Extracted from:

Arduino

A Quick-Start Guide

This PDF file contains pages extracted from *Arduino*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

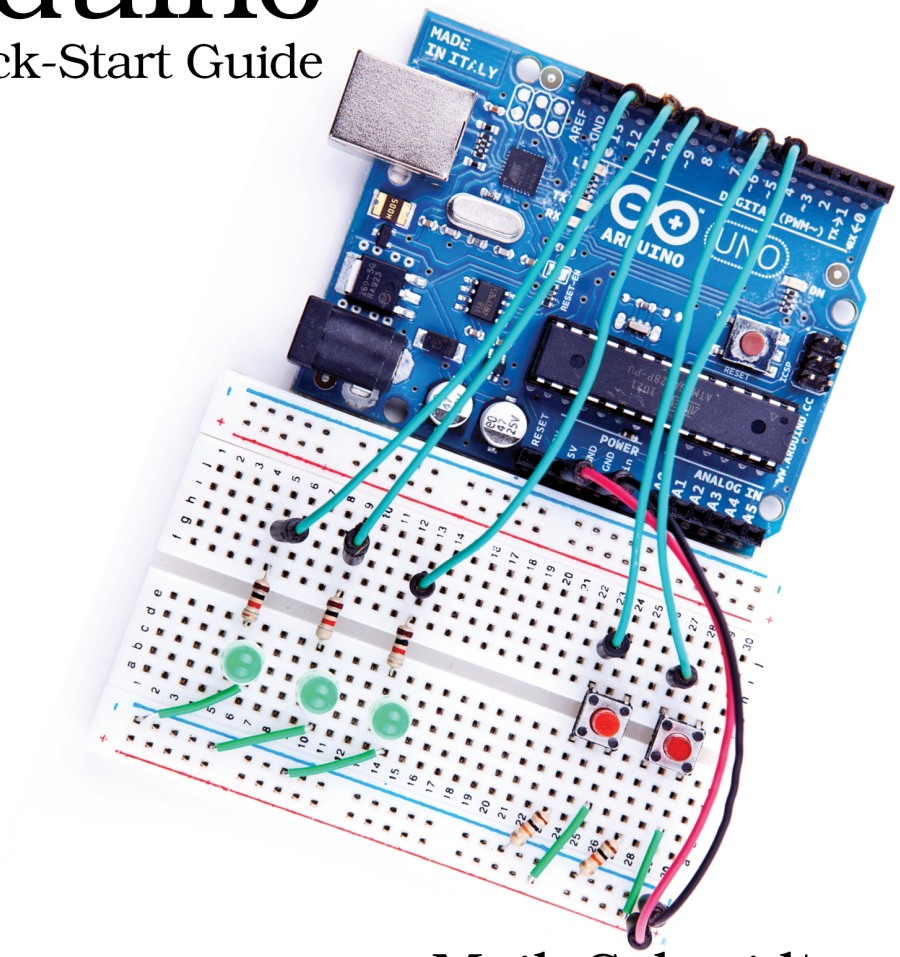
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

For Arduino 1.0

Arduino

A Quick-Start Guide



Maik Schmidt

Edited by Susannah Davidson Pfalzer

It's astonishing how quickly we get used to new technologies. A decade ago, not many people would have imagined that we would use devices someday to unobtrusively follow our movements. Today it's absolutely normal for us to physically turn our smartphones when we want to change from portrait to landscape view. Even small children intuitively know how to use motion-sensing controllers for video game consoles such as Nintendo's Wii. You can build your own motion-sensing devices using an Arduino, and in this chapter you'll learn how.

We'll work with one of the most widespread motion-sensing devices: the *accelerometer*. Accelerometers detect movement in all directions—they notice if you move them up, down, forward, backward, to the left, or to the right. Many popular gadgets such as the iPhone and the Nintendo Wii controllers contain accelerometers. That's why accelerometers are cheap.

Both fun and serious projects can benefit from accelerometers. When working with your computer, you certainly think of projects such as game controllers or other input control devices. But you can also use them when exercising or to control a real-life marble maze. You can also use them to measure acceleration more or less indirectly, such as in a car.

