Arduino Development Cookbook

What this book will do for you...

- Read data from sensors and take action based on the environment
- Use the Arduino to turn on lights, write to screens, or play light shows
- Manipulate motors and other actuators to control the movement of different objects
- Set up electronic circuits on a breadboard to interact with the Arduino
- Explore hacks to push your project to the next level
- Make your projects wireless and make them communicate with the computer

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Arduino Development Cookbook comprises clear and step-by-step recipes that give you the toolbox of techniques to construct any Arduino project, from the simple to the advanced. Each chapter gives you more essential building blocks for Arduino development, from learning about programming buttons through to operating motors, managing sensors, and controlling displays. Throughout, you'll find tips and tricks to help you troubleshoot your development problems and push your Arduino project to the next level!

Over 50 hands-on recipes to quickly build and understand Arduino projects, from the simplest to the most extraordinary

Cornel Amariei

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Working with Buttons'
- A synopsis of the book’s content
- More information on Arduino Development Cookbook

About the Author

Cornel Amariei is a Romanian inventor and entrepreneur in the fields of Robotics and 3D printing. He has been working with the Arduino platform since its early days in 2007. His past experience involves large cargo gamma ray scanning robotics, ATM security systems, and blind assisting devices. In his spare time, he is a performing musician playing multiple instruments—predominately the guitar. He is also a swimmer, water polo player, and photographer.

Over the years, he has built hundreds of Arduino projects, ranging from flying Quadcopters to levitating magnets and underwater robots. Currently, he splits his time between doing his undergraduate studies in electric engineering and computer science at Jacobs University in Bremen, Germany, and his start-ups and research and development job.
The year was 2005 when a few guys from the Interaction Design Institute Ivrea, Italy wanted to create a simple microcontroller board for their students—a board that was more modern, cheaper, and easier to use than the designs available at that moment. And they named it Arduino, after the local bar, which was named after King Arduino.

The initial version was bulky, complicated to connect, and lacked USB, and other features commonly found these days, but the board had potential. Now, Arduino is renowned for its simplicity and ease of use. Children are building projects using Arduino that only 10 years ago would have required engineers.

The whole design is open sourced and clones of the board can be found everywhere in the world. There is no known number of Arduino boards but it is in the range of hundreds of thousands or even more. Everybody can design their own custom implementation of the standard invented in 2005.

Today, Arduino has been to every corner of the planet and even above it. It has fueled other revolutions such as the maker, the open source and 3D printing movements. It is continuously upgraded to be faster and handle more. But what is Arduino?

Arduino is a microcontroller board, designed to connect to electronics and control them. We can write code for the Arduino that will get data from the environment, and make decisions and take actions based on the data. Robots, 3D printers, toys, even toasters may have an Arduino inside, powering up all the interaction.

This book contains recipes that show how to implement key topics of the Arduino, starting from basic interaction with buttons and LEDs, going up to interaction with the Global Positioning System (GPS), making music, or communicating with the Internet. It is intended for programming or electronics enthusiasts who want to combine the best of both worlds to build interactive projects.
What This Book Covers

Chapter 1, Power on – Arduino Basics, will teach you to connect, install, and transfer the first program to the Arduino board. This chapter covers the basics of how to use the Arduino board, the types of boards, and how to use the Arduino IDE.

Chapter 2, Blinking LEDs, covers one of the basic uses of Arduino, controlling LEDs. Various types and implementations have been covered, RGB LEDs, blinking and fading LEDs, 7-segment displays, or more advanced control techniques.

Chapter 3, Working with Buttons, will show you how to detect and use buttons as a key input method. Several types of buttons have been covered along with solutions to the most common button implementation issues. Also, ways of connecting more buttons than available digital pins have been shown.

Chapter 4, Sensors, covers the most important sensors that can be connected to the Arduino. Probably the most important thing for Arduino is to be able to read as many parameters from the environment as possible. Using sensors, it can read distance, temperature, light intensity, or even global localization.

Chapter 5, Motor Control, will show you how to connect and control multiple types of motors. Making things move is incredibly easy using motors and Arduino. Small and large, brushless and servos motor along with speed and direction control, have all been covered here.

Chapter 6, More Output Devices, talks about getting more out of Arduino. This chapter covers how to control different loads, how to make sound, how to isolate and protect the board, and how to command more outputs.

Chapter 7, Digital Communication with Arduino, covers several communication protocols such as UART, I2C, Serial, and Ethernet, to get the most out of the communication interfaces available on Arduino. Arduino can communicate with other boards, computers, and even the Internet.

Chapter 8, Hacking, talks about the small hacks that can help an Arduino design go further. It includes speeding up the PWM, reacting to external interrupts, or even storing data inside the Arduino forever.

