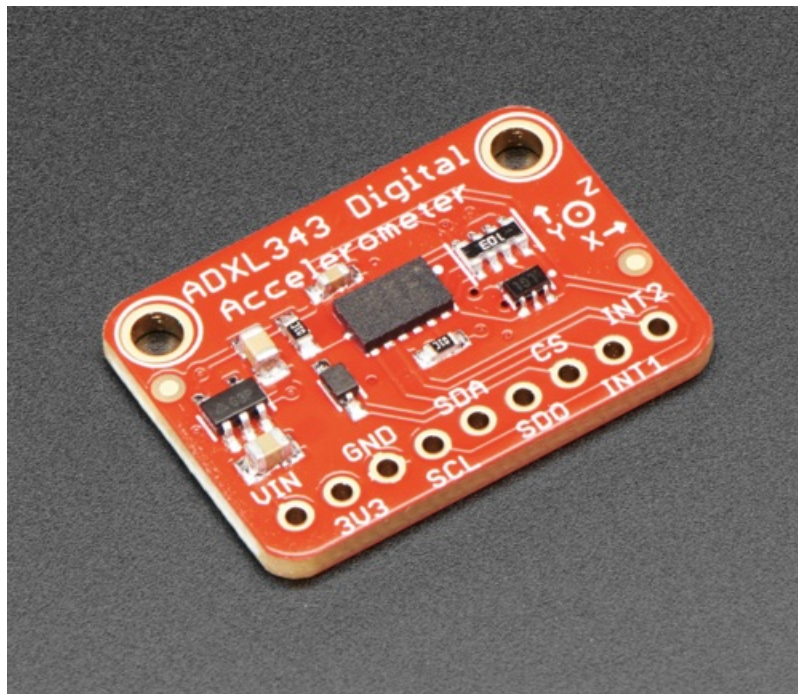


Analog Devices ADXL343 Breakout Learning Guide

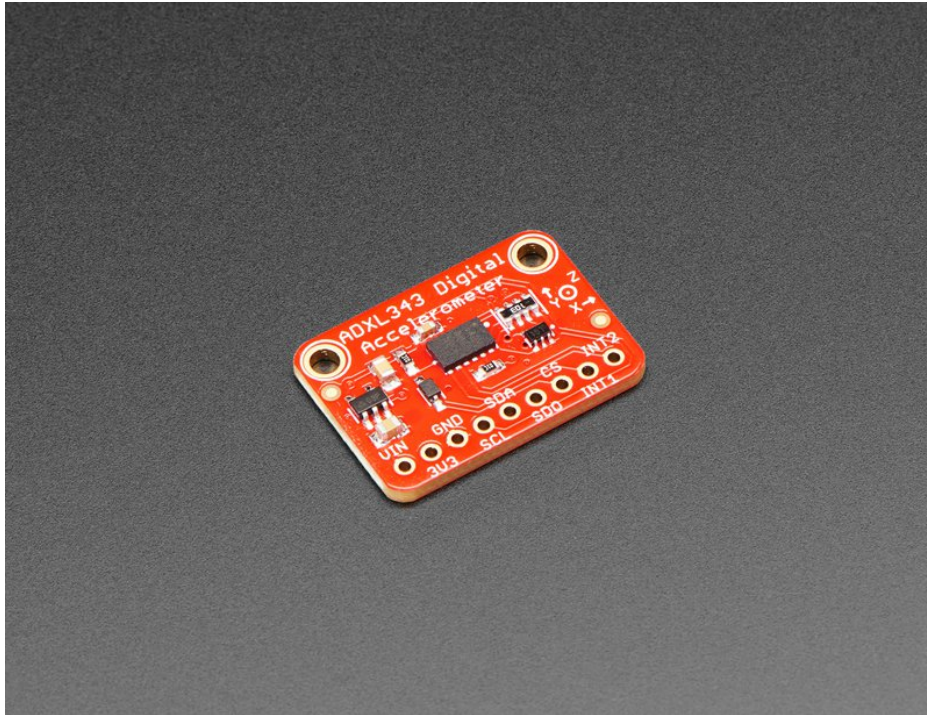
Created by Kevin Townsend



Last updated on 2019-03-02 05:26:45 AM UTC

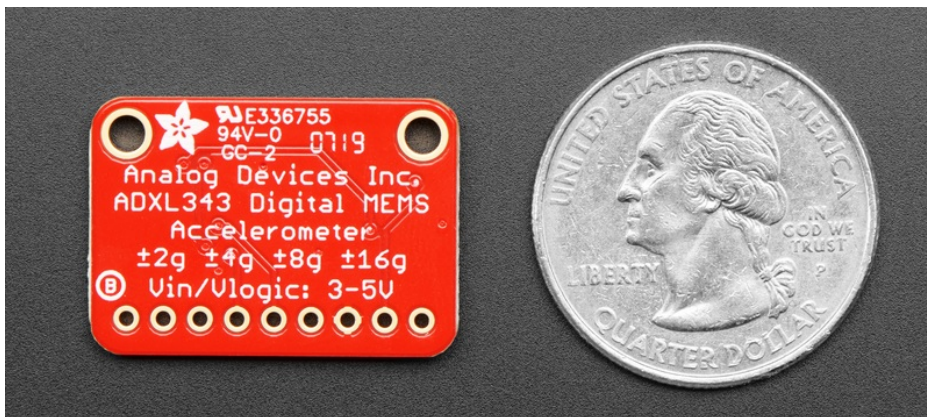
Overview

Analog Devices has followed up on their popular classic, the ADXL345, with this near-drop-in-replacement, the ADXL343. Like the original, this is a triple-axis accelerometer with digital I2C and SPI interface breakout. It has a wide sensitivity range and high resolution, operating with an 10 or 13-bit internal ADC. Built-in motion detection features make tap, double-tap, activity, inactivity, and free-fall detection trivial. There's two interrupt pins, and you can map any of the interrupts independently to either of them



The ADXL343 is nearly identical in specifications to the ADXL345, and code written for the '345 will likely work on the '343 as-is. This new accelerometer has some nice price improvements to stay within your budget.

The sensor has three axes of measurements, X Y Z, and pins that can be used either as I2C or SPI digital interfacing. You can set the sensitivity level to either $\pm 2g$, $\pm 4g$, $\pm 8g$ or $\pm 16g$. The lower range gives more resolution for slow movements, the higher range is good for high speed tracking. The ADXL343 is the latest and greatest from Analog Devices, known for their exceptional quality MEMS devices.