You will learn how to interpret accelerometer data correctly and how to get the most accurate results. Then you'll use an accelerometer to build a motion-sensing game controller, and you'll implement a game that uses it.

6.1 What You Need

1. A half-size breadboard or—even better—an Arduino Prototyping shield with a tiny breadboard
2. An ADXL335 accelerometer
3. A pushbutton
4. A 10kΩ resistor
5. Some wires
6. An Arduino board such as the Uno, Duemilanove, or Diecimila
7. A USB cable to connect the Arduino to your computer
8. A 6 pin 0.1" standard header

See Figure 50, *All the parts you need in this chapter*, on page 6.

• Click HERE to purchase this book now. discuss

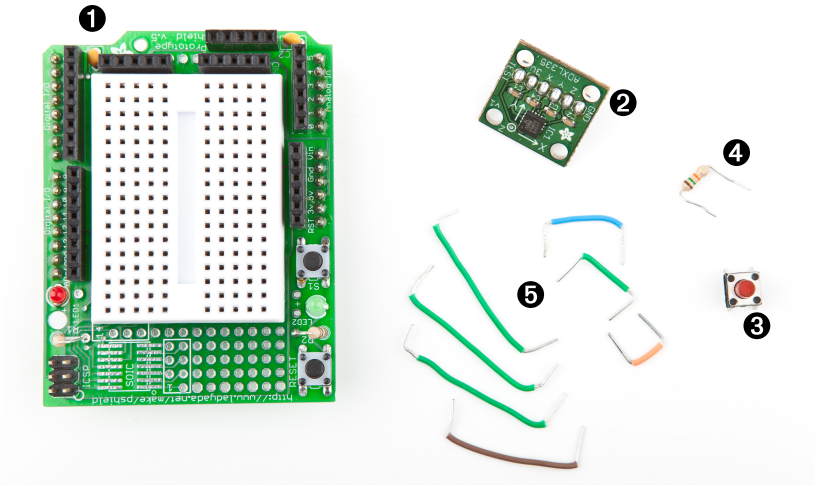


Figure 50—All the parts you need in this chapter

6.2 Wiring Up the Accelerometer

There are many different accelerometers, differing mainly in the number of spacial axes they support (usually two or three). We use the ADXL335 from Analog Devices—it's easy to use and widely available.[1]

In this section, we'll connect the ADXL335 to the Arduino and create a small demo program showing the raw data the sensor delivers. At that point, we will have a quick look at the sensor's specification and interpret the data.

In Figure 51, *An ADXL335 sensor on a breakout board*, on page 7, you see a breakout board containing an ADXL335 sensor on the right. The sensor is the small black integrated circuit (IC), and the rest is just a carrier to allow connections. On the top, you see a 6 pin 0.1" standard header. The sensor has six connectors labeled GND, Z, Y, X, 3V, and TEST. To use the sensor on a breadboard, solder the standard header to the connectors. This not only makes it easier to attach the sensor to a breadboard but also stabilizes the sensor, so it does not move accidentally. You can see the result on the left side of the photo (note that the breakout board on the left is not the same as on the right, but it's very similar). Don't worry if you've never soldered before. In Section A1.2, *Learning How to Solder*, on page ?, you can learn how to do it.

1. http:/www.analog.com/en/sensors/inertial-sensors/adxl335/products/product.html