Appendix, Electronics – the Basics, covers the basics of electronics, such as breadboards, Ohm's law, and so on.
Working with Buttons

In this chapter, we will cover the following recipes:

- Connecting a button
- Button with no resistor
- The toggle switch
- Button to serial
- Button debouncing
- 1,000 buttons, 1 pin
- Button multiplexing

Introduction

Buttons are the basis of human interaction with the Arduino. We press a button, and something happens. They are simple components, as they only have two states: opened or closed. When a button is closed, current can pass through it. When it's opened, no current can pass. Some buttons are closed when we push them, some when they are released.

In this chapter, we will explore various button configurations and see how to tackle common problems with these. Let's jump in!

Connecting a button

One of the basic interactions you can have with the Arduino is pushing a button, which causes another action. Here, we will see how to connect and use a button.
To keep the example simple, we will connect a button to the Arduino, and whenever we press and hold it, the internal Arduino LED will light up. But first, we need to talk a little about buttons. There are a few common configurations found in everyday electronics. We can categorize buttons/switches based on three main characteristics:

- Momentary and maintained buttons
- Open or closed buttons
- Poles and throws

**Momentary** buttons are active as long as they are pressed, while **maintained** buttons keep the state we let them in. Keyboards have momentary buttons while the typical light switch is a maintained button.

Momentary buttons can either be opened or closed. This reflects the connection state when not pressed. A closed momentary switch will conduct current while not pressed and interrupt the current when pressed. An opened button will do the opposite.

Lastly, there are poles and throws. The following figure explains the main two types:

**Single Pole Single Throw (SPST)** switches have a closed state in which they conduct current, and an opened state in which they do not conduct current. Most momentary buttons are SPST.
**Single Pole Double Throw (SPDT)** switches route the current from the common pin to one of the two outputs. They are typically maintained; one example of this is the common light switch.

The common button we'll be using in this chapter is a push button. It's a small momentary opened switch. It typically comes in a 4-pin case:

![Push button diagram](image)

The pins inside the two red ellipses are shorted together. When we press the button, all four pins are connected.

### Getting ready

These are the ingredients needed to execute this recipe:

- An Arduino board connected to a computer via USB
- A breadboard and jumper wires
- A push button, which can be found at any electronics store, such as the online shops of Sparkfun, Radioshack, Adafruit, and Pololu
- A resistor between 1K–100K ohm

### How to do it...

The following are the steps to connect a button:

1. Connect the Arduino GND and 5V to separate long strips on the breadboard.
2. Mount the push button on the breadboard and connect one terminal to the 5V long strip and the other to a digital pin on the Arduino—in this example, pin 2.
3. Mount the resistor between the chosen digital pin and the GND strip. This is called a pull-down setup. More on this later.
Working with Buttons

**Schematic**

This is one possible implementation on the second digital pin. Other digital pins can also be used.

![Schematic Diagram]

Here is an example of wiring it on a breadboard:

![Breadboard Diagram]
Code
The following code will read if the button has been pressed and will control the built-in LED:

```cpp
// Declare the pins for the Button and the LED
int buttonPin = 2;
int LED = 13;

void setup() {
    // Define pin #2 as input
    pinMode(buttonPin, INPUT);
    // Define pin #13 as output, for the LED
    pinMode(LED, OUTPUT);
}

void loop() {
    // Read the value of the input. It can either be 1 or 0.
    int buttonValue = digitalRead(buttonPin);

    if (buttonValue == HIGH) {
        // If button pushed, turn LED on
        digitalWrite(LED, HIGH);
    } else {
        // Otherwise, turn the LED off
        digitalWrite(LED, LOW);
    }
}
```

If the button is connected to a different pin, simply change the `buttonPin` value to the value of the pin that has been used.

How it works...

The purpose of the button is to drive the digital pin to which it's connected to either HIGH or LOW. In theory, this should be very simple: just connect one end of the button to the pin and the other to 5V. When not pressed, the voltage will be LOW; otherwise it will be 5V, HIGH. However, there is a problem. When the button is not pressed, the input will not be LOW but instead a different state called floating. In this state, the pin can be either LOW or HIGH depending on interference with other components, pins, and even atmospheric conditions!

That's where the resistor comes in. It is called a pull-down resistor as it pulls the voltage down to GND when the button is not pressed. This is a very safe method when the resistor value is high enough. Any value over 1K will work just fine, but 10K ohm is recommended.
**Code breakdown**

The code takes the value from the button. If the button is pressed, it will start the built-in LED. Otherwise, it will turn it off.