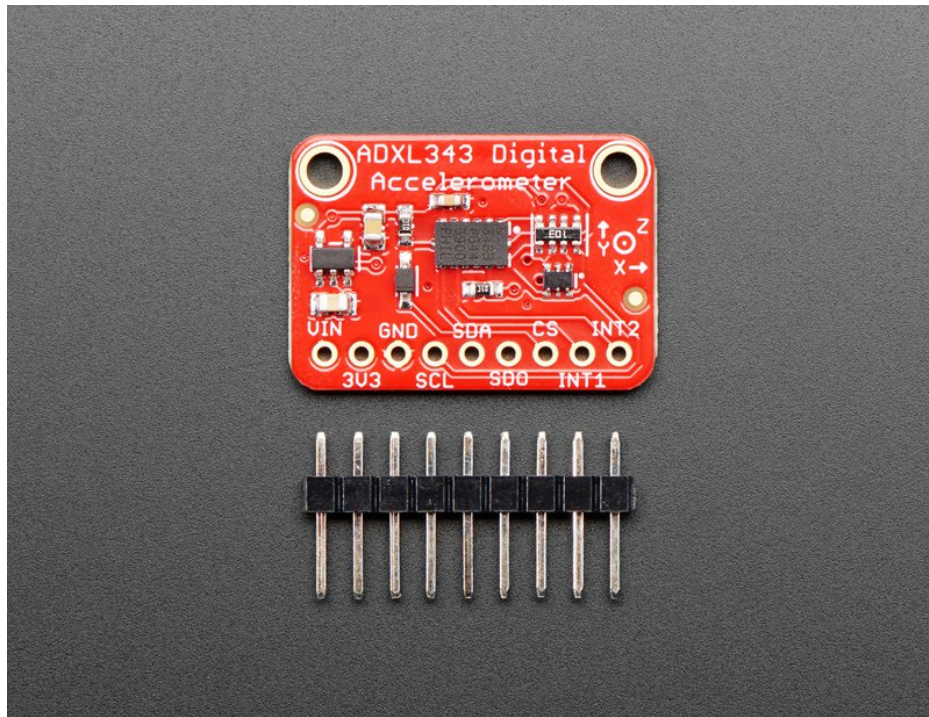


We added an on-board 3.3V regulator and logic-level shifting circuitry, making it a perfect choice for interfacing with

any 3V or 5V microcontroller or computer such as Arduino or Raspberry Pi. We even have library and example code for both Arduino/C++ and CircuitPython, so you can get started super fast with any platform!

Each order comes with a fully tested and assembled breakout and some header for soldering to a PCB or breadboard. Comes with 9 pin 0.1" standard header in case you want to use it with a breadboard or perfboard. Two 2.5mm (0.1") mounting holes for easy attachment. You'll be up and running in under 5 minutes!

Thanks to Digi-Key (<https://adafru.it/BJr>) and Analog Devices (<https://adafru.it/DPF>) for sponsoring the development of this breakout board - we've made the PCB "Digi-Key red (<https://adafru.it/BJr>)" in their honor!



Technical Characteristics

- 3-axis MEMs digital accelerometer
- Input voltage: 2.0-3.6 V
- User-selectable output resolution:
 - +/- 2 g (10-bit data, or +/- 512)
 - +/- 4 g (11-bit data, or +/- 1024)
 - +/- 8g (12-bit data, or +/- 2048)
 - +/- 16 g (13-bit data, or +/- 4096)
- User-selectable data rate (0.1 .. 3200 Hz)
- Hardware support for free-fall detection, tap detection, and activity/inactivity
- HW support for both I2C and SPI, although I2C is recommended for ease of use.

Pinout

The ADXL343 breakout has the following pinout:



Power Pins

This breakout board can be run on **3.3V** and **5V** systems, although only the **SCL** and **SDA** lines are 5V safe (other pins like INT will need to be manually level-shifted by you).

- **VIN** - This is the input to the 3.3V voltage regulator, which makes it possible to use the 3.3V sensor on 5V systems. It also determines the logic level of the SCL and SDA pins. Connect this to **3.3V** on the MCU for 3.3V boards (Adafruit Feathers), or **5.0V** for 5V Arduinos (Arduino Uno, etc.).
- **3V3** - This is the **OUTPUT** of the 3.3V regulator, and can be used to provide 3.3V power to other parts of your project if required (<100mA).
- **GND** - Connect this to the **GND** pin on your development board to make sure they are sharing a common GND connection, or the electrons won't have anywhere to flow!



NOTE: Only SCL and SDA are 5V safe on this board. Using any other pins on a 5V system will require manual level shifting of the pins used (INT, etc.)

Digital Pins

- **SCL** - The clock line on the I2C bus. This pin has an internal pullup resistor on the PCB, which is required as part of the I2C spec, meaning you don't need to add one externally yourself. This also functions as **SCK** in SPI mode.
- **SDA** - The data line on the I2C bus. This pin has an internal pullup resistor on the PCB, which is required as part of the I2C spec, meaning you don't need to add one externally yourself. This also functions as **MISO** in SPI mode.
- **SDO/ALT ADDR** - This pin can be used as **MOSI** in SPI mode, but is more commonly used as an optional bit in the I2C bus address. By default this pin is pulled down, meaning it has a value of **0** at startup, which will result in an I2C address of **0x53**. If you set this pin high (to 3.3V), and reset, the I2C address will be updated to **0x1D**.
- **CS**: This dual purpose pin can be used as the chip select line in SPI mode, but also determines whether the board will boot up into I2C or SPI mode. The default of logic high sets the board up for I2C, and manually setting this pin low and resetting will cause the device to enter SPI mode. Please note that SPI mode is not actively supported and the SPI pins are not all 5V safe and level shifted, so care will be required when using it!
- **INT1** and **INT2**: There are two optional interrupt output pins on this sensor, which can be configured to change their state when one or more 'events' occur. For details on how to use these interrupts, see the Arduino/HW Interrupts page later in this guide.

Special HW Features

While the ADXL343 shares some things in common with most other general-purpose 3-axis accelerometers, it has the following additional features to make it easier to use in certain situations. Because these are implemented in HW inside the sensor, there is a lot less heavy lifting to do on the MCU side.

Freefall Detection

You can use this accelerometer to detect a freefall condition (device falling off a desk, etc.) with user-defined thresholds, and one or the two INT pins can be setup to fire when a freefall condition is detected, allowing you to shut any motors or moving parts off, or indicate in a logging system that the data may not be valid (such as a plant sensor in a pot that likely got knocked over by the wind).