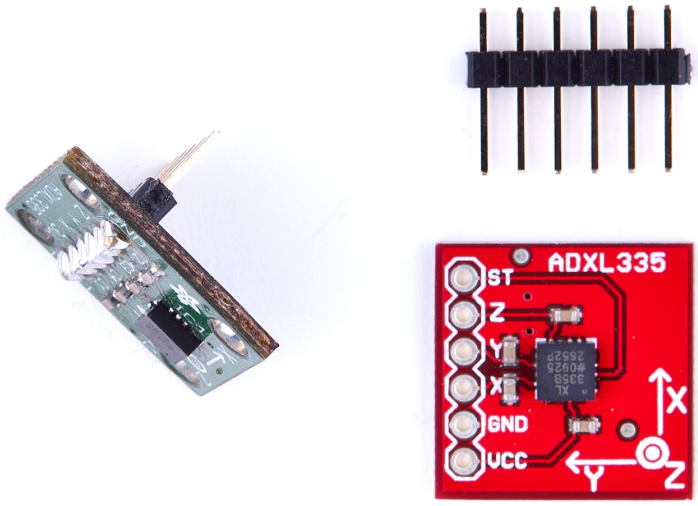


Figure 51—An ADXL335 sensor on a breakout board

You can ignore the connector labeled TEST, and the meaning of the remaining connectors should be obvious. To power the sensor, connect GND to the Arduino's ground pin and 3V to the Arduino's 3.3 volts power supply. X, Y, and Z will then deliver acceleration data for the x-, y-, and z-axes.

Like the TMP36 temperature sensor we used in Section 5.4, *Increasing Precision Using a Temperature Sensor*, on page ?, the ADXL335 is an analog device: it delivers results as voltages that have to be converted into acceleration values. So, the X, Y, and Z connectors have to be connected to three analog pins on the Arduino. We connect Z to analog pin 0, Y to analog pin 1, and X to analog pin 2 (see Figure 52, *How to connect an ADXL335 sensor to an Arduino*, on page 8, and double-check the pin labels on the breakout board you're using!).

Now that we've connected the ADXL335 to the Arduino, let's use it.

6.3 Bringing Your Accelerometer to Life

A pragmatic strategy to get familiar with a new device is to hook it up and see what data it delivers. The following program reads input values for all three axes and outputs them to the serial port:

Download Arduino_1_0/MotionSensor/SensorTest/SensorTest.ino

```
const unsigned int X_AXIS_PIN = 2;
const unsigned int Y_AXIS_PIN = 1;
```

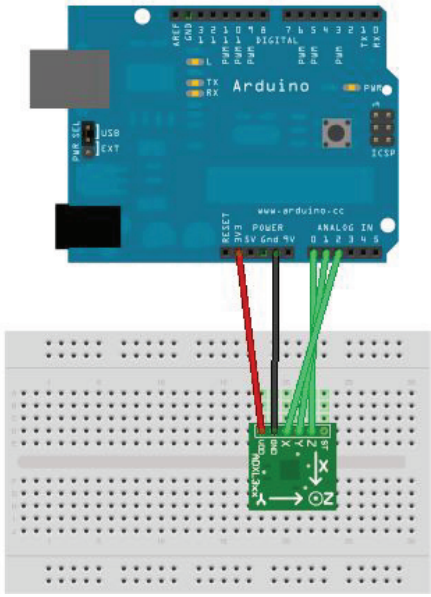


Figure 52—How to connect an ADXL335 sensor to an Arduino

```
const unsigned int Z_AXIS_PIN = 0;
const unsigned int BAUD_RATE = 9600;

void setup() {
  Serial.begin(BAUD_RATE);
}

void loop() {
  Serial.print(analogRead(X_AXIS_PIN));
  Serial.print(" ");
  Serial.print(analogRead(Y_AXIS_PIN));
  Serial.print(" ");
  Serial.println(analogRead(Z_AXIS_PIN));
  delay(100);
}
```

Our test program is as simple as it can be. We define constants for the three analog pins and initialize the serial port in the setup() function. Note that we did not set the analog pins to INPUT explicitly, because that's the default anyway.

In the loop() function, we constantly output the values we read from the analog pins to the serial port. Open the serial monitor, and move the sensor around a bit—tilt it around the different axes. You should see an output similar to the following:

```
344 331 390
364 276 352
388 286 287
398 314 286
376 332 289
370 336 301
379 338 281
```

These values represent the data we get for the x-, y-, and z-axes. When you move the sensor only around the x-axis, for example, you can see that the first value changes accordingly. In the next section, we'll take a closer look at these values.

6.4 Finding and Polishing Edge Values

The physical world often is far from being perfect. That's especially true for the data many sensors emit, and accelerometers are no exception. They slightly vary in the minimum and maximum values they generate, and they often jitter a bit. They might change their output values even though you haven't moved them, or they might not change their output values correctly. In this section, we'll determine the sensor's minimum and maximum values, and we'll flatten the jitter.

