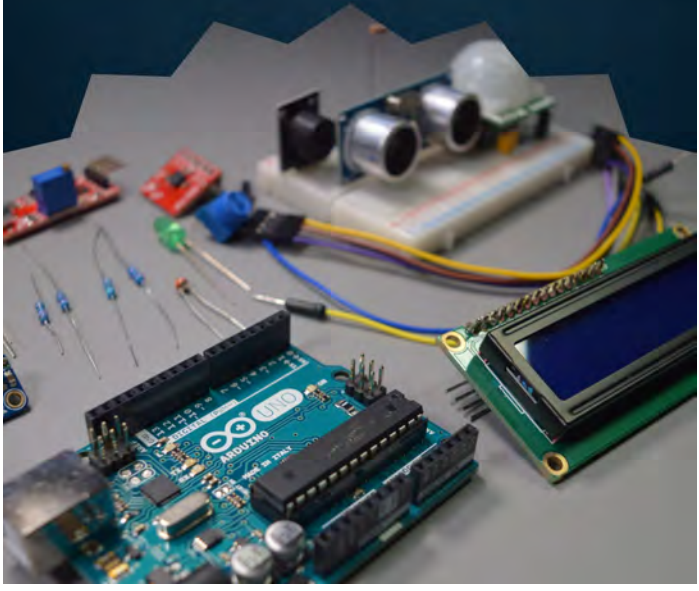
Tech Explorations

ARDUINO PERIPHERALS & CIRCUITS

GET THE MOST OUT OF YOUR ARDUINO WITH ARTICLES FROM THE TECH EXPLORATIONS BLOG



Peter Dalmaris, PhD

Arduino Peripherals and Circuits

Get the most out of your Arduino with articles from the Tech Explorations Blog

Welcome to this special collection of articles, meticulously curated from the Tech Explorations blog and guides. As a token of appreciation for joining our email list, we offer these documents for you to download at no cost. Our aim is to provide you with valuable insights and knowledge in a convenient format. You can read these PDFs on your device, or print.

Please note that these PDFs are derived from our blog posts and articles with limited editing. We prioritize updating content and ensuring all links are functional, striving to enhance quality continually. However, the editing level does not match the comprehensive standards applied to our Tech Explorations books and courses.

We regularly update these documents to include the latest content from our website, ensuring you have access to fresh and relevant information.

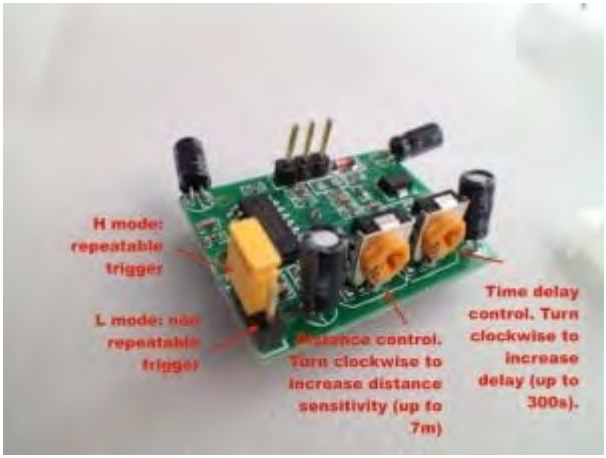
License statement for the PDF documents on this page

1. Infrared sensor (PIR) tips

Arduino peripherals guide series

PIR sensor calibration

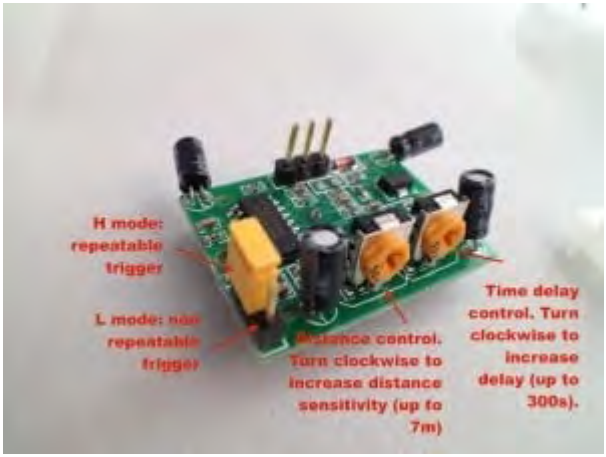
Learn how to calibrate and improve the reliability of the infrared motion sensor.



The PIR sensor is very sensitive and often generates false triggers.

The example sketch in lecture 0500 of Arduino Step by Step Getting Started is very simple; it merely reports what the sensor is telling it, and does not try to compensate for its limitations.There are two things you can do to create a more reliable detection system (and this can deliver the functionality you are asking for).

1. Calibrate the sensor (hardware)



Look at the annotated photograph (above) where I have marked the purpise of each knob.

Notice the two orange knobs on one side of the sensor?

The one on the left controls the sensor range. Turn it clockwise to calibrate the effective range. The maximum range is around 7 meters.

I usually calibrate the range to its minimum setting, but turning the left knob all the way anticlockwise. I do this because most of my experiments involve small distances.The knob on the right controls sensitivity. Turn it clockwise to decrease sensitivity. Decreased sensitivity means that the device will trigger (i.e. report movement) only when said movement is prolonged.

I keep the sensitivity setting high by turning the sensitivity knob all the way anti-clockwise. This means that the sensor will trigger even for very small movements in front of it. Beware, that at this sensitivity setting you will have erroneous triggers.There is also a jumper swtch that controls the trigger mode. I normally use H-mode triggering (as in the photo). In H-

mode, the sensor will repeatedly trigger when it “thinks” that there is movement.

2. Improve reliability in software

You can improve the reliability of the sensor by adding additional logic to your Arduino sketch.

In this example (heavily borrowing from a sketch I found in the Arduino playground), I define a cutoff time. Any LOW triggers from the sensor within this cutoff time are ignored as false, and the Arduino continues to think that there is movement. Have a look at it (but get a copy from Github if you want to try it out):

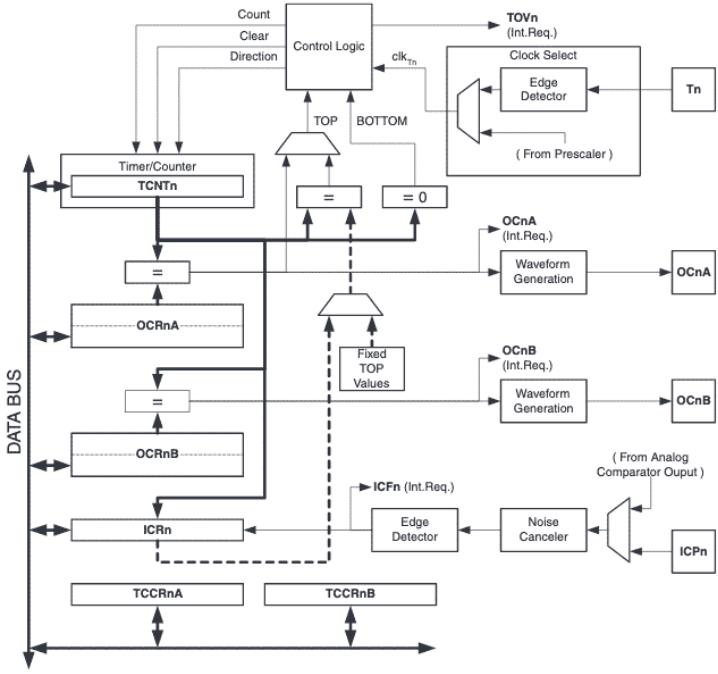
/** PIR sensor tester*/ int ledPin = 13; // choose the pin for the LEDint inputPin = 3; // choose the input pin (for PIR sensor)int pirState = true; // we start, assuming no motion detectedint val = 0; // variable for reading the pin statusint minimummSecsLowForInactive = 5000; // If the sensor reports low for// more than this time, then assume no activitylong unsigned int timeLow;boolean takeLowTime; //the time we give the sensor to calibrate (10-60 secs according to the datasheet)int calibrationTime = 30; void setup() { pinMode(ledPin, OUTPUT); // declare LED as output pinMode(inputPin, INPUT); // declare sensor as input Serial.begin(9600); //give the sensor some time to calibrate Serial.print(“calibrating sensor “); for(int i = 0; i < calibrationTime; i++){ Serial.print(“.”); delay(1000); } Serial.println(” done”); Serial.println(“SENSOR ACTIVE”); delay(50);} void loop(){ val = digitalRead(inputPin); // read input value if (val == HIGH) { // check if the input is HIGH digitalWrite(ledPin, HIGH); // turn LED ON if (pirState) { // we have just turned on pirState = false; Serial.println(“Motion detected!”); // We only want to print on the output change, not state delay(50); } takeLowTime = true; } else { digitalWrite(ledPin, LOW); // turn LED OFF if (takeLowTime) { timeLow = millis(); takeLowTime = false; } if(!pirState && millis() – timeLow > minimummSecsLowForInactive){ pirState

= true; Serial.println(“Motion ended!”); delay(50); } }}

Arduino peripherals guide series

The basic functions of the TimerOne library

Learn the basic functions of the TimerOne library that makes it easy to use the Atmega328's 16-bit counter.



Timer1 s one of the libraries written to take advantage of the 16-bit counter that comes with the Atmega328 (datasheet, see section 16, page 111).

Think of it as a clock. Every time the Atmega oscillator ticks, the counter increases by one.

The Timer1 library makes it easy to start, stop and reset/restart the counter, just like you can do with a regular timer wrist-watch (remember those?).

Initialization

To initialize the timer1 object, you can use this syntax:

```
Timer1.initialize(1000);
```

This will set the period of the timer object to 1000 microseconds. In practical terms, this code will setup the counter to generate a timer interrupt every 1000 microseconds.

To handle this interrupt, you must write an interrupt service routine.

Using the timer

Imagine this scenario: let's say that you want to stop this timer, temporarily.

You can connect a button to a digital input. When the user presses the button, your sketch will stop the timer like this:

```
Timer1.stop();
```

When the user wants to restart the times, he can press the same button again, and your sketch start the timer like this:

```
Timer1.start();
```

This call will get the timer to continue where it left off. Just like with a standard timer watch.

The user may want to reset the timer and start from the beginning. You can add another button to your circuit, the reset button. When the user presses this button, the sketch will reset the counter like this:

Timer.restart();

As a result, the timer will start again from zero.

An example sketch

The Timer1 library comes with an example sketch which I copy below.

Here is the original.

I have removed the comments to compact the code.

There are a few things to notice in this sketch:

1. The interrupt handler. Notice the code "**Timer1.attachInterrupt(blinkLED);**" This registers the function "**blinkLED**" as the one that will handle the interrupts from Timer1.
2. The variable "**blinkCount**" is declared as "volatile" as it is used inside the interrupt hander ("**blinkLED**") and the rest of the sketch. Volatile variables are loaded from the RAM, always, instead of the CPU register. Registers contains temporary variable values which may loose consistency when are accessed by interrupt request handlers.

3. Any code in the regular part of the sketch that must not be interrupted (i.e. "**critical code**") is enclosed between the "**noInterrupts();**" and "**interrupts();**" functions. This way, we ensure that variables must be updated reliably, contain reliable values.

```
#include <TimerOne.h>const int led = LED_BUILTIN; // the pin with a LED

void setup(void){ pinMode(led, OUTPUT); Timer1.initialize(150000); Timer1.attachInterrupt(blinkLED); Serial.begin(9600);}

int ledState = LOW;volatile unsigned long blinkCount = 0;

void blinkLED(void){ if (ledState == LOW) { ledState = HIGH; blinkCount = blinkCount + 1; } else { ledState = LOW; } digitalWrite(led, ledState);}

void loop(void){ unsigned long blinkCopy; // holds a copy of the blinkCount noInterrupts(); blinkCopy = blinkCount; interrupts(); Serial.print("blinkCount = "); Serial.println(blinkCopy); delay(100);}
```

Now that you know the basics of the TimerOne library go ahead and give it a try.

How to find your device I2C address

How do you find out the address of an I2C device that is poorly documented?

The I2C bus is shared so that a single data line can deliver communications between a single master (like an Arduino) and several slave devices (like LCD screens and real-time clocks).

How to find the I2C address of a device

So, how do you know which address belongs to a device, especially since in most cases, the manufacturer will not tell you?

The easiest way is to use an I2C address scanner.

The scanner is a small sketch that you can upload on your Arduino.

Connect the device you want to probe to the Arduino (preferably without connecting other I2C devices at the same time), and run the sketch.

The scanner will cycle through all possible I2C addresses, and once it receives a response from the device, it will inform you of the address that worked.

Then, repeat the process for other I2C devices.

How to deal with I2C address conflict

What if there is a conflict? What if you have two devices that "listen" on the same address?

In that case, you must select an alternate address for one of the devices.

Let's take, for example, the common parallel to I2C LCD adapter. In most cases, it has a default address of 0x27. You can choose to change this address to something else.

An I2C LCD display adaptor. The red circle shows the position of the I2C address configuration pads.

On this board, there are three pairs of pads, titled A0, A1, and A2.

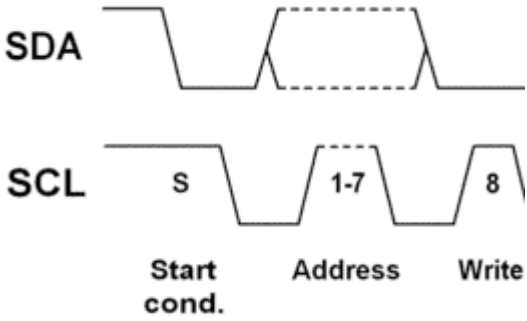
You can change the address by creating shorts between these pads with a soldering iron and some solder.

If there is no documentation to tell you which short combination creates which address, then repeat the probing process I described at the start of this email to determine the new I2C of the device.

Many I2C devices offer a similar method of configuring an alternate address. While solder pads are most common, you can also find versions that use jumpers or switches.