Here, we declare the pin to which the button is connected as pin 2, and the built-in LED on pin 13:

```cpp
int buttonPin = 2;
int LED = 13;
```

In the `setup()` function, we set the button pin as a digital input and the LED pin as an output:

```cpp
void setup() {
    pinMode(buttonPin, INPUT);
    pinMode(LED, OUTPUT);
}
```

The important part comes in the `loop()` function. The first step is to declare a variable that will equal the value of the button state. This is obtained using the `digitalRead()` function:

```cpp
int buttonValue = digitalRead(buttonPin);
```

Lastly, depending on the button state, we initiate another action. In this case, we just light up the LED or turn it off:

```cpp
if (buttonValue == HIGH){
    digitalWrite(LED, HIGH);
} else {
    digitalWrite(LED, LOW);
}
```

**There's more...**

In this example, we've seen how to connect the button with a pull-down resistor. However, this is not the only way. It can also be connected using a pull-up resistor.

**Pull-up configuration**

In a pull-up configuration, the resistor will pull up the voltage to 5V when the button is not pressed. To implement it, connect one terminal of the button to the digital pin and the other one to GND. Now, connect the resistor between the digital pin and 5V.
This configuration will return inverted values. When pressed, the button will give LOW, not HIGH, as it will draw the pin down to GND, 0 V. However, it brings no advantages over the pull-down configuration.

**Multiple buttons**
What if we want to implement multiple buttons? We only need to use one digital pin configured as input for each button we use. Also, each button needs its independent pull-down or pull-up resistor.

**See also**
For other topics regarding buttons, check the following important recipes in this chapter:

- The *Button with no resistor* recipe
- The *Button to serial* recipe
- The *Button debouncing* recipe

**Button with no resistor**
It is simple to connect a button to the Arduino. You need the button, some wires, and a resistor. But what if we no longer need the resistor and want to still be able to use the button with no false readings?

The resistor is mandatory for proper operation of a button, and everybody will insist on using it. However, there is a little secret embedded in each Arduino pin. Each pin already has a pull-up resistor that we can enable with just one small change in our code.

**Getting ready**
For this recipe, you will need just two components:

- An Arduino board connected to a computer via USB
- A push button

**How to do it...**
There is just one simple step in this recipe:

1. Connect the Arduino GND to a terminal on the button and connect the chosen digital pin to the other terminal.
**Schematic**
Here is one implementation on the 12th digital pin. Other digital pins can also be used.

For most buttons with standard through-hole terminals, we can directly input the pins into the terminals on the Arduino.
# Code

The following code will read if the button has been pressed and will control the built-in LED:

```cpp
// Declare the pins for the Button and the LED
int buttonPin = 12;
int LED = 13;

void setup() {
    // Define pin #12 as input and activate the internal pull-up resistor
    pinMode(buttonPin, INPUT_PULLUP);
    // Define pin #13 as output, for the LED
    pinMode(LED, OUTPUT);
}

void loop() {
    // Read the value of the input. It can either be 1 or 0
    int buttonValue = digitalRead(buttonPin);

    if (buttonValue == LOW) {
        // If button pushed, turn LED on
        digitalWrite(LED, HIGH);
    } else {
        // Otherwise, turn the LED off
        digitalWrite(LED, LOW);
    }
}
```

If the button is connected to a different pin, change the `buttonPin` value to the value of the pin that has been used.

---

## How it works...

When we press the button, the value of the Arduino pin should be either LOW or HIGH. In this configuration, when we press the button, the pin is connected directly to GND, resulting in LOW. However, when it is not pressed, the pin will have no value; it will be in a floating state.

To avoid this, an internal pull-up resistor is connected between each pin and 5V. When we activate the resistor, it will keep the pin at HIGH until we press the button, thus connecting the pin to GND.
**Code breakdown**

The code takes the value from the button. If the button is pressed, it will start the built-in LED. Otherwise it will turn it off.

Here, we declare the pin to which the button is connected as pin 12, and the built-in LED as pin 13:

```cpp
int buttonPin = 12;
int LED = 13;
```

In the `setup()` function, we set the button pin as a digital input and we activate the internal pull-up resistor using the `INPUT_PULLUP` macro. The LED pin is declared as an output:

```cpp
void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(LED, OUTPUT);
}
```

In the `loop()` function, we continuously read the value of the button using the `digitalRead()` function, and we store it in a newly declared variable called `buttonValue`:

```cpp
int buttonValue = digitalRead(buttonPin);
```

Lastly, depending on the button state, we initiate another action. In this case, we just light up the LED or turn it off:

```cpp
if (buttonValue == LOW){
  // If button pushed, turn LED on
  digitalWrite(LED, HIGH);
} else {
  // Otherwise, turn the LED off
  digitalWrite(LED, LOW);
}
```

There's more...

It is easy to connect a button to the Arduino without any resistors. What if we need more buttons?