Activity/Inactivity Detection

Rather than constantly polling an accelerometer to see if movement is detected, you can configure the ADXL343 to let you know when there is (one or both of) activity or inactivity on the device, with user-adjustable thresholds. This can be configured to fire an INT pin, which you could use to wakeup your device, for example, or put it to sleep after a certain amount of inactivity.

Tap/Double-Tap Detection

Rather than having to do complex tap and double-tap detection of the device by analyzing the magnitude of acceleration changes over time, you can detect a 'tap' or 'double-tap' of your device in HW, and fire one of the INT pins when the event is detected, significantly reducing the code and data parsing on the MCU side.

See the **HW Interrupts** page later in this guide for details on how to use these features in practice!

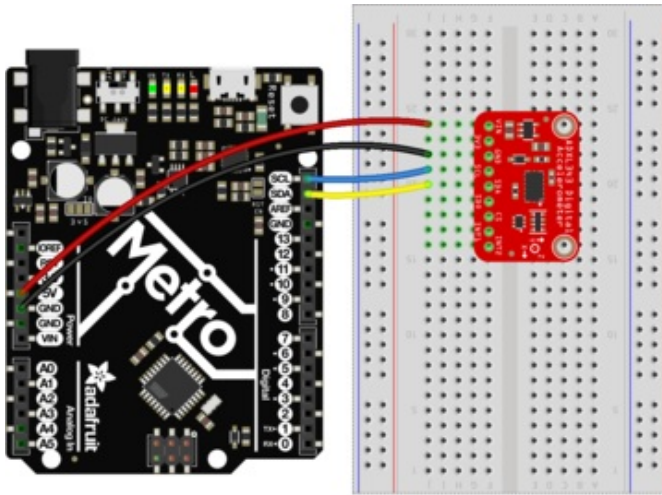
Arduino

Wiring

Hooking up the ADXL343 to your Arduino is easy:



For Arduino Metro and other 5V microcontrollers, use 5V for Vin. For Feather and other 3V microcontrollers, use 3.3V



- Connect **SCL** on the Metro to **SCL** on the ADXL343
- Connect **SDA** on the Metro to **SDA** in the ADXL343
- Connect **GND** on the Metro to **GND** on the ADXL343
- For **3.3V LOGIC** boards: connect **3.3V** on the Arduino/Metro to **VIN** on the ADXL343
- For **5.0V LOGIC** boards: Connect **5V** on the Arduino/Metro to **VIN** on the ADXL343

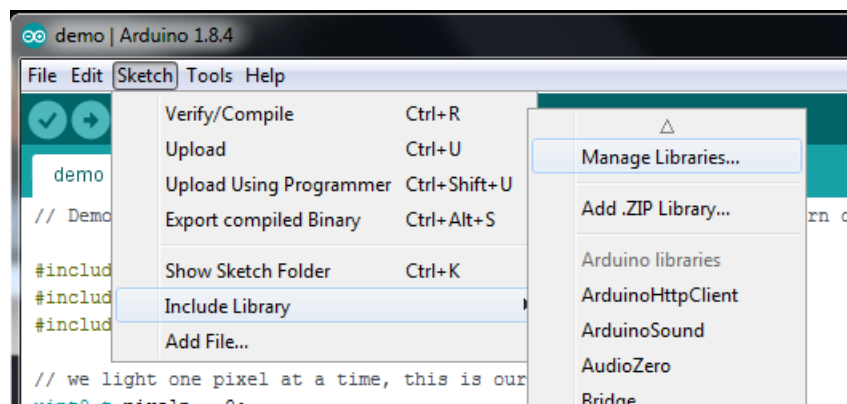
The final results should resemble the illustration above, showing an Adafruit Metro development board.



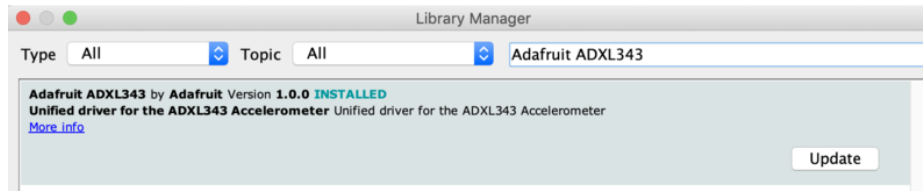
Only the SCL and SDA pins on the ADXL343 are level shifted and safe to use on 5V systems like the Arduino Uno. If you are using other pins on the breakout (INT, etc.) on a 5V system, you will need to level shift these yourself. We have some tutorials on how to do this in the learning system, simply search for 'level shifting'!

Installation

The Adafruit_ADXL345 library can be installed using the Arduino **Library Manager**, accessible through the **Manage Libraries ...** menu item.



Click the **Manage Libraries ...** menu item, search for **Adafruit ADXL343**, and select the **Adafruit ADXL343** library:



Load Example

To make sure that everything is wired up correctly, you can run the **sensortest** example available in the **Adafruit_ADXL343** examples folder, loadable via the **File -> Examples -> Adafruit ADXL343 -> sensortest** menu item.

Upload the sketch to your board and open up the Serial Monitor (**Tools->Serial Monitor**). You should see some acceleration data for for X/Y/Z.

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL343.h>

/* Assign a unique ID to this sensor at the same time */
/* Uncomment following line for default Wire bus */
Adafruit_ADXL343 accel = Adafruit_ADXL343(12345);

/* NeoTrellis M4, etc. */
/* Uncomment following line for Wire1 bus */
//Adafruit_ADXL343 accel = Adafruit_ADXL343(12345, &Wire1);

void displaySensorDetails(void)
{
  sensor_t sensor;
  accel.getSensor(&sensor);
  Serial.println("-----");
  Serial.print ("Sensor: "); Serial.println(sensor.name);
  Serial.print ("Driver Ver: "); Serial.println(sensor.version);
  Serial.print ("Unique ID: "); Serial.println(sensor.sensor_id);
  Serial.print ("Max Value: "); Serial.print(sensor.max_value); Serial.println(" m/s^2");
  Serial.print ("Min Value: "); Serial.print(sensor.min_value); Serial.println(" m/s^2");
  Serial.print ("Resolution: "); Serial.print(sensor.resolution); Serial.println(" m/s^2");
  Serial.println("-----");
  Serial.println("");
  delay(500);
}

void displayDataRate(void)
{
  Serial.print ("Data Rate: ");

  switch(accel.getDataRate())
  {
    case ADXL343_DATARATE_3200_HZ:
      Serial.print ("3200 ");
      break;
    case ADXL343_DATARATE_1600_HZ:
      Serial.print ("1600 ");

```