Getting started with I2C on the Arduino

What is I2C? How many devices are supported on a bus? Is I2C supported by the Arduino Uno? How does it work?



Many peripheral devices use the I2C bus to communicate with the outside world. The Arduino and the various Atmega microcontrollers, of course, support I2C.

What is I2C?

The I2C bus is a technology that allows multiple devices to communicate over a single pair of wires. The pair contains a wire for data, and a wire for the clock signal.

To regulate traffic, a I2C bus contains a single node (often referred to as “master”), and multiple regular nodes (often referred to as “slaves”). The master will control the bus by

sending commands and information to individual slaves.

It is also possible to have multi-master I2C environments. In such environments, and the term reveals, multiple master devices can be connected to multiple regular devices on a single bus.

Because the bus is shared by many devices, only one of them can communicate with the host at one time, and the system depends on each device having a unique address.

How are I2C devices addressed?

An I2C bus is a byte with seven bits, which allows for 128 addresses. Therefore, theoretically, an I2C bus can support up to 128 connected devices.

However, several of these addresses are reserved for "special purposes."

From I2C-bus.org, I copy these special-purpose addresses, in case you are wondering (here is the original):

10-bit addresses, binary noted, MSB is left	Purpose
0000000 0	General Call
0000000 1	Start Byte
0000001 X	CBUS Addresses
0000010 X	Reserved for Different Bus Formats
0000011 X	Reserved for future purposes
00001XX X	High-Speed Master Code
11110XX X	10-bit Slave Addressing
11111XX X	Reserved for future purposes

Once we subtract the reserved addresses, we are left with 112 addresses available to use on a single I2C bus, using 7-bits for each address.

You probably think that 112 devices are more than enough for any conceivable project. That's true, but consider that I2C is designed for use in industrial, telecommunications and medical applications, to name a few.

Practically, it is everywhere, including in your car. So 112 addresses are not enough! That's why I2C supports 10-bit addressing that increases the address space by 10 times to 1023, in case you need it.

You can find more details here.

How to use I2C in the Arduino Uno?

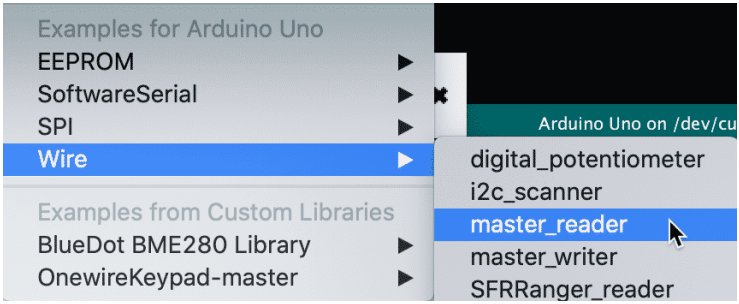
In the Arduino Uno, which is powered by an Atmega328P microcontroller unit, you can use a compatible version of the original I2C, known as TWI.

TWI is short for "Two Wire Interface".

You can find details in the Atmega328P documentation (see page 215).

TWI on the Atmega328P supports the 7-bit I2C addressing scheme, multi-master, and can operate as both a master or a slave device.

The Arduino IDE provides the Wire library which makes it easy to use TWI in your sketches. You can find example sketches under Examples –> Wire.



Let's have a looks at two of the provided examples so that you can learn how your Arduino can read and write data using the I2C bus.

Example: Arduino reads I2C data as master

In the first one, the Arduino operates as a master device, and requests data from a slave device. The slave could be another Arduino, or a sensor.

This example is titled "master_reader", and ships with the Arduino IDE under "Wire".

#include <Wire.h>void setup() { Wire.begin(); Serial.begin(9600); }void loop() { Wire.requestFrom(8, 6); while (Wire.available()) { char c = Wire.read(); Serial.print(c); } delay(500);}

In the sketch, we include the Wire library at the top with the "include" statement.

Inside setup(), we initiate the Arduino on the I2C bus using "Wire.begin()". We only need to do this once. In this example, the "begin()" function does not contain an address parameter. This means that the Arduino will join the I2C bus as a master device.

If you wanted to make the Arduino a slave, you would provide an address, like this:

Wire.begin(8);

With this, the Arduino will join the bus as a slave, listening to address “8”.

In the loop(), we use “Wire.requestFrom(8, 6);” to ask a slave device to return data. The first parameter (“8”) is the slave device address, and the second (“6”) is the quantity of data (in bytes) that we want from the slave.

When the slave starts sending the requested data, the Arduino will store them temporarily in a buffer. We can use “Wire.available()” to check to data. If data is available, we can use “Wire.read()” to get one byte at a time from the buffer and store it in a variable for later use.

We can repeat this process in a loop until the buffer is empty.

To know what to do with each byte, you will need to have some information about the way that the slave device formats data. The slave, for example, may be storing a 2-byte integer in the first two bytes of its response, and a single character (byte) in the 3rd byte of its response. Every device is different in this regard, and I2C is only responsible for the reliable communications between the devices, not for the content of the communications.

Example: Arduino writes I2C data as master

In the second example, the Arduino operates again as a master device, but this time it sends data to another device. The second device could be another Arduino, or a sensor.

This example is titled “master_writer”, and ships with the Arduino IDE under “Wire”.

Here’s the sketch:

#include <Wire.h>void setup() { Wire.begin();}byte x = 0;void loop() { Wire.beginTransmission(8); Wire.write(“x is “); Wire.write(x); Wire.endTransmission(); x++; delay(500);}

Much of this sketch is familiar. We include the Wire library in the header, and call “Wire.begin()” to join the I2C bus.

Inside the loop(), we use “beginTransmission(8);” to begin the transmission of an arbitrary number of bytes to a listening device with address “8”.

We then use “Wire.write(“x is “);” and “Wire.write(x);” to send a string and a single byte respectively.You can use either one repeatedly.

The “write()” function is overloaded with three types so that you can either send a single byte, or a string, or a predetermined number of bytes when used with a second parameter in the form of “Wire.write(data, length)”.

If you use the version of “write” with the two parameters, then the first parameter must be an array of bytes, and the second is the number of bytes to transmit.

To signal to the bus listeners that the Arduino has finished sending data, use “Wire.endTransmission();”. This way, another device will be able to use the bus.

Using I2C: True digital to analog conversion on the Arduino Uno

The Arduino Uno, with its Atmega328P MCU, does not have true digital to analog conversion capability. For this, we turn to an external device, the PCF8591.

The Arduino Uno, based on the Atmega328p, has no true digital to analog capability. It can only simulate analog signals using pulse width modulation.

Although PWM is sufficient for a lot of applications, there are many more where true digital to analog conversion (DAC) is more appropriate.

For example, making sound in the form of audio effects and simple music, is a typical application of DAC. You can store audio in a "wav" file stored on an SD card (which the Arduino can use), then playback the audio to a speaker. Using PWM is

not a good choice for something like this; you need true analog output to reproduce the information in the audio file properly.

To create true analog signals with an Arduino Uno, you can use a DAC module. There are many options in the market.

The NXP PCF8591

A popular DAC module that you can use with your Arduino is the PCF8591 from NXP.

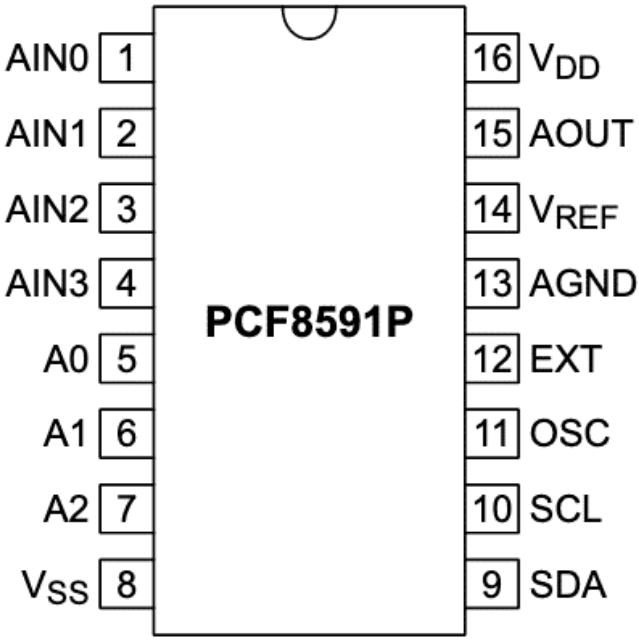
This module contains one 8-bit digital to analog converter and four analog to digital converters, also 8-bit each.

You can connect this device to your Arduino via the I2C bus, using one of 8 possible and configurable addresses. It operates at 5V and 3.3V, so this module is also an excellent choice for 3.3V hosts, like the Arduino Zero and the Raspberry Pi.

Programming the module is easy. Of course, there is a library, but you can also use the raw I2C/TWI interface.

Wiring

Here is the PCF8591 package pin-out, from the datasheet:



Connect the module to your Arduino by doing these connections:

- Device pin 16 (VDD) and 14 (VREF) to Arduino 5V.
- Device pin 13 (AGND) and 12 (EXT) to Arduino GND.
- Device pin 10 (SCL) to Arduino pin A5 (SCL).
- Device pin 9 (SDA) to Arduino pin A4 (SDA).
- Use two 10k resistors to pull up the SCL and SDA pins (connect them to 5V).
- Device pin 8 (Vss), pin 5 (A0), pin 6 (A1) and pin 7 (A2) to GND. This sets the device I2C address to 0x90.

- Connect a 10F capacitor between 5V and GND.

The analog output pin is device pin 15. This is where you can connect an oscilloscope (so you can see the waveform of the analog output), or an amplified speaker (so you can hear it).

Addressing

What is the address of the device? Because we have grounded the three addressing pins 5, 6, and 7, the address is 144, in decimal, or 90 in hexadecimal. To understand how this works, look at the device datasheet, on page 13. There, you will see this table:

Eight different I²C-bus slave addresses can be used to address the PCF8591 (see Table 5).

Table 5. I²C slave address byte

Bit	Slave address							
	7 MSB	6	5	4	3	2	1	0 LSB
slave address	1	0	0	1	A2	A1	A0	R/$\overline{W}$

Because bits 3, 2 and 1 of the slave address are 0, the full 8-bit address byte is “10010000”. Convert this binary number to decimal, and you’ll get 144. In hexadecimal, “0x90”.

Sketch

Now, to the sketch. It is as simple as it gets. Our objective is to generate a sawtooth waveform by using a for() loop that sends a single point 8-bit value to the DAC each time:

#include “Wire.h”void setup(){ Wire.begin();}void loop(){ for (int i=0; i<256; i++) { Wire.beginTransmission(0x90 >> 1); Wire.write(0x40); Wire.write(i); Wire.endTransmission(); }}

In the example below, inside the for() loop, we transmit three

bytes each time:

1. **Wire.beginTransmission(0x90 >> 1)**: We start with the device address to enable the slave device and make it listen to the remainder of the communication. Because our Arduino works with 7-bit I2C addresses, but the PCF8591 expects 8 bits, we are doing a right-shift of the slave address by one bit.
2. **Wire.write(0x40)**: We send the control byte which tells the device that we want to enable the analog output. Hexadecimal "0x40" is binary "01000000" (see below for more details about this control byte).
3. **Wire.write(i)**: We send the byte that contains the actual value that we want to convert into an analog voltage on device pin 15.

At the end of the for() loop, we use "Wire.endTransmission();" to finish the current communications session and release the bus.

I2C bus protocol

Let's have a quick look at the structure of the communication between the Arduino and the PCF8591. As I have mentioned in another article, the I2C bus takes care of facilitating the delivery of messages across the bus. What the devices do with the data is up to them. This means that in order to successful exchange data through I2C, you must know the protocol use by the specific devices you are working on.

To learn about the I2C protocol of the PCF8591 device, you must refer to its datasheet. In page 13 you will find Figure 15 which depicts the protocol used in write mode. In this example, we are using the write more to “write” an analog value to the analog out pin. It looks like this:

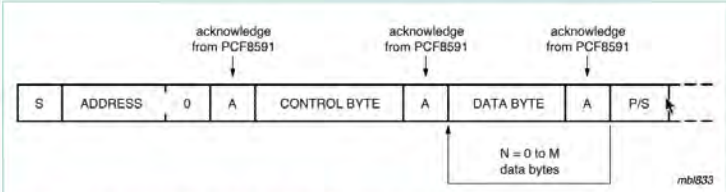


Fig 15. Bus protocol for write mode, D/A conversion

Notice that the protocol expects an address first, followed by a control byte, followed by one or more data bytes. This is exactly the protocol that we have implemented inside the for() loop in our example sketch.

Next, let’s focus on the control byte.

Again, the datasheet is your guide. Go to page 6, where you will find Figure 4:

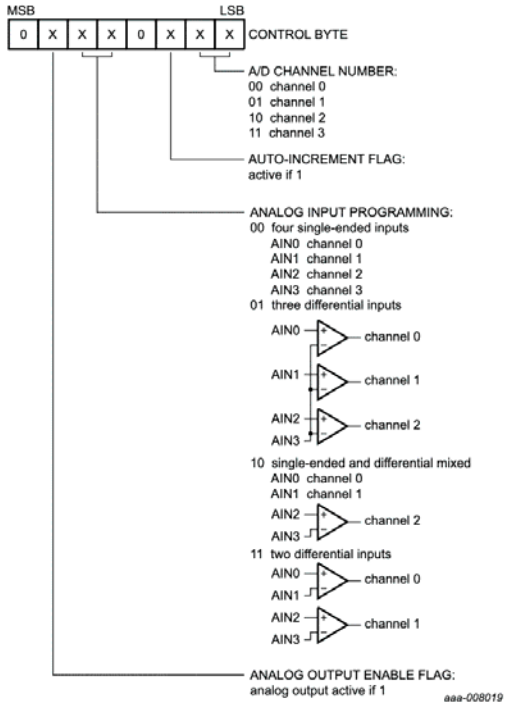


Fig 4. Control byte

Remember that we want to write a value to the analog out pin. The second bit from the left controls this function. This bit is the “analog output enable flag”. To turn on the analog output pin, we must write a “1” to this bit. The rest of the bits don’t matter, so we’ll write zeros everywhere else.