**Multiple buttons**

Each button requires its own digital pin and resistor. The Arduino already has one pull-up resistor in each digital and analog pin, so in the end, all that is needed is one pin for each individual button. The other terminal of the buttons is tied together to GND.
The toggle switch

A toggle switch can be very useful for various projects. It can hold one or more constant states. For example, we can have a few of them and configure a certain system to work a certain way based on the configuration. This is done all the time on computer motherboards and other electronic devices.

A two-state toggle switch is just like a standard push button; only, it remains in the state we put it in. An on-off switch is a two-state toggle switch. It becomes more useful when we have a three-state toggle switch as in this recipe. It has two usable states and an off state.

In this recipe, we will use a basic toggle switch to light up two LEDs. When the toggle switch is in one end position, only one LED will be switched on. If it is in the other end position, the other LED will be switched on. Finally, if the toggle switch is in the center, both LEDs will be switched off.

Getting ready

The following are the ingredients required to execute this recipe:

- An Arduino board connected to a computer via USB
- A breadboard and jumper wires
- At least one toggle switch, which we can always take out of an old electric toy or buy from Sparkfun, Digikey, and so on
- Two LEDs and two resistors between 220-1,000 ohm

How to do it...

Follow these steps in order to connect the toggle switch to the LEDs:

1. Connect the Arduino GND to a long strip on the breadboard.
2. Mount the toggle switch and connect the middle terminal to the long GND strip on the breadboard.
3. Connect the other two terminals of the toggle switch to digital pins 2 and 3 on the Arduino board.

4. Mount the two LEDs and connect their ground terminal to the long GND strip on the breadboard.

5. Connect pin 5 to one of the LEDs using a resistor between pin 5 and the input of the LED. Do the same for pin 6 and the other LED.

**Schematic**

This is one possible implementation. Other digital pins can also be used.

Here is a possible breadboard implementation:
**Code**

The following code will check the status of the toggle switch and will drive the LEDs accordingly:

```c
int buttonPin1 = 2;
int buttonPin2 = 3;

int LED1 = 5;
int LED2 = 6;

void setup() {
  // Define the two LED pins as outputs
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);

  // Define the two buttons as inputs with the internal pull-up resistor activated
  pinMode(buttonPin1, INPUT_PULLUP);
  pinMode(buttonPin2, INPUT_PULLUP);
}

void loop() {
  // Read the value of the inputs. It can be either 0 or 1
  // 0 if toggled in that direction and 1 otherwise
  int buttonValue1 = digitalRead(buttonPin1);
  int buttonValue2 = digitalRead(buttonPin2);

  if (buttonValue1 == HIGH && buttonValue2 == HIGH) {
    // Switch toggled to the middle. Turn LEDs off
    digitalWrite(LED1, LOW);
    digitalWrite(LED2, LOW);
  } else if (buttonValue1 == LOW) {
    // Button is toggled to the second pin
    digitalWrite(LED1, LOW);
    digitalWrite(LED2, HIGH);
  } else {
    // Button is toggled to the third pin
    digitalWrite(LED1, HIGH);
    digitalWrite(LED2, LOW);
  }
}
```
If the toggle switch is connected to other pins, simply change the `buttonPin1` and `buttonPin2` variables. The same goes for the LED pins.

How it works...

When the toggle switch is toggled to one of the two end positions, it will connect one of the Arduino pins to GND. The pin that is not connected to GND will stay HIGH due to the internal pull-up resistor in the Arduino pin. If the toggle switch is in the central position, no pin will be connected to GND and both will be HIGH.

Code breakdown

The code takes the value from the two pins connected to the toggle switch. If one of them goes LOW, it will turn on one of the LEDs.

Here, we declare the pins to which the toggle switch is connected as pins 2 and 3. The LEDs are defined on pins 5 and 6:

```cpp
int buttonPin1 = 2;
int buttonPin2 = 3;

int LED1 = 5;
int LED2 = 6;
```

In the `setup()` function, we set the pins for the LEDs as outputs and the two pins going to the toggle switch as inputs. Also, we activate the internal pull-up resistor so that it does not need an external one:

```cpp
void setup() {
  // Define the two LED pins as outputs
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);

  // Define the two buttons as inputs with the internal pull-up resistor activated
  pinMode(buttonPin1, INPUT_PULLUP);
  pinMode(buttonPin2, INPUT_PULLUP);
}
```

In the `loop()` function, we continuously read the values of the two pins going to the toggle switch, and we store them in the variables `buttonValue1` and `buttonValue2`:

```cpp
int buttonValue1 = digitalRead(buttonPin1);
int buttonValue2 = digitalRead(buttonPin2);
```
Lastly, depending on the toggle switch state, we initiate another action:

```c
if (buttonValue1 == HIGH && buttonValue2 == HIGH){
    // Switch toggled to the middle. Turn LEDs off
} else if (buttonValue1 == LOW){
    // Button is toggled to the second pin, one LED ON
} else {
    // Button is toggled to the third pin, the other LED ON
}
```

There's more...