```

    break;
case ADXL343_DATARATE_800_HZ:
    Serial.print ("800 ");
    break;
case ADXL343_DATARATE_400_HZ:
    Serial.print ("400 ");
    break;
case ADXL343_DATARATE_200_HZ:
    Serial.print ("200 ");
    break;
case ADXL343_DATARATE_100_HZ:
    Serial.print ("100 ");
    break;
case ADXL343_DATARATE_50_HZ:
    Serial.print ("50 ");
    break;
case ADXL343_DATARATE_25_HZ:
    Serial.print ("25 ");
    break;
case ADXL343_DATARATE_12_5_HZ:
    Serial.print ("12.5 ");
    break;
case ADXL343_DATARATE_6_25HZ:
    Serial.print ("6.25 ");
    break;
case ADXL343_DATARATE_3_13_HZ:
    Serial.print ("3.13 ");
    break;
case ADXL343_DATARATE_1_56_HZ:
    Serial.print ("1.56 ");
    break;
case ADXL343_DATARATE_0_78_HZ:
    Serial.print ("0.78 ");
    break;
case ADXL343_DATARATE_0_39_HZ:
    Serial.print ("0.39 ");
    break;
case ADXL343_DATARATE_0_20_HZ:
    Serial.print ("0.20 ");
    break;
case ADXL343_DATARATE_0_10_HZ:
    Serial.print ("0.10 ");
    break;
default:
    Serial.print ("???? ");
    break;
}
Serial.println(" Hz");
}

void displayRange(void)
{
    Serial.print ("Range:      +/- ");

    switch(accel.getRange())
    {
        case ADXL343_RANGE_16_G:
            Serial.print ("16 ");
            break;
        case ADXL343_RANGE_8_G:

```

```

        Serial.print ("8 ");
        break;
    case ADXL343_RANGE_4_G:
        Serial.print ("4 ");
        break;
    case ADXL343_RANGE_2_G:
        Serial.print ("2 ");
        break;
    default:
        Serial.print ("?? ");
        break;
    }
    Serial.println(" g");
}

void setup(void)
{
    Serial.begin(9600);
    while (!Serial);
    Serial.println("Accelerometer Test"); Serial.println("");

    /* Initialise the sensor */
    if(!accel.begin())
    {
        /* There was a problem detecting the ADXL343 ... check your connections */
        Serial.println("Ooops, no ADXL343 detected ... Check your wiring!");
        while(1);
    }

    /* Set the range to whatever is appropriate for your project */
    accel.setRange(ADXL343_RANGE_16_G);
    // accel.setRange(ADXL343_RANGE_8_G);
    // accel.setRange(ADXL343_RANGE_4_G);
    // accel.setRange(ADXL343_RANGE_2_G);

    /* Display some basic information on this sensor */
    displaySensorDetails();
    displayDataRate();
    displayRange();
    Serial.println("");
}

void loop(void)
{
    /* Get a new sensor event */
    sensors_event_t event;
    accel.getEvent(&event);

    /* Display the results (acceleration is measured in m/s^2) */
    Serial.print("X: "); Serial.print(event.acceleration.x); Serial.print(" ");
    Serial.print("Y: "); Serial.print(event.acceleration.y); Serial.print(" ");
    Serial.print("Z: "); Serial.print(event.acceleration.z); Serial.print(" ");Serial.println("m/s^2 ");
    delay(500);
}

```

You should get something resembling the following output when you open the Serial Monitor at **9600 baud**:

```
-----  
Sensor:      ADXL343  
Driver Ver:  1  
Unique ID:   12345  
Max Value:   -156.91 m/s^2  
Min Value:   156.91 m/s^2  
Resolution:  0.04 m/s^2  
-----  
  
Data Rate:   100 Hz  
Range:       +/- 16 g  
  
X: 0.35 Y: -0.04 Z: 9.26 m/s^2  
X: 0.27 Y: 0.00 Z: 9.30 m/s^2  
X: 0.31 Y: 0.00 Z: 9.26 m/s^2  
X: 0.27 Y: -0.04 Z: 9.30 m/s^2  
X: 0.31 Y: 0.00 Z: 9.26 m/s^2  
X: 0.27 Y: 0.00 Z: 9.34 m/s^2
```

Setting the Range

You can adjust the response range of the accelerometer by setting an appropriate value in your `setup` loop, using one of the lines in the code shown below:

```
/* Set the range to whatever is appropriate for your project */  
accel.setRange(ADXL343_RANGE_16_G);  
// accel.setRange(ADXL343_RANGE_8_G);  
// accel.setRange(ADXL343_RANGE_4_G);  
// accel.setRange(ADXL343_RANGE_2_G);
```

By default, the sensor will be set to the maximum range of +/- **16g**.

HW Interrupts

Interrupt Events

The ADXL343 includes two configurable HW interrupt pins, where one or more of the following events can be 'mapped' to one of the interrupt pins:

- **Overflow:** The overflow bit is set when new data replaces unread data. This can be useful in situations where it's important to know if any data samples were 'skipped', such as in sensor fusion algorithms that depend on a specific sample rate for the best possible results.
- **Watermark:** This can be used with register 0x38 (**FIFO_CTL**) to trigger an interrupt when a user-specified number of samples are available in the internal FIFO buffer.
- **Freefall:** The **FREE_FALL** bit is set when acceleration of less than the value stored in the **THRESH_FF** register (Address 0x28) is experienced for more time than is specified in the **TIME_FF** register (Address 0x29) on all axes (logical AND). The **FREE_FALL** interrupt differs from the inactivity interrupt as follows: all axes always participate and are logically AND'ed, the timer period is much smaller (1.28 sec maximum), and the mode of operation is always dc-coupled.
- **Inactivity:** The inactivity bit is set when acceleration of less than the value stored in the **THRESH_INACT** register (Address 0x25) is experienced for more time than is specified in the **TIME_INACT** register (Address 0x26) on all participating axes, as set by the **ACT_INACT_CTL** register (Address 0x27). The maximum value for **TIME_INACT** is 255 sec.
- **Activity:** The activity bit is set when acceleration greater than the value stored in the **THRESH_ACT** register (Address 0x24) is experienced on any participating axis, set by the **ACT_INACT_CTL** register (Address 0x27).
- **Double Tap:** The **DOUBLE_TAP** bit is set when two acceleration events that are greater than the value in the **THRESH_TAP** register (Address 0x1D) occur for less time than is specified in the **DUR** register (Address 0x21), with the second tap starting after the time specified by the latent register (Address 0x22) but within the time specified in the window register (Address 0x23). See the Tap Detection section for more details.
- **Single Tap:** The **SINGLE_TAP** bit is set when a single acceleration event that is greater than the value in the **THRESH_TAP** register (Address 0x1D) occurs for less time than is specified in the **DUR** register (Address 0x21).
- **Data Ready:** The **DATA_READY** bit is set when new data is available and is cleared when no new data is available.

Mapping Interrupts to INT1/INT2

The first step to enable interrupts in your sketch is to 'map' one or more interrupt functions to either the **INT1** or **INT2** pins. This can be accomplished with the following function in **Adafruit_ADXL343**:

```
bool mapInterrupts(int_config cfg);
```

`cfg` is an 8-bit bit-field where setting the individual interrupt bit to **1** will cause the interrupt to be mapped to **INT2**, and setting the interrupt bit to **0** will map it to **INT1**. The following code shows how this works in practice (based on the `g_int_config_map` variable in the **interrupts.ino** example sketch):

```

/* Map specific interrupts to one of the two INT pins. */
g_int_config_map.bits.overrun      = ADXL343_INT1;
g_int_config_map.bits.watermark    = ADXL343_INT1;
g_int_config_map.bits.freefall     = ADXL343_INT1;
g_int_config_map.bits.inactivity   = ADXL343_INT1;
g_int_config_map.bits.activity     = ADXL343_INT1;
g_int_config_map.bits.double_tap   = ADXL343_INT1;
g_int_config_map.bits.single_tap   = ADXL343_INT1;
g_int_config_map.bits.data_ready   = ADXL343_INT2;
accel.mapInterrupts(g_int_config_map);

```

Enabling Interrupts

After mapping specific interrupt events to either INT1 or INT2, you need to 'enable' the interrupt via a second function:

```
bool enableInterrupts(int_config cfg);
```

An example of enabling the **OVERRUN** and **DATA READY** interrupts is shown below:

```

/* Enable interrupts on the accelerometer. */
g_int_config_enabled.bits.overrun    = true;      /* Set the INT1 */
g_int_config_enabled.bits.watermark  = false;
g_int_config_enabled.bits.freefall   = false;
g_int_config_enabled.bits.inactivity = false;
g_int_config_enabled.bits.activity   = false;
g_int_config_enabled.bits.double_tap = false;
g_int_config_enabled.bits.single_tap = false;
g_int_config_enabled.bits.data_ready = true;     /* Set to INT2 */
accel.enableInterrupts(g_int_config_enabled);

```

Connecting ADXL343 INT pins to the MCU

In order to 'detect' the interrupt generated by the ADXL, you also need to connect the INT1 and/or INT2 pins on the ADXL to an appropriate interrupt-enabled input pin on your MCU.

The interrupt input on the MCU needs to have the following pin characteristics:

- Must have support for interrupt mode (if you want to automatically fire an interrupt service routine when the ADXL's INT pins are triggered)
- Must be configured as an input
- Must be 'attached' to an interrupt service routine, which is the function that will be called when a **RISING** edge is detected on the MCU's interrupt input.

Some sample code of setting these pins up properly is shown below (assumed an **Adafruit Feather M0 Basic**, see the documentation for pin selection on other boards):

```

/** The input pins to enable the interrupt on, connected to INT1 and INT2 on the ADXL. */
#define INPUT_PIN_INT1  (5) // Uno = (2)
#define INPUT_PIN_INT2  (6) // Uno = (3)

...

/** Interrupt service routine for INT1 events. */
void int1_isr(void)
{
  /* TODO: Do something! */
}

/** Interrupt service routine for INT2 events. */
void int2_isr(void)
{
  /* TODO: Do something! */
}

...

/* Attach interrupt inputs on the MCU. */
pinMode(LED_BUILTIN, OUTPUT);
pinMode(INPUT_PIN_INT1, INPUT);
pinMode(INPUT_PIN_INT2, INPUT);
attachInterrupt(digitalPinToInterrupt(INPUT_PIN_INT1), int1_isr, RISING);
attachInterrupt(digitalPinToInterrupt(INPUT_PIN_INT2), int2_isr, RISING);

```

Complete Example

To see how all of these pieces fit together, you can see the code for the **interrupts** example that is part of the standard Adafruit driver, shown below for convenience sake.

This examples enables two interrupt events on two different pins, and tracks the number of times those interrupt handlers are fired. The main loop of the program continually checks if a new interrupt event was detected, and display some details on the interrupt source when an event it detected.

```

#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL343.h>

/* Assign a unique ID to this sensor at the same time */
Adafruit_ADXL343 accel = Adafruit_ADXL343(12345);

/** The input pins to enable the interrupt on, connected to INT1 and INT2 on the ADXL. */
#define INPUT_PIN_INT1  (5) // Uno = (2)
#define INPUT_PIN_INT2  (6) // Uno = (3)

/**
 * This struct is used to count the number of times that specific interrups
 * have been fired by the ADXL and detected on the MCU. They will increment
 * by one for each event associated with the specified interrupt 'bit'.
 */
struct adxl_int_stats {
  uint32_t data_ready;
  uint32_t single_tap;
  uint32_t double_tap;
  uint32_t activity;
};

```

```

    uint32_t inactivity;
    uint32_t freefall;
    uint32_t watermark;
    uint32_t overrun;
    uint32_t total;
};

/** Global stats block, incremented inside the interrupt handler(s). */
struct adxl_int_stats g_int_stats = { 0 };

/** Global counter to track the numbers of unused interrupts fired. */
uint32_t g_ints_fired = 0;

/** Global variable to determine which interrupt(s) are enabled on the ADXL343. */
int_config g_int_config_enabled = { 0 };

/** Global variables to determine which INT pin interrupt(s) are mapped to on the ADXL343. */
int_config g_int_config_map = { 0 };

/** Interrupt service routine for INT1 events. */
void int1_isr(void)
{
    /* By default, this sketch routes the OVERRUN interrupt to INT1. */
    g_int_stats.overrun++;
    g_int_stats.total++;
    g_ints_fired++;