Therefore, the binary version of the control byte is “01000000”. To save a bit of space in our sketch, we can use the equivalent hexadecimal, which is “0x40”.

Either way is correct:

- Wire.write(0x40); // hexadecimal
- Wire.write(B01000000); // binary

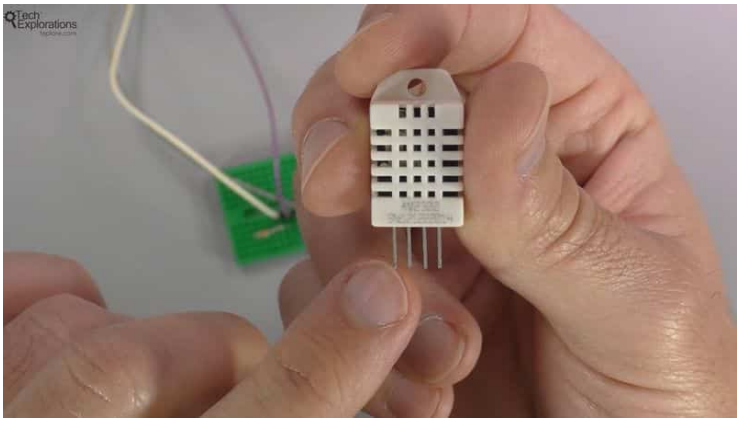
- Wire.write(64); // decimal

As you can see, using I2C is very simple as long as you understand the protocol used by the devices involved. The datasheet is your guide, and with relatively small effort you can significantly expand the capabilities of your Arduino.

The NXP PCF8591 can provide you with true digital to analog conversion capabilities for just couple of dollars, which is a fraction ofthe cost of an Arduino that has this capability built-in.

How accurate are thermometer sensors?

Is the temperature reading from your sensors close to being real? What is real temperature? What are some of the main considerations?



If you need to measure temperature with your gadget, you will need a temperature sensor. Typical sensors include the popular DHT11/22, the BME280, and the TMP36.

The manufacturer publishes accuracy information on the datasheet, and they often look impressive. But when you use these sensors in “real life”, their measurements often seem substantially different to the actual temperature. The actual temperature can be given by another thermometer that you

trust, or by a calibrated reference thermometer.

Which of the aforementioned sensors is “better”?

Here are a few things to remember when you use them.

Comparing temperature sensors

It may sound weird, but comparing these devices is like comparing oranges and apples (and lemons). They are all fruit, but they are different.

In the case of these thermometer modules, they are all able to measure ***a*** temperature. This temperature will only be moderately accurate within a short-range compared to a reference (i.e., “true”) temperature.

The two common digital modules (DHT22 and BME280) are calibrated in the factory but are made with different tolerances, accuracy, and materials.

DHT22

Have a look at the DHT22 datasheet.

It’s operating temperature is -40 to 80 Celsius, but you can assume that its stated accuracy of 0.5C is only achieved around the middle of this range. Still, this accuracy is measured in a lab with controlled conditions. Parameters such as air currents, humidity, exposure to sunlight, the age of the device itself are within specific ranges. The stated accuracy for this sensor will be different when you use it in any other conditions.

BME280

The BME280 is a more professional sensor (and expensive) in

the sense that although it is factory calibrated, it provides a way for the user to custom-calibrate it for the given application and conditions; this provides a way to adjust the sensor. If you plan to use it in warm, humid climates, for example, you can take temperature and humidity with a reference instrument, or use a calibration chamber to calibrate the offset of the BMP180 so that it matches the reference. Here is an excellent article on the calibration topic.

Once the BME280 is calibrated, you can expect a typical temperature accuracy of ±1.0°C. The datasheet (look at page 12) becomes as bad as ±1.25°C for temperatures towards the edges of its useful range. It gets even worse (±1.5°C) towards the edges of its operating range of -40°C to -60°C.

TMP36

The TMP36 is a linear analog sensor. It gives you a voltage that is linearly proportional to the temperature it senses. Although its stated accuracy is ±2°C (see first page of the datasheet), on top of this and everything else discussed above, a lot depends on the conversion formula you use.

To get the temperature from the TMP36, you have to convert the charts in Figure 5, page 5, of the datasheet into a formula (I have copied the charts below).

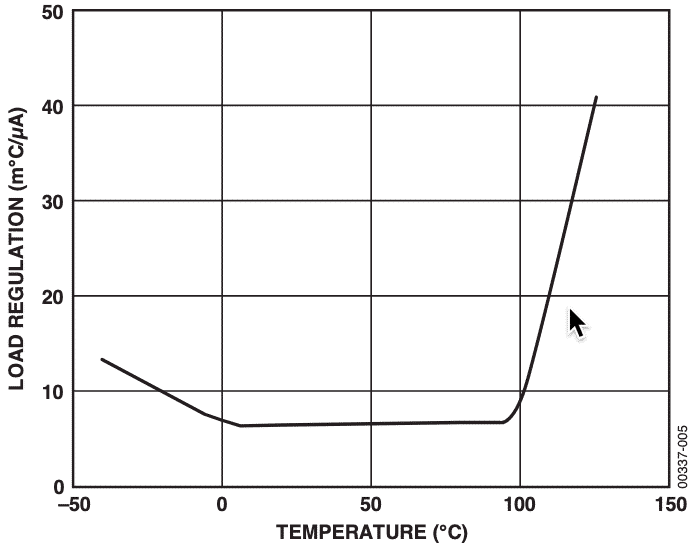


Figure 5. Load Regulation vs. Temperature (m°C/μA)

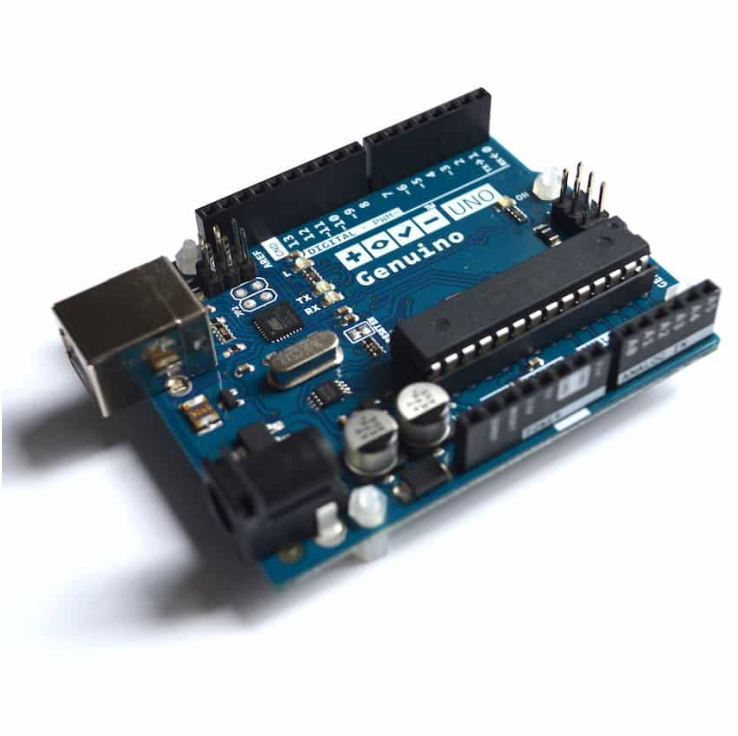
The temperature you get from the sensor depends on how well your calculation formula matches the line, and how accurate your microcontroller can measure the voltage from the sensor. As you can see from the chart, the line is mostly straight and flat for temperatures that we are likely to encounter in a regular application (between 5°C and 90°C). Other issues to consider (and perhaps include in your calculation) are the heat produced by other parts of your circuit, and even accurate the calculation is performed on the microcontroller..

From all this, you can see that there are significant differences between these sensors, which result in the differences that we see when we take measurements with one next to the other.

Without a good reference or at least trusted thermometer, you can’t be sure how close either of the mentioned sensors are to the true temperature.

If the true temperature is what you want, you can’t depend on subjective observation, like “it feels like 25C, but the sensor

says 21C" because you just can't trust yourself for things like this. You'll have to get into the details of how the sensor works, and how to calibrate it or compensate its readings.



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

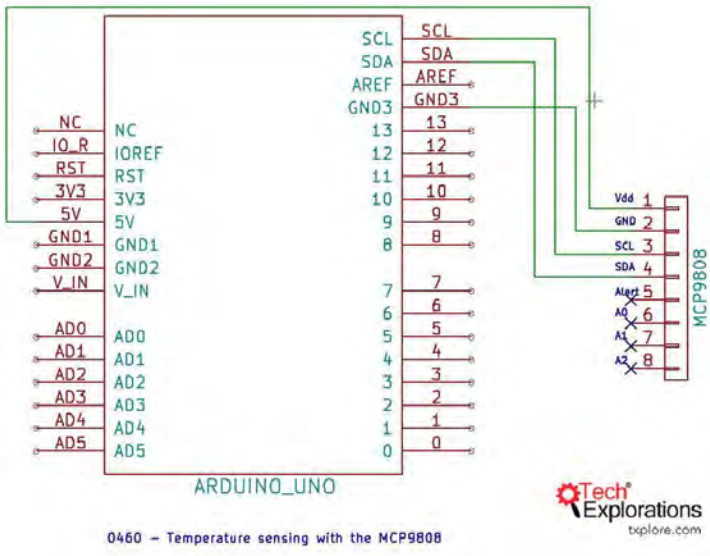
This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.

Learn more

Arduino peripherals guide series

MCP9808: an accurate thermometer module for your Arduino

The MCP9808 is a very accurate temperature sensor for your Arduino. It offers user-selected resolutions, programmable alerts, I2C connectivity, and works with 5V and 3.3V Arduinos.



In a previous article, I discussed three common thermometer sensors used by Arduino makers. A student recently asked if

any thermometer sensors are more accurate than the common DHT11/22, TMP36, and BMP280.

The Microchip MCP9808 high-accuracy temperature sensor

Of course, there is! Presenting the MCP9808 from Microchip.

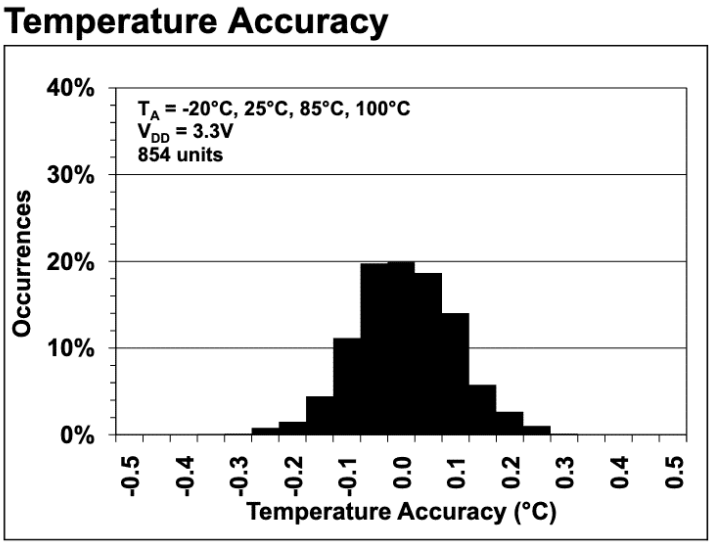
You can see that this module is made for accuracy by looking at the first page of its datasheet; this is what you see:

- Accuracy:
  - ±0.25 (typical) from -40°C to +125°C
  - ±0.5°C (maximum) from -20°C to 100°C
  - ±1°C (maximum) from -40°C to +125°C
- User-Selectable Measurement Resolution:
  - +0.5°C, +0.25°C, +0.125°C, +0.0625°C

There's more information than usual about the accuracy of the device, with typical and maximum values at different ranges. It is also possible to select the measurement resolution!

Evaluation of the MCP9808 accuracy plots

The first page also contains a very interesting temperature accuracy distribution plot:



This plot tells us that 20% of readings from a sample of 854 units of this module contain **approximately** a 0.0C error!

The data shows that this is a very accurate sensor!

The datasheet contains several more interesting plots, like these in page 7:

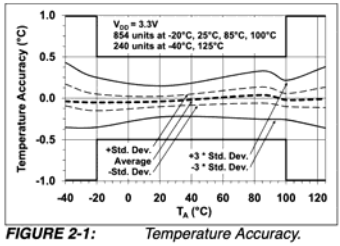


FIGURE 2-1: *Temperature Accuracy.*

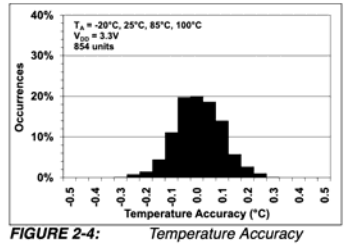


FIGURE 2-4: *Temperature Accuracy Histogram.*

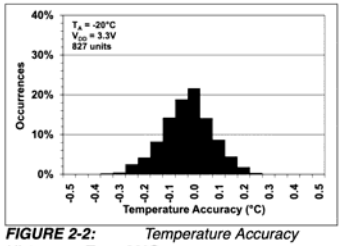


FIGURE 2-2: *Temperature Accuracy Histogram, T_A = -20°C.*

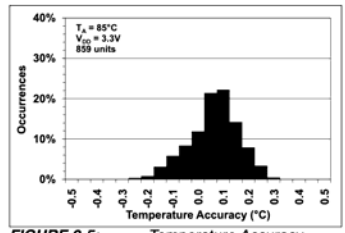


FIGURE 2-5: *Temperature Accuracy Histogram, T_A = +85°C.*

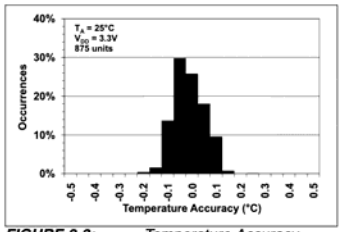


FIGURE 2-3: *Temperature Accuracy Histogram, T_A = +25°C.*

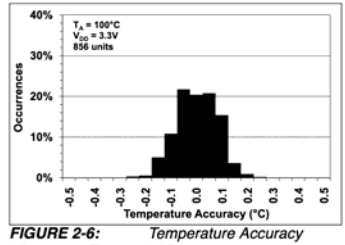


FIGURE 2-6: *Temperature Accuracy Histogram, T_A = +100°C.*

These plots illustrate how accuracy changes over the range of temperatures that the sensor is capable of measuring. Notice that in figure 2-3, which shows the error distribution at 25C, the distribution is tall and concentrated thinly around the 0 mark on the horizontal axis; this indicates that the manufacturer calibrated this sensor to work best in regular room temperatures, where most of our electronics work.

