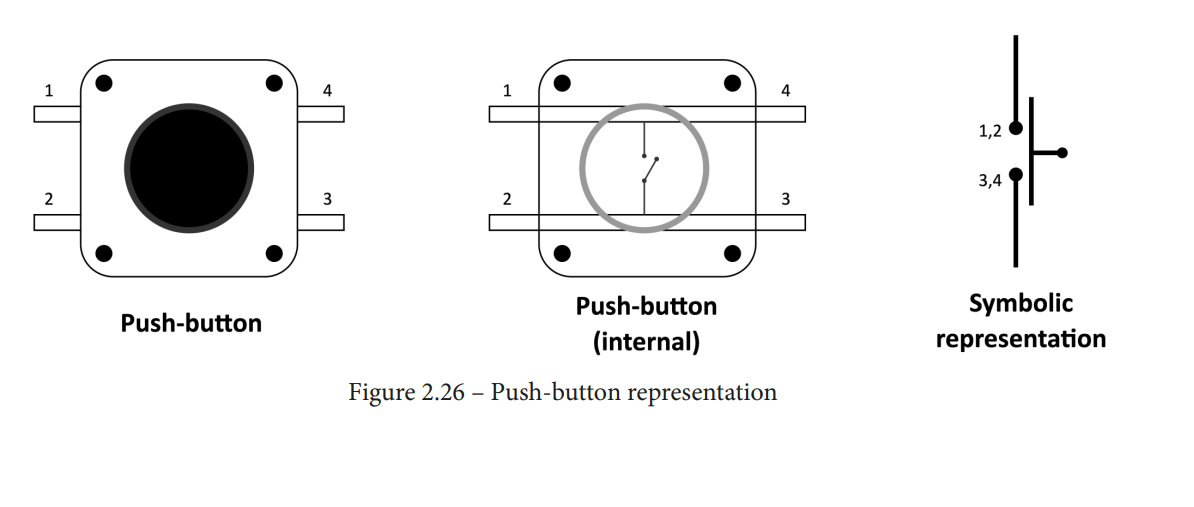
Mbed, or rather Mbed OS, is a real-time operating system (RTOS) specifically for Arm Cortex-M processors, which offers functionalities typical of a canonical OS and drivers to control microcontroller peripherals

**Push Button.**

It’s a device that makes or breaks the connection between two wires. When we press the button, we connect the wires through a mechanical system, allowing the current flow.

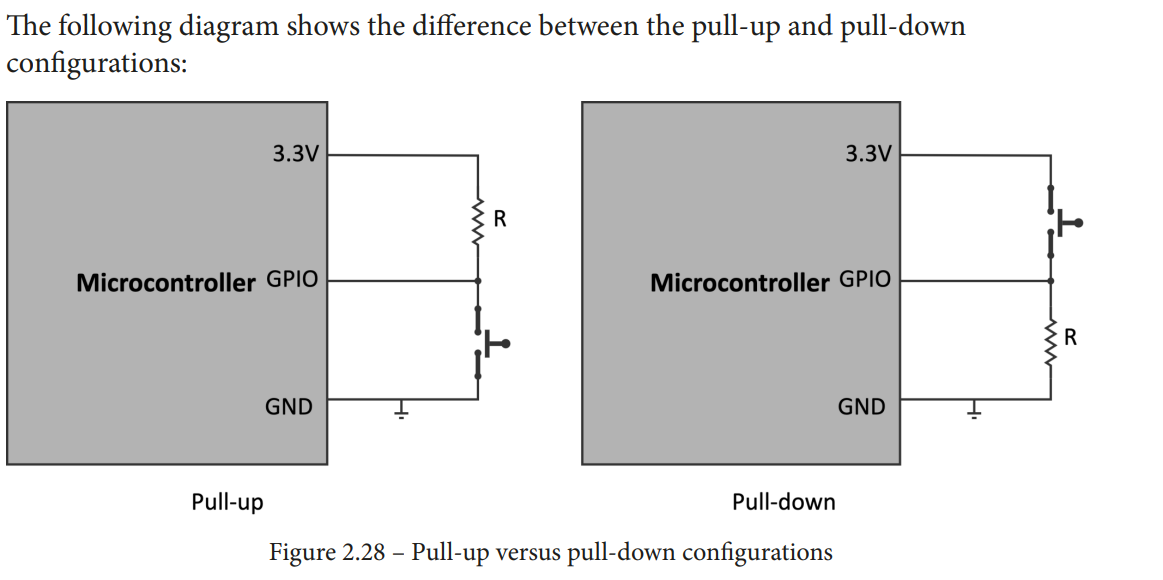
However, it is not like a standard light switch that keeps the wires connected when released. When we don’t apply pressure to the button, the wires disconnect, and the current stopd flowing.

Alhtough this device has four metal legs, it is a two-dimension because the contacts on the opposite side (1,4 and 2,3) are connected



Depending on what logical level we want in the pushed state, the resistor can be as follows:

* **Pull-up:** The resistor connects the GPIO pin to the 3.3V. Thus, the GPIO pin reads LOW in the pushed state and HIGH in the released state.
* **Pull-down:** The resistor connects the GPIO pin to GND in contrast to the pull-up configuration. Thus, the GPIO pin reads the logical level HIGH in the pushed state and LOW in the released state.

****

**Using interrupts to read the push-button state.**

The previous recipe explained how to read digital signals with the GPIO peripheral. However, the proposed solution is inefficient because the CPU wastes cycles waiting for the button to be pressed while it could do something else in the meantime. Furthermore, this could be a scenario where we would keep the CPU in low-power mode when there is nothing else to do.