    /* TODO: Toggle an LED! */
}

/** Interrupt service routine for INT2 events. */
void int2_isr(void)
{
    /* By default, this sketch routes the DATA_READY interrupt to INT2. */
    g_int_stats.data_ready++;
    g_int_stats.total++;
    g_ints_fired++;

    /* TODO: Toggle an LED! */
}

/** Configures the HW interrupts on the ADXL343 and the target MCU. */
void config_interrupts(void)
{
    /* NOTE: Once an interrupt fires on the ADXL you can read a register
    * to know the source of the interrupt, but since this would likely
    * happen in the 'interrupt context' performing an I2C read is a bad
    * idea since it will block the device from handling other interrupts
    * in a timely manner.
    *
    * The best approach is to try to make use of only two interrupts on
    * two different interrupt pins, so that when an interrupt fires, based
    * on the 'ISR' function that is called, you already know the int source.
    */

    /* Attach interrupt inputs on the MCU. */
    pinMode(LED_BUILTIN, OUTPUT);
    pinMode(INPUT_PIN_INT1, INPUT);
    pinMode(INPUT_PIN_INT2, INPUT);
}

```

```

attachInterrupt(digitalPinToInterrupt(INPUT_PIN_INT1), int1_isr, RISING);
attachInterrupt(digitalPinToInterrupt(INPUT_PIN_INT2), int2_isr, RISING);

/* Enable interrupts on the accelerometer. */
g_int_config_enabled.bits.overrun    = true;    /* Set the INT1 */
g_int_config_enabled.bits.watermark = false;
g_int_config_enabled.bits.freefall   = false;
g_int_config_enabled.bits.inactivity = false;
g_int_config_enabled.bits.activity   = false;
g_int_config_enabled.bits.double_tap = false;
g_int_config_enabled.bits.single_tap = false;
g_int_config_enabled.bits.data_ready = true;    /* Set to INT2 */
accel.enableInterrupts(g_int_config_enabled);

/* Map specific interrupts to one of the two INT pins. */
g_int_config_map.bits.overrun    = ADXL343_INT1;
g_int_config_map.bits.watermark = ADXL343_INT1;
g_int_config_map.bits.freefall   = ADXL343_INT1;
g_int_config_map.bits.inactivity = ADXL343_INT1;
g_int_config_map.bits.activity   = ADXL343_INT1;
g_int_config_map.bits.double_tap = ADXL343_INT1;
g_int_config_map.bits.single_tap = ADXL343_INT1;
g_int_config_map.bits.data_ready = ADXL343_INT2;
accel.mapInterrupts(g_int_config_map);
}

void setup(void)
{
  Serial.begin(9600);
  while (!Serial);
  Serial.println("ADXL343 Interrupt Tester"); Serial.println("");

  /* Initialise the sensor */
  if(!accel.begin())
  {
    /* There was a problem detecting the ADXL343 ... check your connections */
    Serial.println("Ooops, no ADXL343 detected ... Check your wiring!");
    while(1);
  }

  /* Set the range to whatever is appropriate for your project */
  accel.setRange(ADXL343_RANGE_16_G);
  // displaySetRange(ADXL343_RANGE_8_G);
  // displaySetRange(ADXL343_RANGE_4_G);
  // displaySetRange(ADXL343_RANGE_2_G);

  /* Configure the HW interrupts. */
  config_interrupts();

  Serial.println("ADXL343 init complete. Waiting for INT activity.");
}

void loop(void)
{
  /* Get a new sensor event */
  sensors_event_t event;
  accel.getEvent(&event);
  delay(10);
}

```



```

while (g_ints_fired) {
  Serial.println("INT detected!");
  Serial.print("\tOVERRUN Count:  "); Serial.println(g_int_stats.overrun, DEC);
  Serial.print("\tDATA_READY Count: "); Serial.println(g_int_stats.data_ready, DEC);

  /* Decrement the unhandled int counter. */
  g_ints_fired--;
}
}

```

Single Tap Example

A slightly simplified example that detects single taps is shown below as well:

```

#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL343.h>

/* Assign a unique ID to this sensor at the same time */
Adafruit_ADXL343 accel = Adafruit_ADXL343(12345);

/** The input pin to enable the interrupt on, connected to INT1 on the ADXL. */
#define INPUT_PIN_INT1  (5) // SAMD21/SAMD51 = 5 for interrupt pin

uint32_t g_tap_count = 0;
int_config g_int_config_enabled = { 0 };
int_config g_int_config_map = { 0 };

/** Interrupt service routine for INT1 events. This will be called when a single tap is detected. */
void int1_isr(void)
{
  g_tap_count++;
}

void setup(void)
{
  Serial.begin(9600);
  while (!Serial);
  Serial.println("ADXL343 Single Tap INT Tester"); Serial.println("");

  /* Initialise the sensor */
  if(!accel.begin())
  {
    /* There was a problem detecting the ADXL343 ... check your connections */
    Serial.println("Ooops, no ADXL343 detected ... Check your wiring!");
    while(1);
  }

  /* Set the range to whatever is appropriate for your project */
  accel.setRange(ADXL343_RANGE_16_G);

  /* Configure the HW interrupts. */
  pinMode(INPUT_PIN_INT1, INPUT);
  attachInterrupt(digitalPinToInterrupt(INPUT_PIN_INT1), int1_isr, RISING);

  /* Enable single tap interrupts on the accelerometer. */
  g_int_config_enabled.bits.single_tap = true;
  accel.enableInterrupts(g_int_config_enabled);
}

```

```

accel.enableInterrupts(g_int_config_enabled);

/* Map single tap interrupts to INT1 pin. */
g_int_config_map.bits.single_tap = ADXL343_INT1;
accel.mapInterrupts(g_int_config_map);

/* Reset tap counter. */
g_tap_count = 0;

Serial.println("ADXL343 init complete. Waiting for single tap INT activity.");
}

void loop(void)
{
  /* Get a new sensor event */
  /* Reading data clears the interrupts. */
  sensors_event_t event;
  accel.getEvent(&event);
  delay(10);

  while (g_tap_count) {
    Serial.println("Single tap detected!");
    /* Clear the interrupt by check the source register.. */
    uint8_t source = accel.checkInterrupts();
    /* Decrement the local interrupt counter. */
    g_tap_count--;
  }
}

```

Running this **singletap** demo and tapping the device gently should give the following output:

```

/dev
ADXL343 Single Tap INT Tester
ADXL343 init complete. Waiting for single tap INT activity.
Single tap detected!
Single tap detected!
Single tap detected!
Single tap detected!
Single tap detected!

```

Arduino API

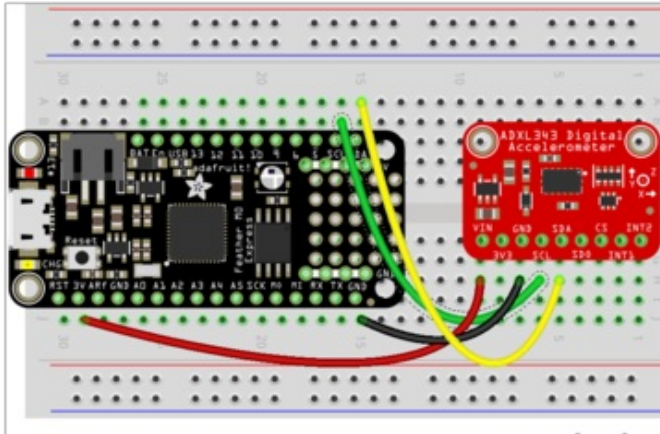
[Arduino API \(https://adafru.it/E7b\)](https://adafru.it/E7b)

Python and CircuitPython

Using the ADXL343 with CircuitPython is easy. The [Adafruit_CircuitPython_ADXL34x \(https://adafru.it/E5S\)](https://adafru.it/E5S) repo on Github always contains the latest public code, and allows you to quickly and easily get started with the Adafruit ADXL343 breakout board.

CircuitPython Wiring

The diagram below shows how you can wire up your CircuitPython board (in Feather form below) to the ADXL343:



- Connect **SCL** on the Feather to **SCL** on the ADXL343
- Connect **SDA** on the Feather to **SDA** in the ADXL343
- Connect **GND** on the Feather to **GND** on the ADXL343
- Connect **3.3V** on the Feather to **VIN** on the ADXL343

Library Installation

You'll need to install the [Adafruit CircuitPython ADXL34x \(https://adafru.it/E5S\)](https://adafru.it/E5S) library on your CircuitPython board.

First make sure you are running the [latest version of Adafruit CircuitPython \(https://adafru.it/Amd\)](https://adafru.it/Amd) for your board.

Next you'll need to install the necessary libraries to use the hardware--carefully follow the steps to find and install these libraries from [Adafruit's CircuitPython library bundle \(https://adafru.it/uap\)](https://adafru.it/uap). Our CircuitPython starter guide has [a great page on how to install the library bundle \(https://adafru.it/ABU\)](https://adafru.it/ABU).

For non-express boards like the Trinket M0 or Gemma M0, you'll need to manually install the necessary libraries from the bundle:

- `adafruit_adxl34x.mpy`
- `adafruit_bus_device`

Before continuing make sure your board's lib folder or root filesystem has the `adafruit_adxl34x.mpy`, and `adafruit_bus_device` files and folders copied over.

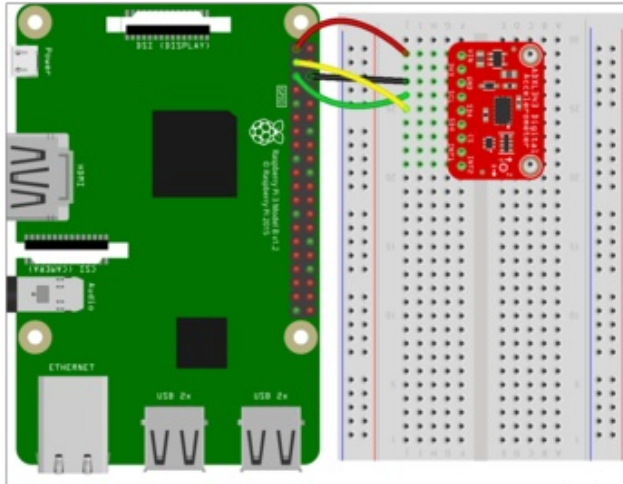
Next [connect to the board's serial REPL \(https://adafru.it/pMf\)](https://adafru.it/pMf) so you are at the CircuitPython `>>>` prompt.

If this works for you, scroll down to the **Example Code** section to get a minimal example of talking to the ADXL343 up and running in no time!

Python Wiring

Since there's *dozens* of Linux computers/boards you can use we will show wiring for Raspberry Pi. For other platforms, [please visit the guide for CircuitPython on Linux to see whether your platform is supported \(https://adafru.it/BSN\)](https://adafru.it/BSN).

Make the following connections between the Pi and the ADXL343:



- Connect **SCL** on the RPi to **SCL** on the ADXL343
- Connect **SDA** on the Rpi to **SDA** in the ADXL343
- Connect **GND** on the Rpi to **GND** on the ADXL343
- Connect **3.3V** on the Rpi to **VIN** on the ADXL343

Python Installation of the ADXL34x Library

You'll need to install the Adafruit_Blinka library that provides the CircuitPython support in Python. This may also require enabling I2C on your platform and verifying you are running Python 3. [Since each platform is a little different, and Linux changes often, please visit the CircuitPython on Linux guide to get your computer ready \(https://adafru.it/BSN\)](https://adafru.it/BSN)!

Once that's done, from your command line run the following command:

- `sudo pip3 install adafruit-circuitpython-adxl34x`

If your default Python is version 3 you may need to run 'pip' instead. Just make sure you aren't trying to use CircuitPython on Python 2.x, it isn't supported!

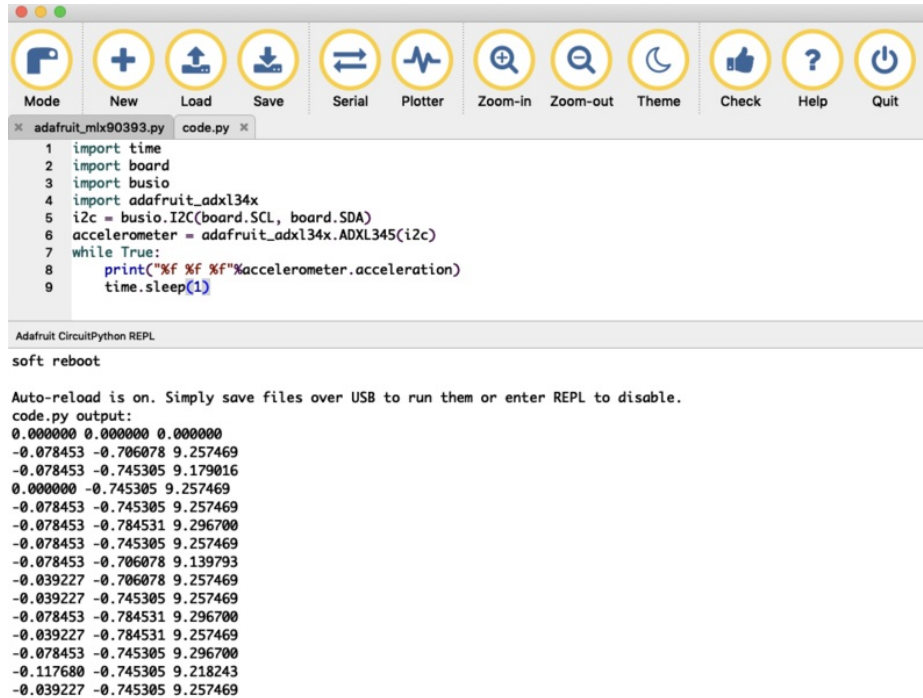
If you open the **Serial** tab in mu-editor with the sample code running, you should get output resembling the following if everything was setup correctly:

Example Code

The ADXL34x codebase contains [several examples \(https://adafru.it/E6E\)](https://adafru.it/E6E), but the easiest place to start is to open up or copy and paste the `adxl34x_simpletest.py` code, shown below for convenience sake:

```
import time
import board
import busio
import adafruit_adxl34x
i2c = busio.I2C(board.SCL, board.SDA)
accelerometer = adafruit_adxl34x.ADXL345(i2c)
while True:
    print("%f %f %f"%accelerometer.acceleration)
    time.sleep(1)
```

If you open this in mu-editor and save it as `code.py`, opening the **Serial** window, you should see something resembling the following output:



The screenshot shows the mu-editor interface. At the top, there is a toolbar with icons for Mode, New, Load, Save, Serial, Plotter, Zoom-in, Zoom-out, Theme, Check, Help, and Quit. Below the toolbar, the code editor shows the following Python code:

```
1 import time
2 import board
3 import busio
4 import adafruit_adxl34x
5 i2c = busio.I2C(board.SCL, board.SDA)
6 accelerometer = adafruit_adxl34x.ADXL345(i2c)
7 while True:
8     print("%f %f %f"%accelerometer.acceleration)
9     time.sleep(1)
```

Below the code editor, the Serial window shows the output of the program:

```
Adafruit CircuitPython REPL
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
0.000000 0.000000 0.000000
-0.078453 -0.706078 9.257469
-0.078453 -0.745305 9.179016
0.000000 -0.745305 9.257469
-0.078453 -0.745305 9.257469
-0.078453 -0.784531 9.296700
-0.078453 -0.745305 9.257469
-0.078453 -0.706078 9.139793
-0.039227 -0.706078 9.257469
-0.039227 -0.745305 9.257469
-0.078453 -0.784531 9.296700
-0.039227 -0.784531 9.257469
-0.078453 -0.745305 9.296700
-0.117680 -0.745305 9.218243
-0.039227 -0.745305 9.257469
```

Python Docs

[Python Docs \(https://adafru.it/E7c\)](https://adafru.it/E7c)

Downloads

Drivers

- CircuitPython: [Adafruit_CircuitPython_ADXL34x](https://adafru.it/E5S) (<https://adafru.it/E5S>)
- Arduino: [Adafruit_ADXL343](https://adafru.it/E5T) (<https://adafru.it/E5T>)

Design Files

The latest PCB design files (for Autodesk Eagle) can be found on Github at:

- [Adafruit_ADXL343_PCB](https://adafru.it/E5U) (<https://adafru.it/E5U>)

If you use Fritzing, the Fritzing object for this breakout is also available on Github at:

- [Adafruit ADXL343 Fritzing Object](https://adafru.it/E5X) (<https://adafru.it/E5X>)

Datasheet

You can download the datasheet for the ADXL343 via the button below:

<https://adafru.it/E5V>

<https://adafru.it/E5V>

Apps Notes

Full-Features Pedometer Design (Analog Devices)

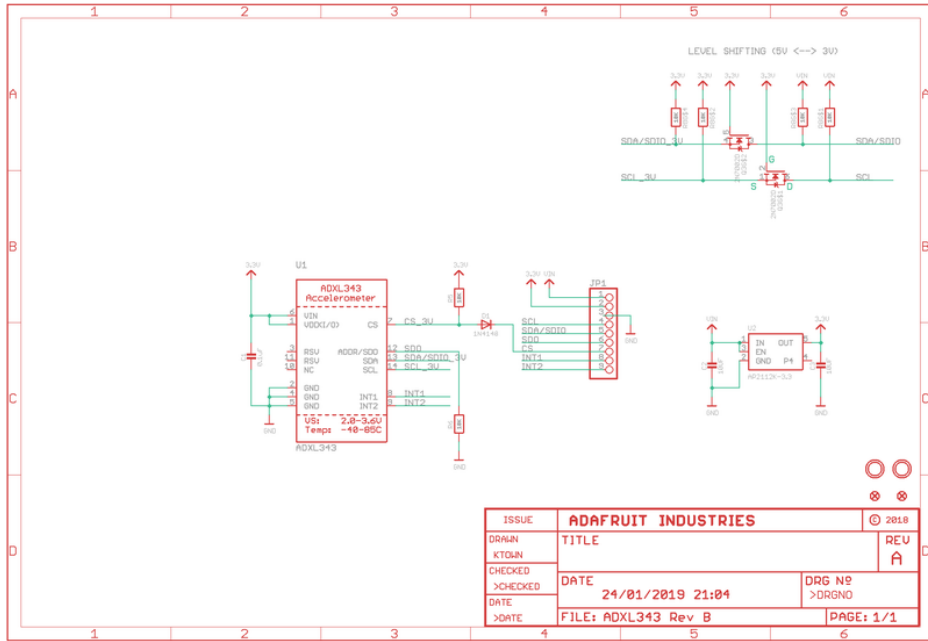
If you're interested in going one step further with the ADXL34x device family, you may find the following app-note from Analog Devices useful for ideas of technical tips:

<https://adafru.it/E5W>

<https://adafru.it/E5W>

This app note details a method of performing 'step detection' with an algorithm using threshold crossing and timing intervals, and may be a useful introduction to correlating sensor data with a specific behaviour model.

Schematic



Board Layout

All measurements below are in inches!