Then, look at Figures 2-5 and 2-4, how the error distributions at 85C and -20C are shorter and more spread out, indicating a larger spread of the measurement errors.

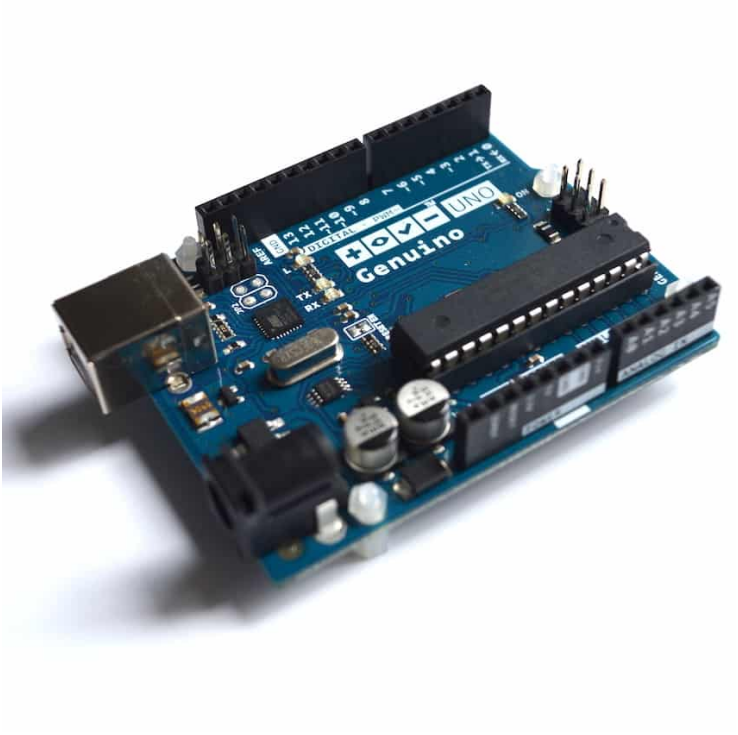
This is a visual description of the fact that the accuracy of a sensor varies along the whole width of the range in which it

operates.

Learn more

This uses the I2C interface to communicate with a microcontroller. Adafruit has published a library that makes using it even easier.

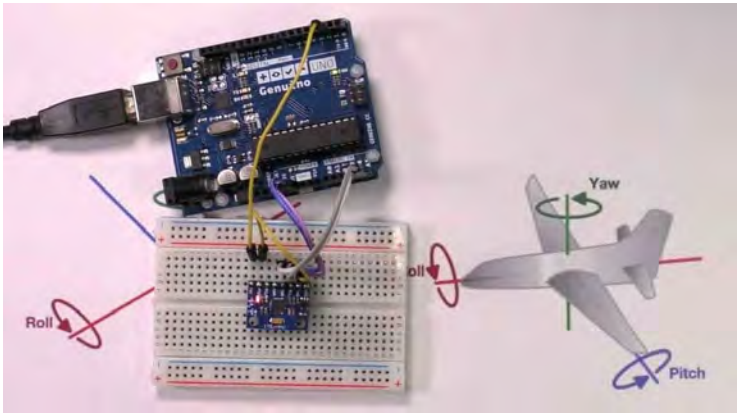
If you want to learn how to use the MCP9808 with your Arduino, consider my course Arduino Step by Step Getting Started, where I cover the sensor in Lecture 14 of Section 10.



Arduino peripherals guide series

Getting useful motion data from the MPU-6050 device

Modern gadgets are mobile and can be “aware” of their motion with the use of motions sensors. There are integrated devices that combine (fuse) multiple motion sensors to produce accurate and reliable motion information for your gadget.



If you have played with motion sensors, like the MPU-6050 6-axis gyroscope and accelerometer device, you may have noticed that the raw data that you can read from them contain too much noise to be useful.

The values can fluctuate wildly, even when the device is sitting motionless on the floor.

This happens because the raw data that are generated by the device contains a significant amount of noise.

Clearly, using the raw data is not an option. There's not much you can do with them other than printing them to the serial monitor. Applications such as an autopilot for a drone are out of the question.

A simple solution: averaging

What you need to do is to process the raw data in order to extract useful information.

Processing raw motion sensor data can be done in a variety of ways.

At the bottom end of the complexity scale, you can use simple techniques such as smoothing or averaging the raw accelerometer values (also, see here). Such techniques can help clear up the noise and generate more reliable and useful motion information.

A better solution: fusion

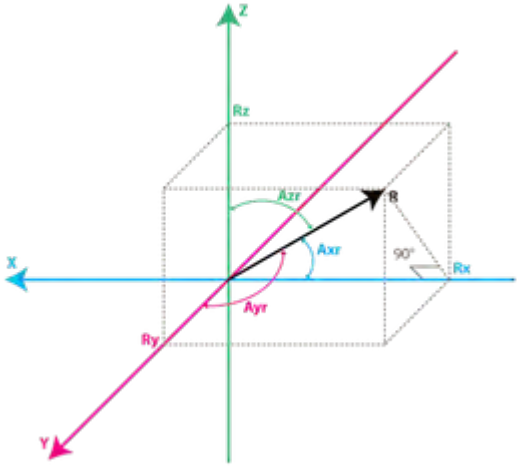
The most reliable motion information come by combining multiple types of motion sensors.

The MPU6050 contains an accelerometer and a gyroscope.

There are various algorithms that take input from both the accelerometer and the gyroscope on the sensor to calculate reliable and useful motion information. Because these algorithms take input from multiple sensor types, we often refer to them as "fusion" algorithms.

You can then use the output of these algorithms to control a drone or or to navigate aircraft (see inertial navigation system).

This page contains an excellent tutorial on how to combine data from the gyro and accelerometer to derive useful values.



Understanding motion sensor averaging will stretch your trigonometry! (source)

You may also want to read about the Kalman filter; this is an algorithm used widely for guidance, navigation, and control of all sorts of vehicles, from planes to spacecraft.

Here is an implementation of the Kalman filter for the Arduino.

The problem is that these algorithms are hard to understand for most of us since they contain a lot of mathematics and

tend to be complicated; they are also demanding in RAM because of the amound of calculations that they entail so that they are barely able to work on an Arduino Uno.

The MPU-6050 Digital Motion Processor

Luckily, there are motion sensing devices that solve this problem by running fusion algorithms internally. This means that your Arduino will be able to receive clean and reliable "fused" motion information from the module, instead of noisy motion data.

The MPU-6050 is such a device.

It contains an integrated Digital Motion Processor (DMP); this is a module that does all sorts of motion-related calculations inside the chip, very efficiently and quickly.

The MPU-6050 DMP is relatively sophisticated. It supports 3D motion processing and gesture recognition algorithms. It even reports temperature, and if you connect an external magnetometer, the DMP can also give you heading information. The external magnetometer can be connected to the MPU-60505 via an auxiliary I2C interface, so again you don't have to tie up the Arduino for this additional sensor.

An MPU-6050 library for Arduino

Even though using the DMP is not as straightforward as just getting the raw data (for some reason, this critical feature is not well documented in the datasheet), brilliant people have managed to create a library for the Arduino that makes use of it. It comes full of sample sketches with which you can play.

If you are serious about creating gadgets that make use of motion sensor data, you need to understand some of the mathematics involved, even if you use a DMP that does the grunt work for you. However, with the MPU-6050 and a good

library, you will be able to create reliable motion-sensing gadgets without needing a degree in higher applied mathematics.

Learn more

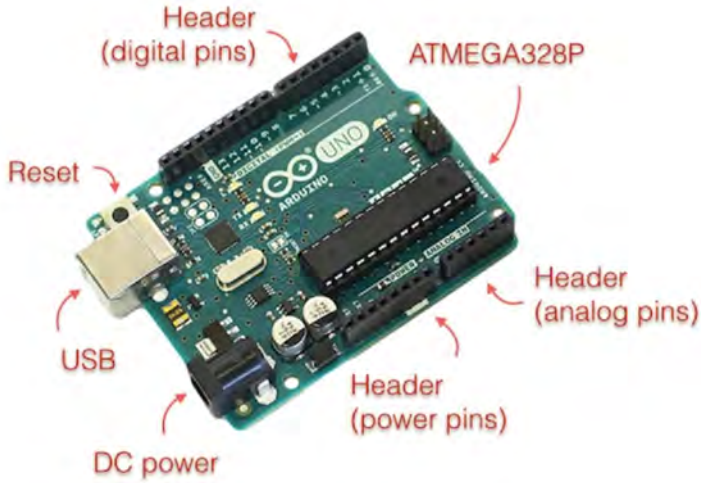
If you want to learn how to use the MPU-6050 with your Arduino, consider my course Arduino Step by Step Getting Serious, where I cover the sensor in Section 3.

9. What to do with unused pins on an Atmega328P or Attiny85?

Arduino peripherals guide series

What to do with unused pins on an Atmega328P or Attiny85?

Is it OK to leave unused pins “floating”? The answer is in the datasheet of your favorite microcontroller. Let’s decipher it here.



In most projects, you will end up with having several I/O pins on your Arduino or Attiny85 (a cousin of the Atmega328p) left

unconnected.

What should you do with these pins?

Attiny85, datasheet recommendation about unused pins

Download the datasheet for the Attiny85.

Go to page 57 and have a read of section 10.2.6 Unconnected Pins.

I am quoting here (emphasis mine):

If some pins are unused, it is recommended to ensure that these pins have a defined level. Even though most of the digital inputs are disabled in the deep sleep modes as described above, floating inputs should be avoided to reduce current consumption in all other modes where the digital inputs are enabled (Reset, Active mode and Idle mode). The simplest method to ensure a defined level of an unused pin is to enable the internal pull-up. In this case, the pull-up will be disabled during reset. If low power consumption during reset is important, **i**t is recommended to use an external pull-up or pull-down. Connecting unused pins directly to VCC or GND is not recommended, since this may cause excessive currents if the pin is accidentally configured as an output.

Atmel 8-bit AVR Microcontroller with 2/4/8K Bytes In-System Programmable Flash, page 57

You will find the exact same sentence in page 62, section 13.2.6, of the ATmega328P datasheet.

Datasheet recommendation

interpretation

So, what does this mean?

If you want to be entirely compliant with the Atmel recommendation, you should connect any unused I/O pins to Vcc or GND via a large (~10KOhm) external resistor. In other words, you should not leave those pins "floating".

GND is preferable to Vcc as it reduces the risk of a short-circuits.

If you don't want to add the additional external resistors in your circuit, you can simplify your design by opting for the MCU internal pull-up resistors. This, in effect, means that you will tie any unused I/O pins to Vcc. To do this, you can use the Arduino language "pinmode(pin, mode)" function.

Extrapolate for other integrated circuits

In general, you can follow the recommendation I have outlined above for other microcontrollers and their I/O pins.

There's one differentiator: for unused **output-only** pins it doesn't matter if you leave them floating.

An example of a an integrated circuit that contains output-only pins is the 595 shift register that I demonstrate in my relevant lectures in Arduino Step By Step Getting Serious. You can leave such pins unconnected.

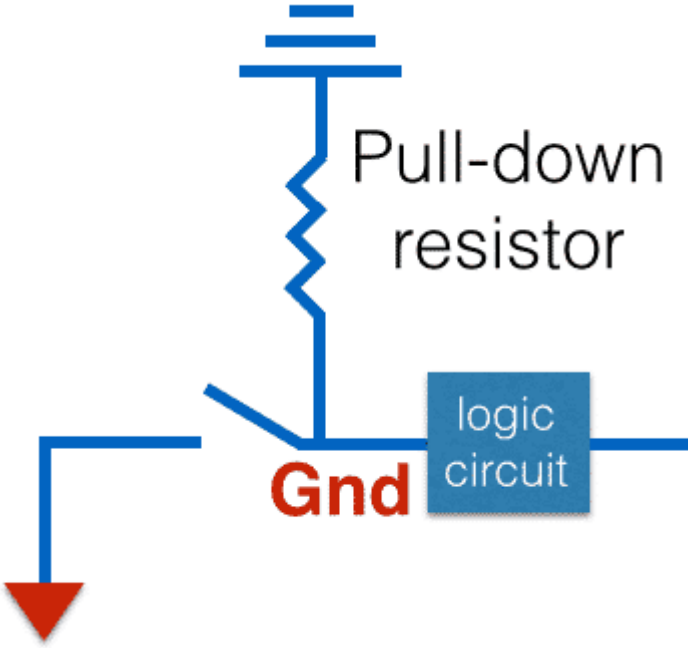
It's the programmable I/Os you need to think about more carefully. The guidelines from the datasheet should cover all practical cases.

1. Pull-up and pull-down resistors

Circuits guide series

Pull-Up & Pull-Down Resistors

People that are new to the Arduino and circuit design frequently ask is about pull-up and pull-down resistors. What is the purpose of a pull-up or pull-down resistor, and how does they work?



A question people new to circuit design frequently ask me is

about pull-up and pull-down resistors.What is the purpose of a pull-up or pull-down resistor, and how does it work?Imagine that you have a switch. One pin is connected to a 5V source, and the other to a logic circuit. A logic circuit is a component that expects voltage that lies within a specific range and values. For example, the Arduino Uno expects voltages between 0V and 5V to be applied on its pins. If the pin is digital, then the Arduino Uno will perceive as HIGH any voltage between 5V and 2V, and LOW any voltage between 0V and 0.8V.