Finding the edge values of the sensor is easy, but it cannot be easily automated. You have to constantly read the sensor's output while moving it. Here's a program that does the job:

Download Arduino_1_0/MotionSensor/SensorValues/SensorValues.ino

```
const unsigned int X_AXIS_PIN = 2;
const unsigned int Y_AXIS_PIN = 1;
const unsigned int Z_AXIS_PIN = 0;
const unsigned int BAUD_RATE = 9600;

int min_x, min_y, min_z;
int max_x, max_y, max_z;

void setup() {
  Serial.begin(BAUD_RATE);
  min_x = min_y = min_z = 1000;
  max_x = max_y = max_z = -1000;
}

void loop() {
  const int x = analogRead(X_AXIS_PIN);
  const int y = analogRead(Y_AXIS_PIN);
  const int z = analogRead(Z_AXIS_PIN);
```

```
  min_x = min(x, min_x); max_x = max(x, max_x);
  min_y = min(y, min_y); max_y = max(y, max_y);
  min_z = min(z, min_z); max_z = max(z, max_z);

  Serial.print("x(");
  Serial.print(min_x);
  Serial.print("/");
  Serial.print(max_x);
  Serial.print("), y(");
  Serial.print(min_y);
  Serial.print("/");
  Serial.print(max_y);
  Serial.print("), z(");
  Serial.print(min_z);
  Serial.print("/");
  Serial.print(max_z);
  Serial.println(")");
}
```

We declare variables for the minimum and maximum values of all three axes, and we initialize them with numbers that are definitely out of the sensor's range (-1000 and 1000). In the loop() function, we permanently measure the acceleration of all three axes and adjust the minimum and maximum values accordingly.

Compile and upload the sketch, then move the breadboard with the sensor in all directions, and then tilt it around all axes. Move it slowly, move it fast, tilt it slowly, and tilt it fast. Use long wires, and be careful when moving and rotating the breadboard so you do not accidentally loosen a connection.

After a short while the minimum and maximum values will stabilize, and you should get output like this:

```
x(247/649), y(253/647), z(278/658)
```

Write down these values, because we need them later, and you'll probably need them when you do your own sensor experiments.

Now let's see how to get rid of the jitter. In principle, it is simple. Instead of returning the acceleration data immediately, we collect the last readings and return their average. This way, small changes will be ironed out. The code looks as follows:

Download Arduino_1_0/MotionSensor/Buffering/Buffering.ino

```
const unsigned int X_AXIS_PIN = 2;
const unsigned int Y_AXIS_PIN = 1;
const unsigned int Z_AXIS_PIN = 0;
const unsigned int NUM_AXES = 3;
const unsigned int PINS[NUM_AXES] = {
```

```
  X_AXIS_PIN, Y_AXIS_PIN, Z_AXIS_PIN
};
const unsigned int BUFFER_SIZE = 16;
const unsigned int BAUD_RATE = 9600;
int buffer[NUM_AXES][BUFFER_SIZE];
int buffer_pos[NUM_AXES] = { 0 };

void setup() {
  Serial.begin(BAUD_RATE);
}

int get_axis(const int axis) {
  delay(1);
  buffer[axis][buffer_pos[axis]] = analogRead(PINS[axis]);
  buffer_pos[axis] = (buffer_pos[axis] + 1) % BUFFER_SIZE;
  long sum = 0;
  for (unsigned int i = 0; i < BUFFER_SIZE; i++)
    sum += buffer[axis][i];
  return round(sum / BUFFER_SIZE);
}

int get_x() { return get_axis(0); }
int get_y() { return get_axis(1); }
int get_z() { return get_axis(2); }
void loop() {
  Serial.print(get_x());
  Serial.print(" ");
  Serial.print(get_y());
  Serial.print(" ");
  Serial.println(get_z());
}
```

As usual, we define some constants for the pins we use first. This time, we also define a constant named `NUM_AXES` that contains the amount of axes we are measuring. We also have an array named `PINS` that contains a list of the pins we use. This helps us keep our code more generic later.

In line 10, we declare buffers for all axes. They will be filled with the sensor data we measure, so we can calculate average values when we need them. We have to store our current position in each buffer, so in line 11, we define an array of buffer positions.