Toggle switches can be very useful when used together. A DIP switch is very interesting, as it usually has multiple small toggle switches. Each time we add a toggle switch, we double the number of configurations. With four two-state switches, we can have up to 16 configurations.

This is useful when we have a system that needs a lot of configurations. Rather than uploading code again and again, we can use the toggle switches to choose what to do.

See also

Use the following links to find some common switches you can buy:

- [https://www.sparkfun.com/products/9276](https://www.sparkfun.com/products/9276)
- [https://www.sparkfun.com/products/8034](https://www.sparkfun.com/products/8034)

Button to serial

If we want to easily track how a button acts, serial communication is the best and simplest way. All we need to do is to read the status of the button and print it to the serial connection.

Testing whether a button is working can be solved by using an LED. However, if we need to check two buttons or better understand what's happening when the button is pressed, serial communication is much safer and may even be simpler.

Getting ready

The following are the ingredients required to execute this recipe:

- An Arduino board connected to a computer via USB
- A button
How to do it...

This recipe uses the Button with no resistor recipe's hardware implementation. Please implement the same schematic as in that recipe. We will have different code here, which will output the values on the serial connection.

**Code**

The following code will print the button status on the serial connection:

```cpp
int buttonPin = 2;

void setup() {
    // Define pin #2 as input
    pinMode(buttonPin, INPUT_PULLUP);

    // Establish the Serial connection with a baud rate of 9600
    Serial.begin(9600);
}

void loop(){
    // Read the value of the input. It can either be 1 or 0.
    int buttonValue = digitalRead(buttonPin);

    // Send the button value to the serial connection
    Serial.println(buttonValue);

    // Delays the execution to allow time for the serial transmission
    delay(25);
}
```

If the button is connected to another digital pin, simply change the value of `buttonPin` to the digital pin that has been used.

How it works...

When the button is pressed, it can either return a value of 1 or 0. Because we activated the internal pull-up resistor inside the Arduino pin, the values will be safe, and no floating condition will be obtained. After we read the value of the pin, we send it to the serial connection.

**Code breakdown**

The code takes the value from the button connected to digital pin 2 and writes it to the serial.
In the `setup()` function, we set the button pin as an input and activate the internal pull-up resistor. Then, we start the Serial connection with a speed of 9,600 bits per second:

```cpp
void setup() {
    pinMode(buttonPin, INPUT_PULLUP);
    Serial.begin(9600);
}
```

In the `loop()` function, we continuously read the value of the connected button:

```cpp
int buttonValue = digitalRead(buttonPin);
```

Then, we print the value using the `Serial.println()` command. We can also use `Serial.print()`, however, `println()` will write the value and go to a new line afterwards. This looks much better and it is easier to understand:

```cpp
Serial.println(buttonValue);
```

At the end, we need a delay to allow the data to be transmitted. The delay can be short since we only send one value; however, it is mandatory to have it. Otherwise, the serial will constantly overflow and no good values will reach the computer:

```cpp
delay(25);
```

**There's more...**

To print more than one button, we can use the `Serial.print()` function to write each button state in line and then use the `Serial.println()` function to go to the next line. Here is a simple implementation:

```cpp
Serial.print(buttonValue1); // Print first value
Serial.print(" "); // Leave an empty space between
Serial.println(buttonValue2); // Print the second value
```

**Button debouncing**

A button is a simple device; when we push it, it gives a value, otherwise it gives another value. Unfortunately, it is not always like that. When we push or release a button, for a very small amount of time the button bounces between pushed or not. This is due to mechanical errors and wear and tear in the button.

Even if it is a small amount of time, the Arduino is quick, and when we press the button, it may read values that are quickly bouncing between pressed and not pressed. In most cases, this is not a problem; but in many cases, this happens and it can take hours to detect what is going wrong. Better be safe than sorry!
Another very important application of this is reading a button only once. When we press the button, we keep it pressed for a few milliseconds. In this time, the Arduino can read it hundreds, even thousands of times. It detects a few hundred times instead of once that we pushed the button. This is the primary use of debouncing in the Arduino world.

**Getting ready**

The following are the ingredients required for this recipe:

- An Arduino board connected to a computer via USB
- A button

**How to do it...**

This recipe uses the hardware implementation in the *Button with no resistor* recipe. Please implement the same schematic as in that recipe. We will have a different code here, which will output the debounced values on the serial connection.