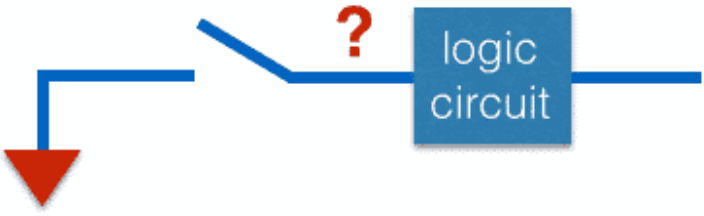
Case 1: Vin is determinable

Schematically, you would model a logic circuit connected to a voltage source like this:

The logic circuit has a single input pin on the left side of the blue box, and a single output on the right side. This pin is connected to a voltage source. The blue wire connects the pin to Vin, a voltage source that is greater than zero (it can also be negative). To depict a voltage source in a schematic, we often use the symbol of a triangle (there are other symbols, but this is a different topic). If the logic circuit is an Arduino, then Vin could be 5V coming out of a sensor.In other words, the logic circuit is sensing Vin at its input pin because the wire that connects the pin to ground is uninterrupted. In the schematic, we denote the voltage sensed by the logic circuit on the input pin as “Vin”.

Case 2: Vin is in-determinable (floating)

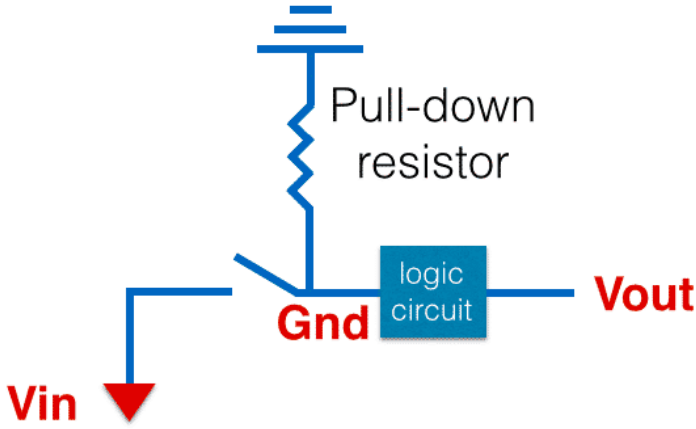
Now, let's add an open switch between ground and the logic circuit's input pin.What is the voltage that the logic circuit is sensing on its input pin?



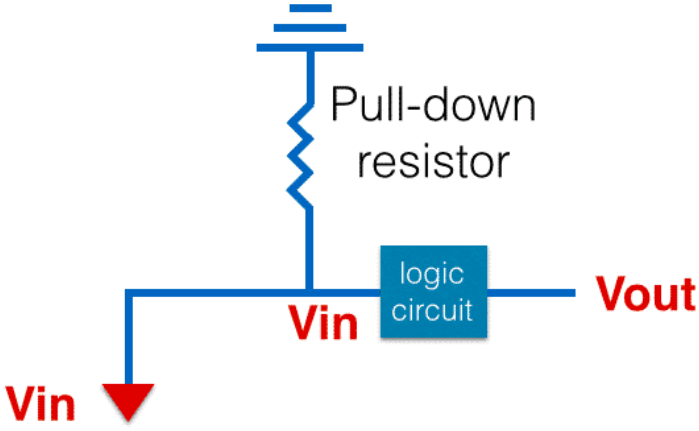
The answer is that we don't know. The input pin of the logic circuit is not connected to a voltage source with a specific value.A term that we use to describe this situation is that the pin is "floating". In other words, a pin that is not connected to a specific voltage is said to be floating.I'll repeat: the problem here is that the input pin of the logic circuit is not connected to a source of defined value, and logic circuits don't like this!How do we fix this problem?

Solution: Pull-up or pull-down resistors

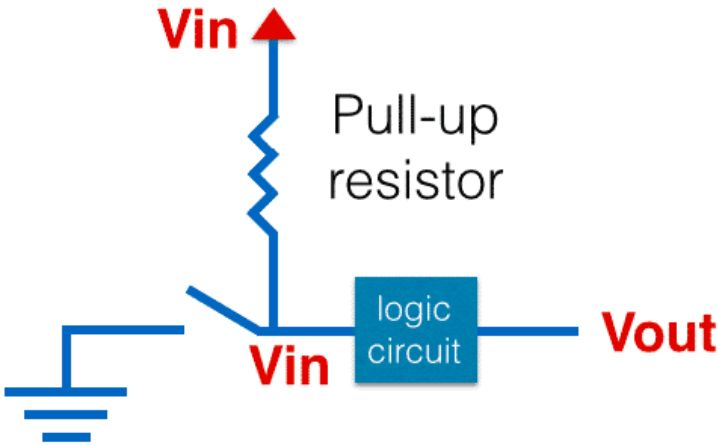
Have a look at the next schematic:



We fix this by using a large resistor (say, 10k or larger) to connect the logic circuit's input pin to ground. In electronics schematics, ground is often depicted with a symbol that consists of three parallel lines, like the one in the example above.By convention, the voltage level at ground is 0V. So, when the switch is open, the 0V level from GND is conveyed to the logic circuit's input pin. Because of the relatively large resistor between GND and the input pin, the current that flows through the wire is very small. We call this current "leakage current". What if we close the switch? Then, we'll have this:



Because the resistor is large, current from Vin will find it much easier to flow through the logic circuit rather than to ground via the resistor. And, because the connection between Vin and the logic circuit input has negligible resistance, Vin's value will be transferred to the input of the logic circuit.Let's recap: by attaching a pull-down resistor to our circuit, we ensure that the logic circuit's input will always have a defined value, and will be happy. You could inverse the circuit and connect the resistor to Vin instead of ground. Now, this resistor would be called "pull-up," because it would be pulling the input of the logic circuit to Vin (HIGH) when the switch was open.A pull-up resistor schematic would look like this:



I have just swapped the positive voltage (differential) to the top of the diagram and the ground to the bottom. When the switch is open, the input of the logic circuit is connected to Vin via the large resistance, so the voltage there is almost Vin. There will be a tiny current flowing so there will also be a minimal drop in voltage from Vin to the input of the logic circuit, but this is small enough to be able to accept an approximate value at the logic circuit as Vin, which is HIGH.

Arduino programming guide series

PWM and buffer overflow

What happens if you write a PWM value that is larger than the maximum value that the Arduino's analogWrite() function can accommodate? This is an interesting case of "buffer overflow".

The Arduino Uno is able to produce Pulse Width Modulation signals via pins 3, 5, 6, 9, 10, and 11. With PWM, you can approximate analog output programmatically and do things like fade an LED on and off or control the speed of a motor.

PWM values and register bits

In the Atmega328 (the chip that powers the Arduino Uno), the register that is used by the PWM function has a resolution of 8 bits. This gives you a total of 256 possible "analog" output levels, from 0 to 255.

If you attach an LED to a PWM-capable pin, you can drive it to

256 different brightness levels, from totally off (PWM value “0”) to totally on (PWM value “255”).

And if you attach a motor, you can drive it to 256 different speed levels.

Why 256? because the register contains 8 bits, and a binary number with 8 bits can be one of 2 ^ 8 = 256.

You can set a PWM value by using the analogWrite(pin, value) instruction.

So, **analogWrite(3, 125)** would set pin 3 to value 125.

How to overflow the PWM register

Now, here is where it gets interesting.

What happens if we set analogWrite to a value bigger than 255? Say, 256?

Let’s think about this for a minute.

If the PWM value is 255, the binary version is 11111111 (total is 8 bits) is stored in the PWM register (feel free to use this calculator for such binary to decimal conversions). A connected LED would light up in maximum brightness.

Let’s add 1 to the register, and make the PWM value 256.

The binary version of 256 is 0000000100000000 (total is 16 bits) since now we need two bytes to represent this value.

But, the Arduino (in reality, its Atmega328P chip) can only fit the first byte in its PWM register, the one in green.

The effect of PWM register overflow

The second byte will overflow and “disappear” (the red part).

So, what is actually stored in the PWM register is 00000000. This is decimal “0”, which means that your LED is turned off.

In other words, **analogWrite(3, 0)** and **analogWrite(3, 256)** would have the exact same effect on an LED or a motor.

Add another “1” to the register, and the PWM value now is 257.

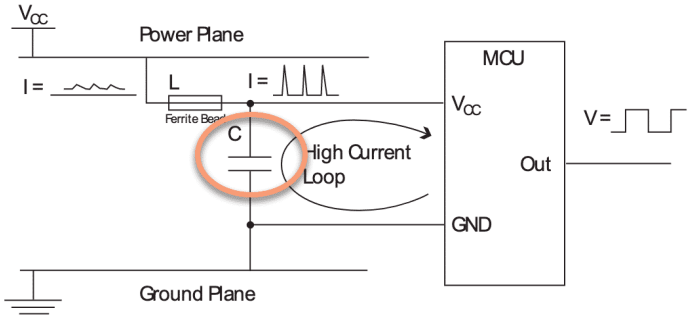
The binary version of 257 is 0000000100000001**.** The byte in green is stored in the PWM register, and the rest (in red) disappears. In the register now the decimal value “1” is stored.

The lesson to take home is that although you can set the PWM value in analogWrite to any decimal you like, only the first byte of this number will fit in the PWM register.

The rest will overflow and disappear.

What is a bypass/decoupling capacitor?

In electronic circuits, capacitors can be used as a source of energy and as a filter to help smooth out voltage spikes.



In the first instance, a capacitor can be used as a store of energy that can substitute a battery or other power supply when the circuit needs more current than what the main source can provide.

I have taken the schematic diagram below from Microchip's (see Figure 2-2 on page 6) that discusses microcontroller hardware design considerations. In microcontroller applications, it is important to ensure that the input voltage is kept as close as possible to the device's operating voltage.

Supercapacitors

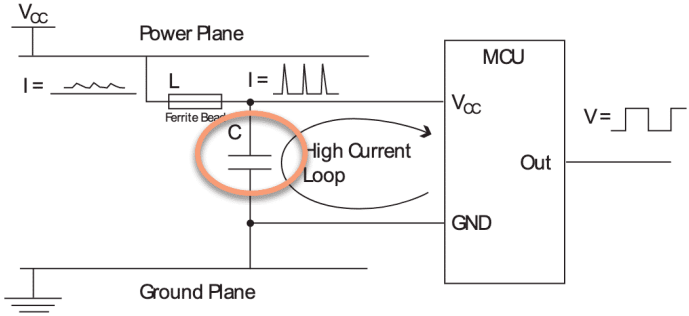
There is a particular type of capacitor, "supercapacitor" used in place of batteries in devices that work in environments that are too harsh for LiPo or other kinds of batteries.

A typical application of supercapacitors is in high-end dash-cams. These devices operate often directly exposed to the sun, and in extreme hot-cold climates. You will also find them in photographic flashes, photovoltaic systems, and laptop computers. You can find large supercapacitors in the power grid. In all cases, these capacitors are used to provide a stable power supply to the circuit.

You don't have to use a super-capacitor to achieve outcomes I described above. Depending on the circuit, a small electrolytic capacitor can stabilize the voltage fed into a microcontroller, like the Arduino.

Decoupling capacitors

I have borrowed the schematic below from the AN2519 AVR microcontroller hardware design consideration documents (page 6), from Microchip. I have marked the location of a decoupling capacitor with the orange circle, placed across the Vcc and GND pins.



Let's examine a practical example. In the Arduino Uno, the ATMega328P microcontroller has an operating voltage (Vcc) of 5V. If the Vcc voltage drops to 4.5V or below, even for a few milliseconds, there is a significant risk of a "brown-out". The result of this is that the MCU will be in an unstable state, and will not operate properly.

The decoupling capacitor will help to prevent a brown out by keeping the voltage stable. It does that by allowing the consumer circuit to draw energy from the capacitor, instead of the main power source.

As you can see in the example circuit above, the appropriate position for a decoupling capacitor in a microcontroller application is as close as possible to the Vcc and GND pins. A typical value for such a capacitor is 10F to 50F (usually tantalum or electrolytic). The exact value is not critical as having a capacitor around this range is much better than not having one.

Capacitors not only store energy, but they also filter out noise.

Which brings me to decoupling capacitors for filtering out noise.

Decoupling capacitors for noise filtering

In many DC applications, you can add a capacitor across the Vcc and GND lines to smooth out noise from the power supply or from other active components such as integrated circuits, motors or AC/DC power supplies.

When your circuit contains noisy components, it is important to add decoupling capacitors. These capacitors act not only as an energy store, but also as a filter for the harmful electrical noise.

You typically find very small-capacity capacitors here, around 0.1F. While they do store energy (as all capacitors do) their

primary purpose in the circuit is to filter out this unwanted electrical noise.

In digital circuits in particular, fast-switching components, such as an Atmega328P, can switch pins on and off (or their internal configuration) rapidly. Each pin on/off flip means that the current drawn from the power supply changes rapidly. These tiny changes in the current draw can cause small fluctuations in the voltages that runs through the circuit. This is noise. This noise is not predictable, and becomes worse as the number of active components in the circuit increases.

Too much of this noise in the circuit will cause it to stop working reliably.

The solution?

We can use a capacitor with appropriate characteristics for the purpose of filtering out (or dampening) the unwanted electrical noise.

The main characteristic of a capacitor that relates to its operation as a filter is its capacity. Connect such capacitor close to the microcontroller's GND and Vcc pins, to help smooth out those fluctuations.

For example, a 0.1F is good for dampening noise at frequencies around 100MHz.

Each circuit generates different kinds of noise, both in terms of frequencies and amplitude. While a generic value of bypass capacitors like 0.1F is a good place to start, it is often necessary to spend time with an oscilloscope to determine the best capacitor for the specific circuit.

When to use a decoupling capacitor?