`setup()` only initializes the serial port, and the real action takes place in the `get_axis()` function. It starts with a small delay to give the Arduino some time to switch between analog pins; otherwise, you might get bad data. Then it reads the acceleration for the axis we have passed and stores it at the current buffer position belonging to the axis. It increases the buffer position and sets

it back to zero when the end of the buffer has been reached. Finally, we return the average value of the data we have gathered so far for the current axis.

That's the whole trick. To see its effect, leave the sensor untouched on your desk, and run the program with different buffer sizes. If you do not touch the sensor, you would not expect the program's output to change. But if you set `BUFFER_SIZE` to 1, you will quickly see small changes. They will disappear as soon as the buffer is big enough.

The acceleration data we measure now is sufficiently accurate, and we can finally build a game controller that will not annoy users because of unexpected movements.

6.5 Building Your Own Game Controller

To build a full-blown game controller, we only need to add a button to our breadboard. In Figure 53, *Game controller with accelerometer and pushbutton*, on page 13, you can see how to do it (please, double-check the pin labels on your breakout board!).

That's how it looks inside a typical modern game controller. We will not build a fancy housing for the controller, but we still should think about ergonomics for a moment. Our current breadboard solution is rather fragile, and you cannot really wave around the board when it's connected to the Arduino. Sooner or later you will disconnect some wires, and the controller will stop working.

To solve this problem, you could try to attach the breadboard to the Arduino using some rubber bands. That works, but it does not look very pretty, and it's still hard to handle.

A much better solution is to use an Arduino Prototyping shield (see Figure 54, *An Arduino prototyping shield*, on page 13). It is a pluggable breadboard that lets you quickly build circuit prototypes. The breadboard is surrounded by the Arduino's pins, so you no longer need long wires. Shields are a great way to enhance an Arduino's capabilities, and you can get shields for many different purposes such as adding Ethernet, sound, displays, and so on.[2]

Using the Proto Shield our game controller looks as in Figure 55, *The complete game controller on a Proto shield*, on page 15. Neat, eh?