**Code**

The following code will read the status of the button and print it over the serial connection:

```c
// Declare the pin for the button
int buttonPin = 2;

// Variable for keeping the previous button state
int previousButtonValue = HIGH;

long lastDebounce = 0; // Last time the button was pressed
long debounceTime = 50; // Debounce delay

void setup() {
    // Define pin #2 as input and activate the internal pull-up resistor
    pinMode(buttonPin, INPUT_PULLUP);
    // Establish the Serial connection with a baud rate of 115200
    Serial.begin(115200);
}

void loop(){
    // Read the value of the input. It can either be 1 or 0
    int buttonValue = digitalRead(buttonPin);
    if (buttonValue != previousButtonValue && millis() - lastDebounce >= debounceTime){
```
// Reading is useable, print it
Serial.println(buttonValue);

// Reset the debouncing timer
lastDebounce = millis();
// Change to the latest button state
previousButtonValue = buttonValue;
}

// Allow some delay for the Serial data to be transmitted
delay(10);
}

If the button is connected to another digital pin, change the value of buttonPin to the value of the digital pin that has been used.

How it works...
To avoid reading the button multiple times and detecting false readings, there are two important steps. First, we only read changes in the button state. If the button has the same value as at the last reading, we ignore it. Second—and here is the important step—if the button has been pressed or released, we don't evaluate its value for the following few milliseconds. This will make sure that no rapid oscillations in the button state are read.

It is possible to implement this with a simple delay() function; however, delay() stops the Arduino board from executing anything else.

Code breakdown
The code takes the value from the button connected on digital pin 2 and uses debouncing logic to assure proper output.

We need to declare a few variables. The previousButtonValue variable keeps track of the previous state of the button. The lastDebounce variable is important; it stores the particular time at which the button was pressed earlier. The debounceTime variable is the amount of time in milliseconds between each reading. It is important for these two variables to be declared as long type, because the numbers get pretty big quite fast.

int previousButtonValue = HIGH;

long lastDebounce = 0; // Last time the button was pressed
int debounceTime = 50; // Debounce delay
The Arduino keeps an internal count of time passed since the program began running. To access this time, we can use the `millis()` function, which returns the time in milliseconds.

In the `setup()` function, we set the button pin as an input and we activate the internal pull-up resistor. Then, we start the serial connection with a speed of 115,200 bits per second:

```cpp
void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  Serial.begin(115200);
}
```

In the `loop()` function, we continuously read the value of the connected button:

```cpp
int buttonValue = digitalRead(buttonPin);
```

Now we need to apply the debouncing part of the code. It consists of an `IF` clause with two conditions:

- The first condition checks whether the new reading is different from the last one. We don't want to detect a button push a hundred times when we press once.
- The second condition checks whether enough time has passed since the last reading. This makes sure the value doesn't bounce between states.

The time is declared in the `debounceTime` variable. A good value is around 50 milliseconds. It can be lower, but we will only need it to be lower if we want to press the button more than 20 times a second.

```cpp
if (buttonValue != previousButtonValue && millis() – lastDebounce >= debounceTime)

```

Then, we print the value using the `Serial.println()` command:

```cpp
Serial.println(buttonValue);
```

It is very important on each usable reading to update the `lastDebounce` and `previousButtonValue` variables. These will be the new values that the debouncing filter will compare.

```cpp
lastDebounce = millis();
previousButtonValue = buttonValue;
```
At the end, we need a short delay to allow the data to be transmitted:

```c
  delay(10);
```

See also

To clearly understand what contact bouncing is, visit http://en.wikipedia.org/wiki/Debounce#Contact_bounce.

1,000 buttons to 1 pin

One button, one pin—that is the way things are usually done on Arduino boards. But it is so limiting. There are some tricks that let you connect more than one button to a pin. Actually, it is even possible to connect 1,000 buttons to just 1 pin. We will explore this possibility in this recipe.

Getting ready

The following are the ingredients required for this recipe:

- An Arduino board connected to a computer via USB
- A breadboard and jumper wires
- Three buttons
- Four resistors of equal value: 1K ohm works well

How to do it...

We implement a simple configuration using only three buttons on the same pin. Here are the steps:

1. Connect the Arduino GND to a long strip on the breadboard. Also connect the Arduino 5V to a long strip.
2. Connect one of the resistors from the GND strip to an analog pin—here, pin A0—on the Arduino.
3. Connect three resistors in series starting at the 5V strip.
4. At each junction of two resistors, connect one button. Also connect the third button at the end of the resistor series.
5. Connect the other terminals of the buttons together and to the A0 analog pin on the Arduino.
Schematic
This is one possible implementation. Other analog pins can also be used.