In practice, when would you use a decoupling capacitor?

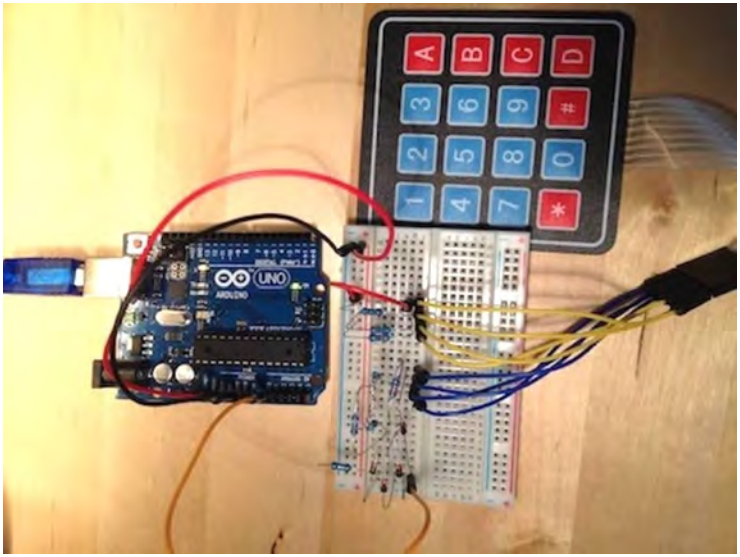
If your circuit contains a microcontroller or something similarly fast switching, then always include a small ceramic decoupling capacitor of around 0.1F connected very close to that fast switching component's Vcc and GND pins. The capacitor will take care of the noise.

If your circuit contains a component that occasionally draws a big burst of current, like a motor or a transmitter, then add a larger bypass capacitor (say, between 10F to 50F).

This Atmel guide recommends that you include decoupling capacitors to all Vcc-GND pairs and also provides the appropriate values and locations (on the PCB) for these capacitors.

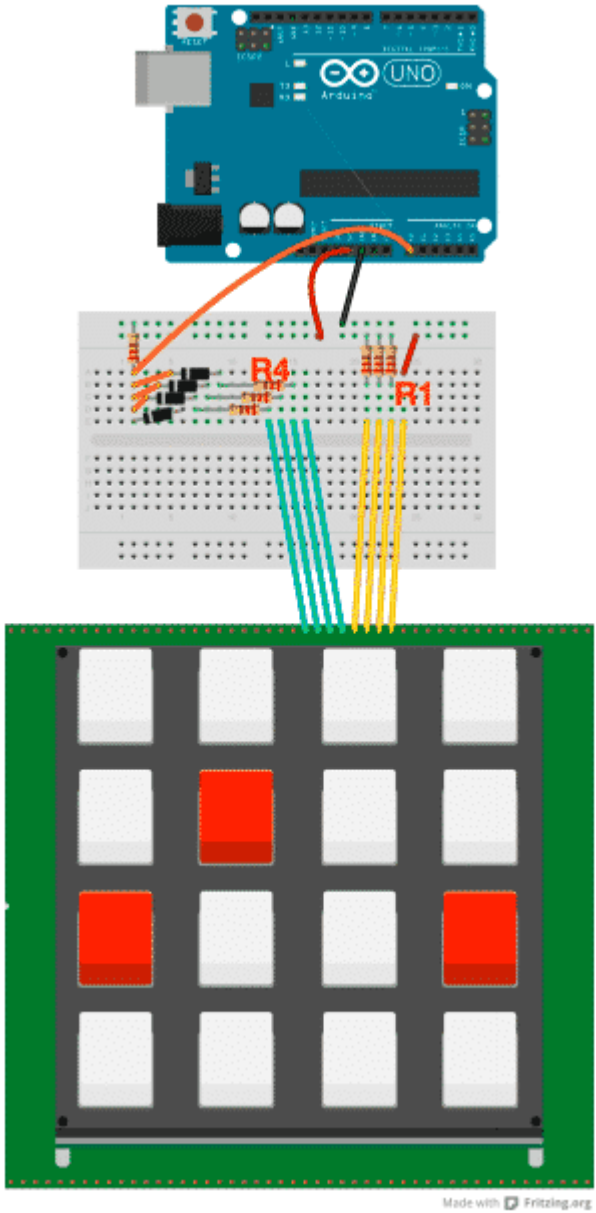
What is the purpose of the diodes in a keypad circuit?

A diode is a semiconducting device that allows the flow of electricity towards only one direction. Diodes are commonly used in applications where we want to prevent back-currents. One such application is in a keypad or keyboard circuit.



The 4×4 membrane keypad

When you wire a 4×4 (or similar) membrane keypad to your Arduino using a single analog data wire, like in the schematic below, most people readily understand the purpose of the resistors.


UNO
Arduino
R4
R1
Made with Fritzing.org

The resistors are used to create a voltage ladder, a circuit that produces multiple voltage reference points that can be measured by the Arduino. The sketch that is running in the Arduino can figure out which key was pressed by measuring the voltage on the sampling pin.But what about the diodes? What is their purpose?

Diodes in the 4×4 keypad circuit

If you can somehow ensure that only one key can be pressed at any given time (perhaps through mechanical means), the diodes are not needed. Try it out, and you should see that the keyboard works fine. But if you happen to press multiple keys on a matrix keypad (like when you hold down Shift-Cmd-S in a computer keyboard), an effect called "ghosting" comes into play.

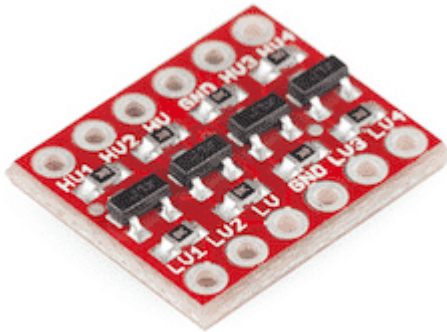
When ghosting occurs, then usually a key that wasn't pressed is read by the host (the Arduino, in our case).In many keyboard designs, diodes are added to deal with ghosting. If you are curious to know more about this, I have found two useful resources:

- First Wikipedia's entry on Rollover Key, describing how a keyboard can deal with multiple pressed keys.
- Second, a blog post on the matrix-configuration keyboard. You can scroll down to the section on ghosting.

Circuits guide series

Logic level shifting

Digital electronics operate at specific voltages. For example, the Arduino Uno operates at 5V. The Arduino Due operates at 3.3V. How can we connect electronic components that operate at different voltages?



Many Arduino boards, like the Arduino Uno, operate at 5V. This means that they are designed to receive and transmit signals at the 5V level.

Other Arduino boards, such as the Arduino Pro and the Arduino Due, operate at the 3.3V logic level. These boards are better suited to mobile applications because they are more efficient.

Many peripherals, like sensors, displays, and integrated circuits, are operate at the 3.3V level. This means that even if they implement a communications protocol that the Arduino

supports, you still have to consider how to connect them electrically.

Connecting a 5V device to a 3.3V device

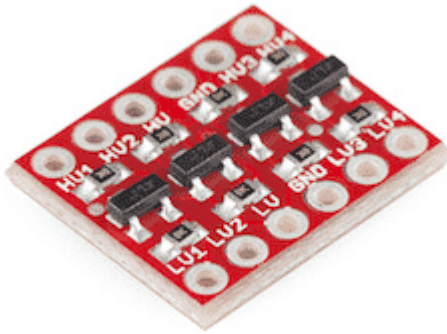
Connecting a 5V signal from an Arduino to a 3.3V input of a sensor like the LSM303 magnetometer integrated circuit will most likely damage the IC (unless it is tolerant of the higher voltage). Similarly, a 3.3V signal from the sensor to the Arduino Uno may or may not be read correctly. Any fluctuation below the 3V level will bring the signal too low to be correctly interpreted as HIGH by the Arduino.

For a 5V Arduino, the 3V is essential. As per the datasheet, any voltage above 0.6*Vcc is interpreted as HIGH. If your Vcc is precisely 5V, then any signal above 3V is HIGH. If your actual Vcc is a bit lower, then the cutoff voltage is affected accordingly.

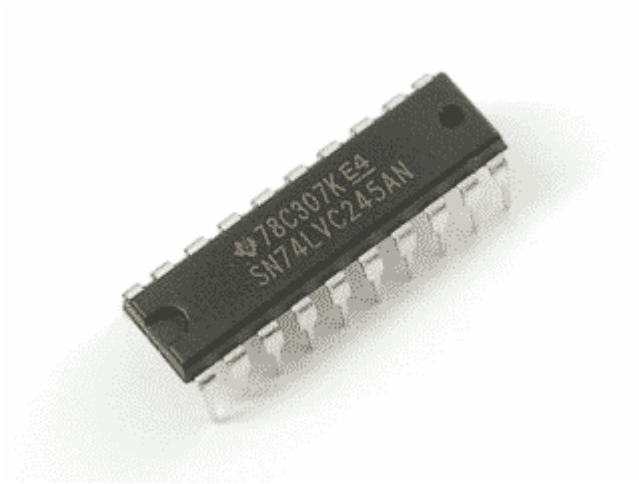
Logic level shifting methods

There are a few ways to deal with interfacing 3.3V and 5V devices.

A common method is the use of resistors configured as voltage divider. If you follow this path, you must be careful to accurately calculate the proper value of the resistors for the target voltage.I think that the best way to go about this is to use bi-directional level shifters. These are devices specifically designed for this objective. They come assembled as breakouts, like this one from Adafruit.



Or, you can go with a standalone integrated circuit, like the 74LVC245 from Texas Instruments (datasheet).



Both options have two rows of pins, one for connecting to the 3.3V device and the other for the 5V device. Both are also designed to work with signals at the 1.8V and 2.8V levels. They simplify the circuit layout and take care of the details. You can use these level shifters to connect the Arduino to 3.3V

computers like the Raspberry Pi and the Beaglebone Black. In general, these devices are more likely to be damaged if you connect their I/O pins to an incorrect voltage, so having a few logic level shifters around can save you money and time.

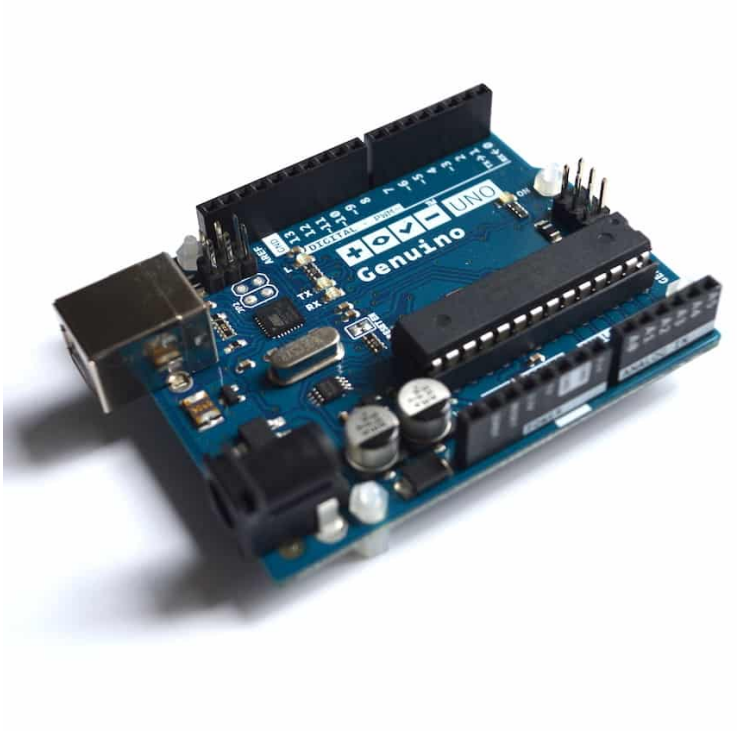
The Tech Explorations Subscription program

Subscribe and access all of our video courses immediately.

With a catalog of 25+ premium video courses (and growing), this subscription gives you an amazing resource to boost your learning.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“62516”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“62516”:{“val”:”var(–tcb-skin-color-0)”}},”gradients”:[]}}]}__CONFIG_colors_palette__
Learn more

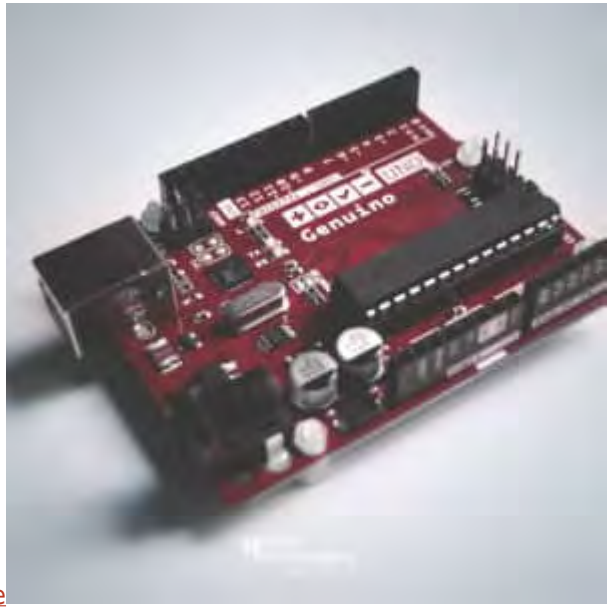
Jump to another article



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.



Learn more

Done with the basics? Looking for more advanced topics?

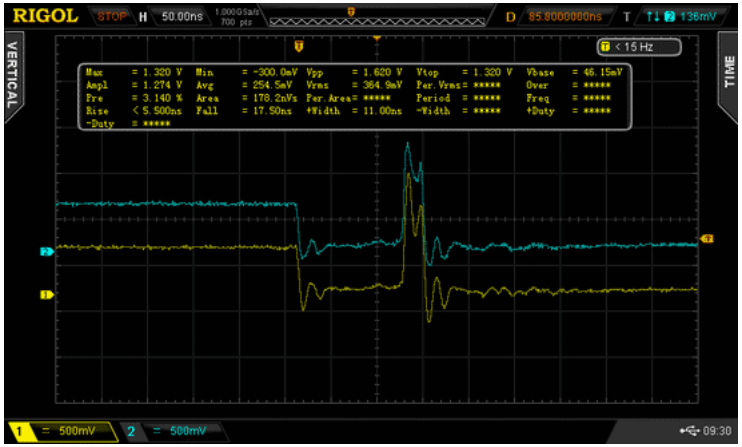
Arduino Step by Step Getting Serious is our comprehensive Arduino course for people ready to go to the next level.