2. At http:/shieldlist.org/, you find a comprehensive list of Arduino shields.

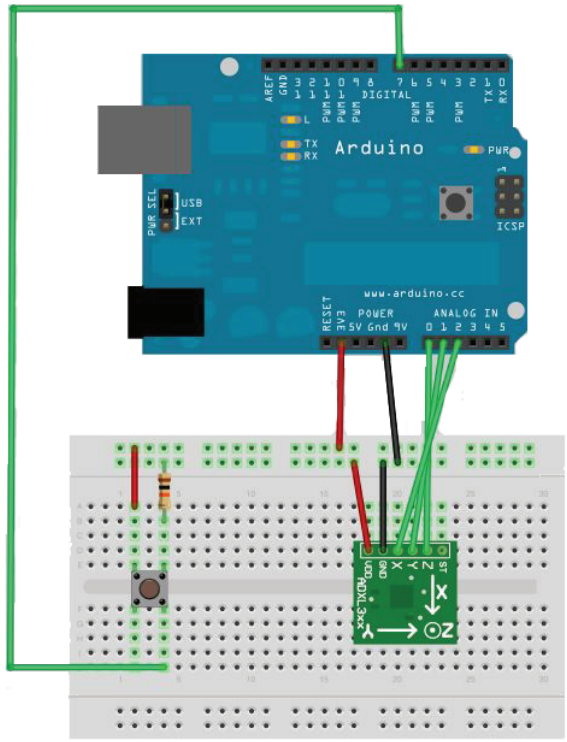


Figure 53—Game controller with accelerometer and pushbutton

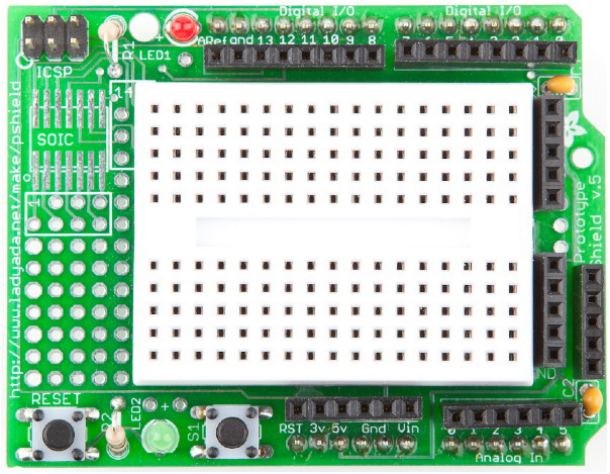


Figure 54—An Arduino prototyping shield

Now that the hardware is complete, we need a final version of the game controller software. It supports the button we have added, and it performs the anti-jittering we have created in Section 6.4, *Finding and Polishing Edge Values*, on page 9:

Download Arduino_1_0/MotionSensor/Controller/Controller.ino

```
#include <Bounce.h>
const unsigned int BUTTON_PIN = 7;
const unsigned int X_AXIS_PIN = 2;
const unsigned int Y_AXIS_PIN = 1;
const unsigned int Z_AXIS_PIN = 0;
const unsigned int NUM_AXES = 3;
const unsigned int PINS[NUM_AXES] = {
  X_AXIS_PIN, Y_AXIS_PIN, Z_AXIS_PIN
};
const unsigned int BUFFER_SIZE = 16;
const unsigned int BAUD_RATE = 19200;
int buffer[NUM_AXES][BUFFER_SIZE];
int buffer_pos[NUM_AXES] = { 0 };

Bounce button(BUTTON_PIN, 20);

void setup() {
  Serial.begin(BAUD_RATE);
  pinMode(BUTTON_PIN, INPUT);
}

int get_axis(const int axis) {
  delay(1);
  buffer[axis][buffer_pos[axis]] = analogRead(PINS[axis]);
  buffer_pos[axis] = (buffer_pos[axis] + 1) % BUFFER_SIZE;

  long sum = 0;
  for (unsigned int i = 0; i < BUFFER_SIZE; i++)
    sum += buffer[axis][i];
  return round(sum / BUFFER_SIZE);
}
int get_x() { return get_axis(0); }
int get_y() { return get_axis(1); }
int get_z() { return get_axis(2); }
void loop() {
  Serial.print(get_x());
  Serial.print(" ");
  Serial.print(get_y());
  Serial.print(" ");
  Serial.print(get_z());
  Serial.print(" ");
  if (button.update())
    Serial.println(button.read() == HIGH ? "1" : "0");
  else
```

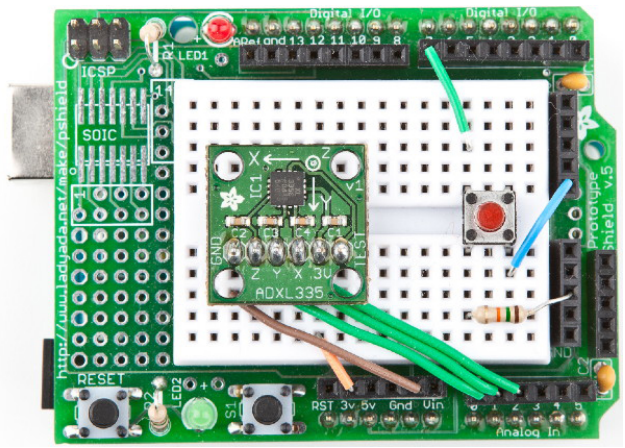


Figure 55—The complete game controller on a Proto shield

```
    Serial.println("0");
}
```

As in Section 3.7, *Building a Dice Game*, on page ?, we use the `Bounce` class to debounce the button. The rest of the code is pretty much standard, and the only thing worth mentioning is that we use a 19200 baud rate to transfer the controller data sufficiently fast.

Compile and upload the code, open the serial terminal, and play around with the controller. Move it, press the button sometimes, and it should output something like the following:

```
324 365 396 0
325 364 397 0
325 364 397 1
325 364 397 0
325 365 397 0
325 365 397 1
326 364 397 0
```

A homemade game controller is nice, but wouldn't it be even nicer if we also had a game that supports it? That's what we will build in the next section.