This is a possible breadboard implementation:
The following code will read the analog pin on the Arduino board and detect which button is pressed:

```cpp
// Declare the Analog pin on the Arduino board
int buttonPin = A0;

void setup() {
    // Establish the Serial connection with a baud rate of 9600
    Serial.begin(9600);
}

void loop() {
    // Read the value of the input. It can vary from 0 - 1023
    int buttonValue = analogRead(buttonPin);

    if (buttonValue < 200) {
        // A value under 200 represents no button pushed
        Serial.println("0");
    } else if (buttonValue >= 200 && buttonValue < 300) {
        // A value between 200 - 300 represents the third button
        Serial.println("S3");
    } else if (buttonValue >= 300 && buttonValue < 400) {
        // A value between 300 - 400 represents the second button
        Serial.println("S2");
    } else if (buttonValue >= 400) {
        // A value greater than 400 represents the first button
        Serial.println("S1");
    }

    // Delays the execution to allow time for the Serial transmission
    delay(25);
}
```

If the buttons are connected to another analog pin, simply change the `buttonPin` variable to the analog pin that has been used.
How it works...

This is all possible due to an electric circuit called the voltage divider. Each time a button is pressed, a different voltage divider is created. Each button brings a different voltage to the analog pin on the Arduino. We can read this analog voltage and attribute a specific value to each button.

Code breakdown

The code takes the value from the analog pin and checks to which button it corresponds. It prints out on the serial which button has been pushed.

Here, we declare the analog pin that has been used:

```c
int buttonPin = A0;
```

In the setup function, we start the serial connection with a speed of 9,600 bits per second:

```c
void setup() {
    Serial.begin(9600);
}
```

In the `loop()` function, we continuously read the value of the analog pin, which can be from 0–1023:

```c
int buttonValue = analogRead(buttonPin);
```

Then, we check which button is pressed. We should be able to attribute one exact value to each button. However, due to component tolerances and errors, it's much safer to use an interval. If the expected value is 250 and the button returns 251, the code will not detect the button. In this example, the intervals are extreme: 0–200 for no button, 200–300 for the third button, 300–400 for the second button, and over 400 for the first. The best way to find out the values is to make a simple program print the analog value of the pin on the serial:

```c
if (buttonValue < 200) {
    // A value under 200 represents no button pushed
    Serial.println("0");
} else if (buttonValue >= 200 && buttonValue < 300) {
    // A value between 200 - 300 represents the third button
    Serial.println("S3");
} else if (buttonValue >= 300 && buttonValue < 400) {
    // A value between 300 - 400 represents the second button
    Serial.println("S2");
} else if (buttonValue >= 400) {
    // A value greater than 400 represents the first button
    Serial.println("S1");
}
```
In the end, always have some delay when working with serial communication to avoid overflow:

```c
    delay(25);
```

---

There's more...

This is a very useful solution when we need multiple buttons but have only a few pins left. Another advantage is the time needed to read the analog pin. It takes around 0.1 millisecond to read the analog pin, which solves some problems with debouncing.

Here are a few tips on how to easily do more with this configuration.

---

More buttons

The title says 1,000 but there are only three buttons here. However, the principle is the same. We can connect as many buttons as we have to a theoretical maximum of 1023. Each button needs a resistor, so for a configuration of 100 buttons, we will use 100 resistors in series and at each junction of two resistors, we will mount a button. Again, we will mount the hundredth button at the end. The Rd resistor that connects the pin to GND is also mandatory.

The values of the resistors are also very important. It is recommended to have a high value for the Rd resistor: somewhere between 100K–1M ohm. The other resistors should be equal to make things easier: somewhere between 1K–10K ohm.

---

Finding each button

The simplest way to find the value of every button connected is to print the value of the analog pin repeatedly while pressing the buttons one at a time. Each one should give a unique value and a value close to 0 when no button is pressed. Here is the code to print pin A0:

```c
    Serial.println(analogRead(A0));
    delay(10);
```

---

Pressing multiple buttons

If we don't use too many buttons, we can actually detect multiple button presses. When we press two buttons, the resistors will be connected in parallel and the overall resistance will drop. This will cause the analog reading to be higher. Use the serial output to check what happens in your configuration.

---

See also

Button multiplexing

Using a multiplexer, it is possible to make the Arduino read over a hundred buttons easily. A multiplexer/demultiplexer is an integrated circuit that selects one of several inputs and forwards them to the output. It requires a few control pins to determine which input to forward to the output.

Getting ready

Following are the ingredients required for this recipe:

- An Arduino board connected to a computer via USB
- A breadboard and jumper wires
- Four buttons
- A 4051 multiplexer or similar, which we can find at any electronics store and online at Digikey, Sparkfun, Adafruit, and so on

How to do it...