Learn about Wifi, BLE and radio, motors (servo, DC and stepper motors with various controllers), LCD, OLED and TFT screens with buttons and touch interfaces, control large loads like relays and lights, and much much MUCH more.

Learn more

Why should you use a diode in a relay driver circuit?

A relay is commonly used to drive large electrical loads. With a relay, your Arduino can control large motors, LED strips, lights, etc. But without a simple diode, your circuit can be easily damaged.



You can easily make a relay driver circuit with a transistor, a current limiting resistor (to turn the transistor on and off), and a diode in parallel with the relay coil.

The diode is often the cause of confusion: why do we need

one?

I will explain...

Inductors

The coil of a relay device is that it is an inductor. An inductor will react to sudden changes in current by producing a large voltage across its ends.To energize the relay, you must turn on the transistor so that current will flow between its emitter and collector. To de-energise the relay, you must turn the transistor off, which will interrupt the current flow between the emitter and collector.

This sudden change in the current that flows through the coil of the relay, will cause the coil to react. The result of this reaction is a large voltage across its leads.

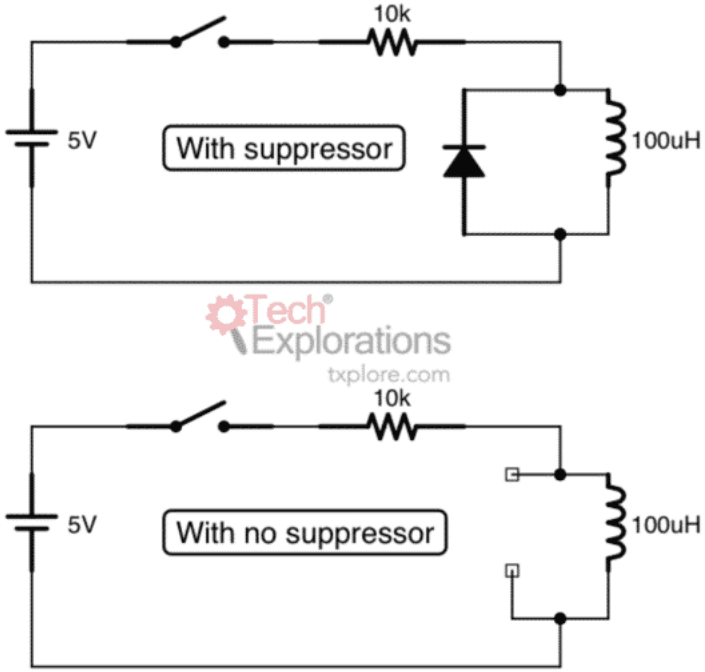
Back-current suppression diode experiment

The diode is there to suppress this voltage so that it cannot damage any components around the relay, such as the transistor and the external battery supply (but also the controlling logic circuit, such as the Arduino).To understand what is going on, I did a quick experiment on my oscilloscope.

I used two versions of a simple circuit with a coil, button, power supply and current limiting resistor (pictured below).

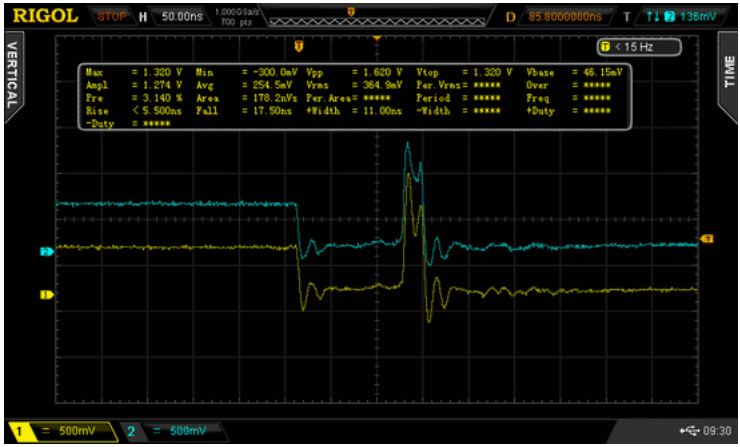
In the first version, I use a diode as a suppressor for the voltage spike in the coil.

In the second version, I did not use the diode.Here are the two circuits:

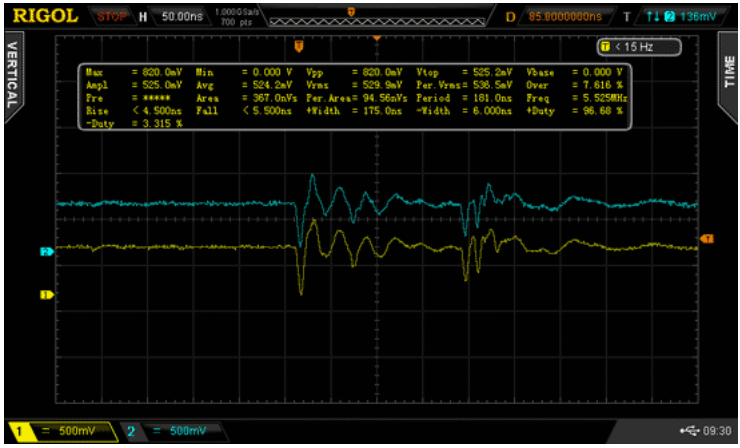


I used a channel (blue) on my scope for the trigger, and the other (yellow) to capture the voltage around the coil. The Yellow line is the coil voltage, and the Blue is the voltage on the switch (the trigger). You can ignore the blue line in the screenshots below.

Here is what the capture looks like without the diode:



The capture with the diode:



Look at the voltages, especially the Vpp one (Vpp: Voltage Peak-to-Peak).

Which one is larger? The one with, or without the diode?

If you look at the ending part of the waveform for the yellow line, for both circuits the voltage will eventually stabilize at the same value. Therefore the longer term (after around 350ns) effect of the diode is negligible (if any).

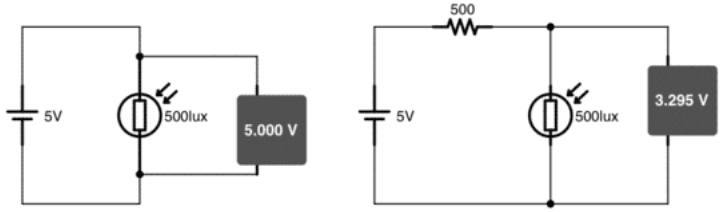
But just after the moment I press the button and energize the relay coil, the first line (no diode) shows a much bigger (in this case, roughly double) peak-to-peak voltage compared to the second line (with diode).As you can see, the addition of the diode in a circuit that contains any kind of coil (like in a relay or DC motor) will significantly dumpen back currents by limiting the effect of the voltage accross the ends of the coil.

6. Why use a voltage divider with a photoresistor?

Circuits guide series

Why use a voltage divider with a photoresistor?

A common question is “why not connect a photoresistor directly to one of Arduino’s analog pins, instead of connecting it via a voltage divider?”.



Voltage dividers tend to confuse people that are new electronics.

Let’s take a photoresistor, as an example.

A common question is “why not connect a photoresistor directly to one of Arduino’s analog pins, instead of connecting it via a voltage divider?”

It is a fair question.

The voltage that the Arduino measures on its analog pin depends on the impedance (resistance) of the photoresistor.

Since the impedance of the photoresistor depends on the light intensity, we should be able to use the direct connection instead of the voltage divider.

But, it doesn't work like that.

Experiment

If you have a multimeter handy, try a simple experiment.

Connect the pins of your photoresistor to the electrodes of the multimeter. Set the multiumeter to measure resistance (ohm-meter). This will allow you to measure the impedance (resistance) of a photoresistor.

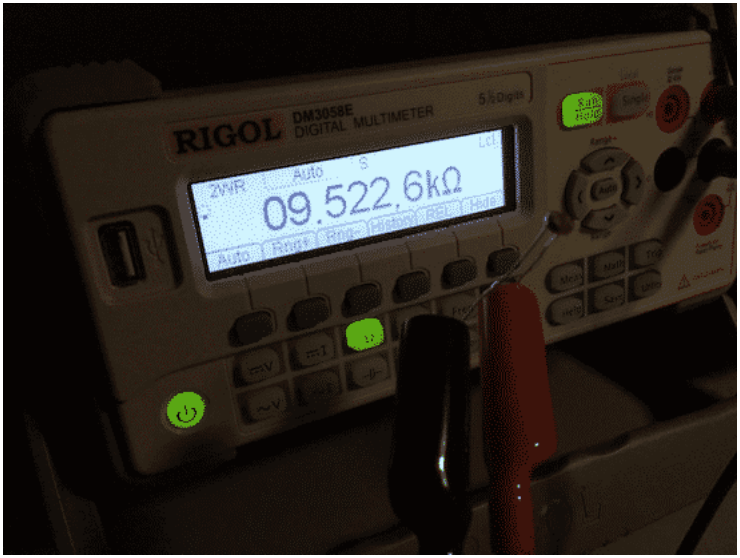
Take a few measurements, under different light conditions. You will see that the impedance varies, but is always very high. For most common photoresistors, the measured resistance can go from 100K to 1M.

Because of this high impedance, if you connect the photoresistor between, say, the Arduino 5V pin and A0, the current that will flow through this component will be very small. As a result, the voltage drop on the photoresistor will be barely noticeable by the Arduino.

On pin A0, the Arduino will measure close to 5V no matter how much light is hitting the photoresistor.

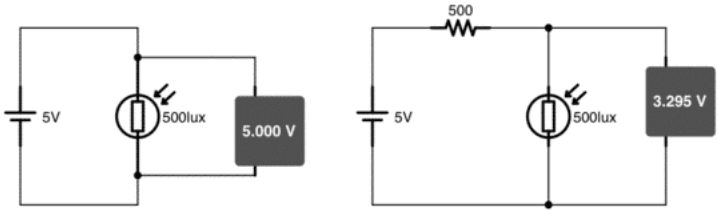
That's not very useful!

Here is a measurement of the impedance of a photoresistor, when directed towards a light source. It is around 10K, a value that is more suitable to a pull-up or pull-down resistor. At 5V, you will not get much current from this device (just ~0.0005A in this case).



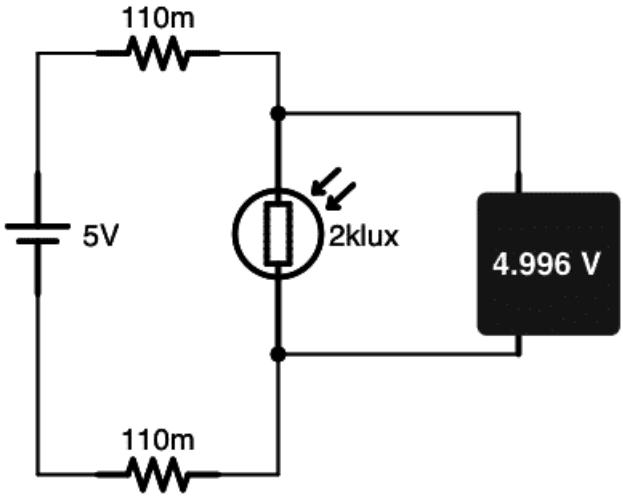
Simulation

I have used my circuit simulator to compare the two ways of connecting a photoresistor to the Arduino. Without a voltage divider (left), and with a voltage divider (right). The simulator allows me to test the photoresistor at arbitrary lux levels.



At three different lux levels on the photoresistor(around 1klux is the light intensity in a low-lit room), the measured voltage on the left circuit didn't budge from the 5V. On the right side, with the voltage divider fixed resistor, we received three different readings.

As you can see, with the help of a suitable voltage divider, the photoresistor becomes a useful sensor for light intensity.On the left side, you can also see that regardless of the current that flows through the photoresistor, the voltmeter measures the voltage across the DC power supply, which is constant at 5V. There is simply no other way to connect the photoresistor so that it can operate on its own and still provide a meaningful reading in proportion to the intensity of the light that hits it. Notice that this is an ideal circuit, without any impedance in the wires. In real life, there is impedance in the wires, and the circuit is more like this:



The impedance in the wires is around 0.11O, for a 10cm jumper wire. This could produce a voltage reading of 4.998V in the voltmeter on the circuit on the left. And that reading would not vary much as the photoresistor's impedance changes since that impedance is so large in comparison.

Conclusion

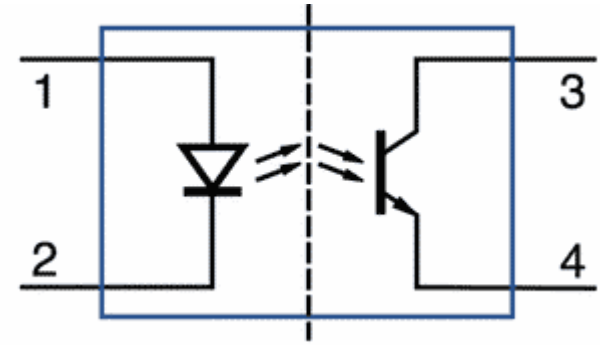
By using a fixed resistor that is much smaller than the minimum impedance of the sensor, we are able to create a voltage drop that it depends a lot more on the smaller component, but still influenced by the larger components (larger, in terms of impedance).

Another benefit is this: because photoresistors come from different manufacturers with different characteristics, by using a fixed resistor in a voltage divider configuration we can reduce the effect of these variances. Therefore, our circuit becomes less dependent on the photoresistor's peculiarities.

Circuits guide series

Optoisolator: a simple way to electrically separate parts of a circuit

An optoisolator is a device that allows two parts of a circuit to communicate using photon instead of electrons. This way, you can electrically isolate two parts of a circuit.



For those cases where you need complete isolation between two circuits, while provisioning for basic on/off type interaction, you can use an optoisolator (also known as “optocoupler”).

Disadvantages of the relay

As an example, consider the relay. With a relay, you can control a high voltage/current circuit using a low-power signal generated from a low-power circuit.

A relay works well, but it is a noisy, electro-mechanical component. Apart from the clicking sound that is the result of the solenoid's mechanical motion that connects two of the relay's conductive pads, you also have to consider the short finite lifetime of the component.

All this means that any gadget that contains an electro-mechanical relay is subject to the wear and tear of the mechanical parts, the relatively low on/off switching speed, and the electrical noise created by the coil every time the current changes.

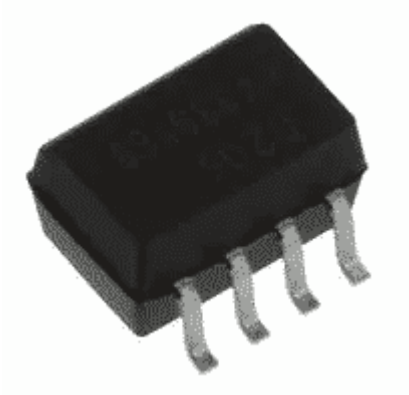
Relays are also bulky when compared to most other parts of an electronic device.

Advantages of the optoisolator

The optoisolator deals with all of these problems.

Typically, an optoisolator comes in the form of a small integrated circuit that looks like any other IC on your PCB.

Below is an example: the ILD205T from Vishay.



The ILD205T optocoupler from Vishay

Optoisolator integrated circuits often contain multiple channels. This means that you can drive more than one high-powered circuits from the same optoisolator. They provide complete electrical isolation between the two circuits since the operating principle is light, rather than electromagnetism through a coil.

Anatomy of the optoisolator

Inside an optoisolator IC you will find an LED that generates light at near-infrared, and a photosensor that is sensitive to this light. In between those two components is a closed optical channel that routes the light from the LED to the sensor. There are also electronics that are used to electrically interface the LED and sensor with their respective sides of the external circuits.

In the diagram below (from Wikipedia), you can see the input pins 1 and two which drive the integrated LED, and the output pins 3 and 4 that are driven by a photodiode or similar photo-sensitive component.

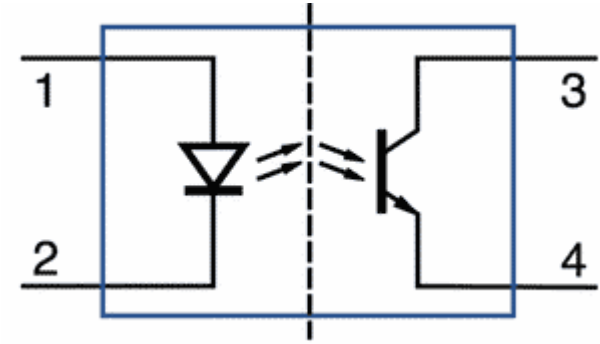


Image courtesy of Wikipedia

Concerning tolerances, optoisolators can be impressive. A small, cheap (~$4), common optoisolator like the ILD205T comes in an SMD SOIC-8 package, contains two channels, require just 10mA (and can tolerate up to 100mA) to turn the embedded LED on at 6V. At the output side, it can drive loads up to 70V. The opto-isolator can deal with differential voltages of up to 4000V.

Here is the datasheet for the ILD205T, a commonly used optoisolator.

Optoisolator applications

What can you do with an optoisolator that you can't do efficiently (or at all) with a relay? Driving a DC motor without a motor driver, PWD of high-powered LEDs are a couple of examples. Another good use of this device is for protecting your Arduino. Anything that involves motors, mains power, and often radios, could benefit from optoisolators.

The Tech Explorations Subscription

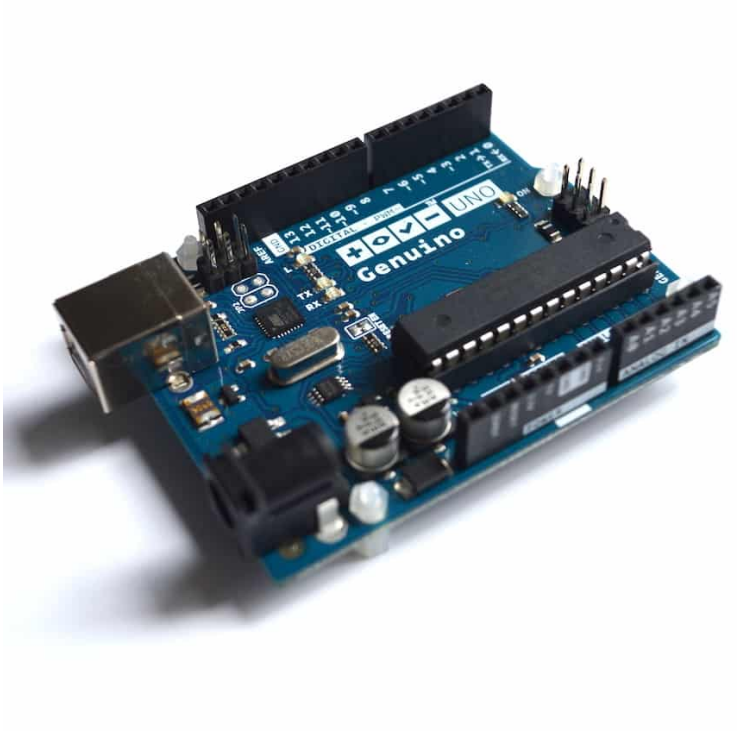
program

Subscribe and access all of our video courses immediately.

With a catalog of 25+ premium video courses (and growing), this subscription gives you an amazing resource to boost your learning.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“62516”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“62516”:{“val”:”var(–tcb-skin-color-0)”}},”gradients”:[]}}]}__CONFIG_colors_palette__
Learn more

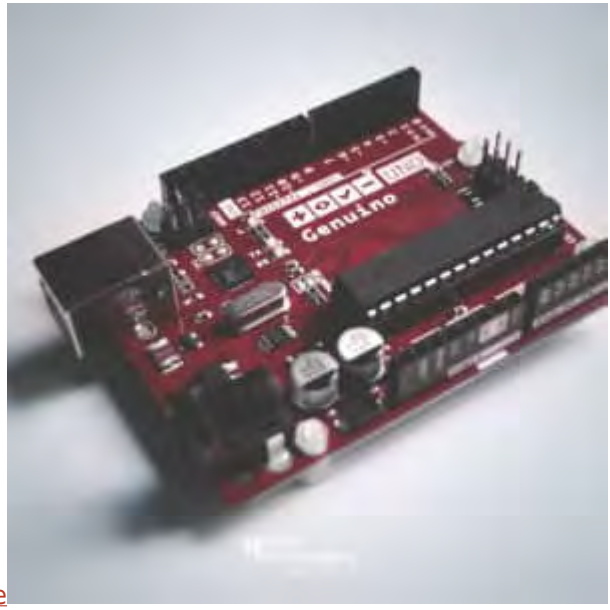
Jump to another article



New to the Arduino?

Arduino Step by Step Getting Started is our most popular course for beginners.

This course is packed with high-quality video, mini-projects, and everything you need to learn Arduino from the ground up. We'll help you get started and at every step with top-notch instruction and our super-helpful course discussion space.



Learn more

Done with the basics? Looking for more advanced topics?

Arduino Step by Step Getting Serious is our comprehensive Arduino course for people ready to go to the next level.

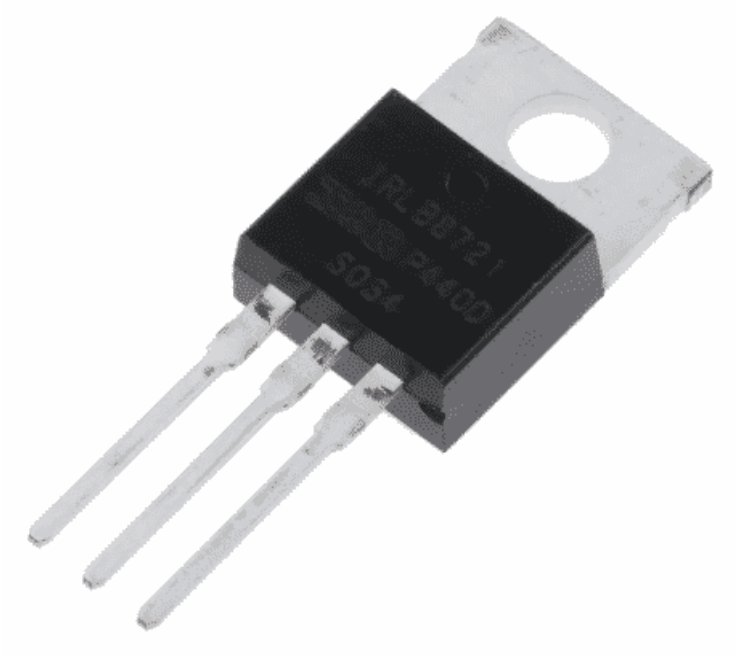
Learn about Wifi, BLE and radio, motors (servo, DC and stepper motors with various controllers), LCD, OLED and TFT screens with buttons and touch interfaces, control large loads like relays and lights, and much much MUCH more.

Learn more

Circuits guide series

Use MOSFETs to drive large(ish) loads

MOSFET transistors are the building block of modern electronics. Between 1960 and 2018, to drive high-power electronics, 13 sextillion (1.3×1022) MOSFETs have been manufactured. Apart from using them to do calculations in CPUs, they are excellent for driving large electrical loads.



Many typical Arduino applications involve driving devices that

require more power than what the Arduino itself can provide through its pins. DC motors, lights and solenoids are example devices that need lots of power to operate. This power translates to higher voltages, higher currents, or both at the same time. As the Arduino cannot provide the required power, we use specialized devices like relays and transistors. These devices are used as interfaces between low-power controller circuits, like the Arduino, to higher power controller circuits, like electric motors, LED strip lights, sirens, strobe lights, etc.In thisarticle I will discuss the MOSFET device.

What is a MOSFET

The acronym “MOSFET” stands for
Metal–Oxide–Semiconductor Field-Effect Transistor.

Yes: it is a type of transistor, but instead of only having three terminals, like a typical field-effect transistor (Base, Collector, Emitter) it has four: source (S), gate (G), drain (D), and body (B).

In most cases though, the B and the S are connected together (shorted) so we end up with MOSFET packages that expose only three terminals: source (S), gate (G), and drain (D).

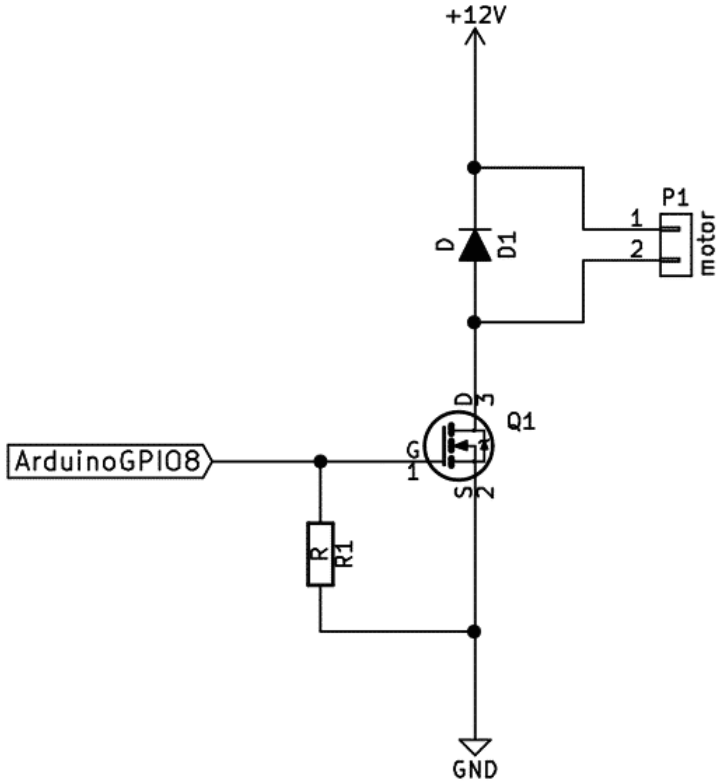
Practically, a MOSFET has an advantage over a “normal” transistor or relay because it requires very little current to operate it (turn it on or off). Less than 1mA at the gate will do the job. This is much less to the current needed by the common field-effect transistor 2N2222 (around 5mA).

Despite the tiny control current, a MOSFET can deliver very high currents, at least 10A and up to 60A for a common device like the IRLB8721PbF.Another advantage of a MOSFET over a relay is its switching speed. It can turn on and off within nanoseconds. The IRLB8721PbF, for example, can switch on within around 100nsecs, which makes it a great option if you want to drive a power LED or motor using pulse-width modulation.And the really nice thing is that you can use a

MOSFET almost as a drop-in replacement for a normal field-effect transistor.

Example use of a MOSFET

Here is an example of use:



In this schematic, a MOSFET transistor is used to turn on the connected motor. Because the motor contains a coil, a diode is connected in parallel to protect from reverse voltages (see relevant article). You should use the same setup for any load that contains a coil, like a relay, solenoid or motor. If instead of a motor you used a high-power LED, you can omit the diode.

The gate of the MOSFET is connected to one of the Arduino's GPIOs. This could be a 5V or a 3.3V Arduino. There is also a pull-down resistor that connects the gate to ground. This resistor is used in the case that the transistor's source pin is "floating". This can happen, for example, if the Arduino is turn off, or the source pin is not connected to anything and therefore its voltage is undefined.In summary, a MOSFET transistor is an excellent choice for controlling a relatively large load using 3.3V or 5V logic, such as Arduinos, and Raspberry Pis.