We implement a simple configuration using only four buttons. Here are the steps:

1. Connect the Arduino **GND** to a long strip on the breadboard. Also connect the Arduino **5V** to a long strip.
2. Mount the four buttons and connect one of their terminals to the long **GND** strip.
3. Connect the other terminal of each button to an individual input/output pin on the 4051—in this case, pins y0, y1, y2, and y3.
4. Connect the **E**, **VEE**, and **GND** pins of the 4051 multiplexer to the long **GND** strip.
5. Connect the **Vcc** pin on the 4051 to the **5V** strip on the breadboard.
6. Connect S0, S1, and S2 to three digital pins on the Arduino—in this example, 8, 9, and 10.
Schematic

This is one possible implementation. Other pins can also be used.

This is a possible breadboard implementation:
Code
The following code will read the four buttons connected to the multiplexer by switching the active pin on it:

```cpp
// Define the input pin on the Arduino and the 3 selection pins connected to the 4051
int buttonPin = 2;
int A = 10;
int B = 9;
int C = 8;

void setup() {
  // Define pin #2 as input with the pull up resistor on
  pinMode(buttonPin, INPUT_PULLUP);

  // Define the output pins going to the control lines of the Multiplexer
  pinMode(A, OUTPUT);
  pinMode(B, OUTPUT);
  pinMode(C, OUTPUT);
  // Establish the Serial connection with a baud rate of 9600
  Serial.begin(9600);
}

void loop(){
  // We first read port I00
  digitalWrite(A, LOW);
  digitalWrite(B, LOW);
  digitalWrite(C, LOW);
  int buttonI00 = digitalRead(buttonPin);

  // Then we read port I01
  digitalWrite(A, HIGH);
  digitalWrite(B, LOW);
  digitalWrite(C, LOW);
  int buttonI01 = digitalRead(buttonPin);

  // Then we read port I02
  digitalWrite(A, LOW);
  digitalWrite(B, HIGH);
  digitalWrite(C, LOW);
  int buttonI02 = digitalRead(buttonPin);

  // Then we read port I03
```
digitalWrite(A, HIGH);
digitalWrite(B, HIGH);
digitalWrite(C, LOW);
int buttonIO3 = digitalRead(buttonPin);

// Then we print to Serial the values
// We print them in-line separated by a space
Serial.print(buttonIO0);
Serial.print(" ");
Serial.print(buttonIO1);
Serial.print(" ");
Serial.print(buttonIO2);
Serial.print(" ");
Serial.println(buttonIO3);

// Delays the execution to allow time for the serial
delay(25);
}

How it works...
The multiplexer/demultiplexer is a useful component, but, a little tricky to understand.

Here we used a demultiplexer configuration. Each demultiplexer has one output and a number
of inputs—in our case, eight. Also, it has control lines—in our example, three. Each control line
represents a number: power of 2 minus 1. For the 4051, A = 1, B = 2 and C = 4. If we want
to read input IO5, we set A and C to HIGH and S1 to LOW. This means the output will be
connected to A + C = 5 input; therefore, pin IO5.

Basically, a multiplexer gives the power to connect one Arduino pin to one I/O pin on
the multiplexer. Only one pin can be connected at any particular time.

Code breakdown
The code commands the connection on the multiplexer using the three command lines. It uses
one input digital pin to get the value from the buttons and prints it on the serial connection.

Here, we declare the used pins:

    int buttonPin = 2;
    int A = 10;
    int B = 9;
    int C = 8;
Working with Buttons

In the `loop()` function, we set the multiplexer to each pin we want to read and we read it. In order to read pin IO0, we set A, B, and C to low, so their sum is 0. When we want to read pin 1, we set A to 1. IO3 will result A and B to HIGH:

```cpp
digitalWrite(A, HIGH);
digitalWrite(B, LOW);
digitalWrite(C, LOW);
int buttonIO1 = digitalRead(buttonPin);
```

We do this for each button we want to read and then we print the output values on the serial.

There's more...

Here we have only four buttons on four pins—not a very good ratio of pins to buttons. However, for the same number of pins we can get eight buttons, as there are four free pins on the multiplexer.

More buttons

Even eight buttons on four pins is not too much. There are 16-channel multiplexers, such as the 4067 that require four control lines, totaling sixteen buttons on five pins. We can go even further! We can use more multiplexers, and we only need one new line for each multiplexer to connect to its output while sharing the control lines. Using a 4067 and all the pins, except 0 and 1, on the Arduino Uno, we can read 224 buttons. On the Arduino Mega, this will result in 800 buttons. The sky is the limit with multiplexers.

See also

Where to buy this book

You can buy Arduino Development Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.