Sarah Wood

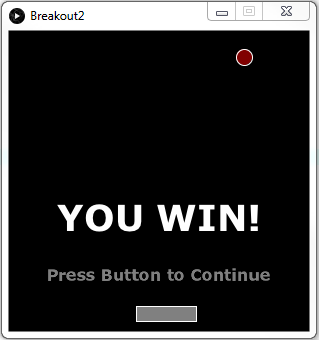
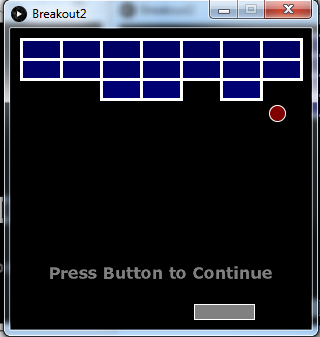
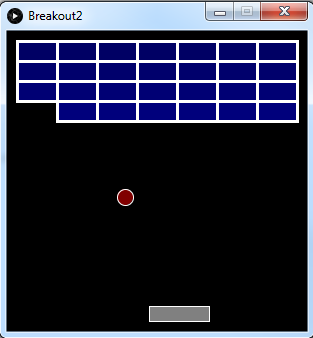
Embedded Programming

Ch6 Assg 1

I had to modify both the book code, and the Processing code for this to work. My modifications are noted in the source code. The Processing code mods were simple; I just had to provide constants instead of variables in a few places. The book code mods were a little more substantial. I imported the class and member functions for the AXDL345 sensor, and changed the “loop” function to orient to the AXDL object.

There was one nagging issue, but the game was still playable. This particular sensor is apparently prone to low-level electrical interference, particularly from fluorescent lighting. The Arduino community is working on a “smoothing” algorithm. If I had endless time, I’d join them in their quest. The game was still playable, but the paddle was jittery.

Screen Caps



Source Code

Processing

// START:game\_serial

import processing.serial.\*;

Serial arduinoPort;

// END:game\_serial

// START:game\_constants

final int COLUMNS = 7;

final int ROWS = 4;

final int BALL\_RADIUS = 8;

final int BALL\_DIAMETER = BALL\_RADIUS \* 2;

final int MAX\_VELOCITY = 8;

final int PADDLE\_WIDTH = 60;

final int PADDLE\_HEIGHT = 15;

final int BRICK\_WIDTH = 40;

final int BRICK\_HEIGHT = 20;

final int MARGIN = 10;

final int WIDTH = COLUMNS \* BRICK\_WIDTH + 2 \* MARGIN;

final int HEIGHT = 300;

final int X\_AXIS\_MIN = 000; //SJW

final int X\_AXIS\_MAX = 300; //SJW

final int LINE\_FEED = 10;

final int BAUD\_RATE = 9600;

// END:game\_constants

// START:game\_objects

int px, py;

int vx, vy;

int xpos = WIDTH / 2;

int[][] bricks = new int[COLUMNS][ROWS];

// END:game\_objects

// START:game\_states

boolean buttonPressed = false;

boolean paused = true;

boolean done = true;

// END:game\_states

// START:game\_init

void setup() {

size(300,300); //SJW

noCursor();

textFont(loadFont("Verdana-Bold-36.vlw"));

initGame();

printArray(Serial.list());

arduinoPort = new Serial(this, Serial.list()[2], BAUD\_RATE);

arduinoPort.bufferUntil(LINE\_FEED);

}

void initGame() {

initBricks();

initBall();

}

void initBricks() {

for (int x = 0; x < COLUMNS; x++)

for (int y = 0; y < ROWS; y++ )

bricks[x][y] = 1;

}

void initBall() {

px = width / 2;

py = height / 2;

vx = int(random(-MAX\_VELOCITY, MAX\_VELOCITY));

vy = -2;

}

// END:game\_init

// START:game\_main

void draw() {

background(0);

stroke(255);

strokeWeight(3);

done = drawBricks();

if (done) {

paused = true;

printWinMessage();

}

if (paused)

printPauseMessage();

else

updateGame();

drawBall();

drawPaddle();

}

// END:game\_main

// START:game\_draw\_objects

boolean drawBricks() {

boolean allEmpty = true;

for (int x = 0; x < COLUMNS; x++) {

for (int y = 0; y < ROWS; y++) {

if (bricks[x][y] > 0) {

allEmpty = false;

fill(0, 0, 100 + y \* 8);

rect(

MARGIN + x \* BRICK\_WIDTH,

MARGIN + y \* BRICK\_HEIGHT,

BRICK\_WIDTH,

BRICK\_HEIGHT

);

}

}

}

return allEmpty;

}

void drawBall() {

strokeWeight(1);

fill(128, 0, 0);

ellipse(px, py, BALL\_DIAMETER, BALL\_DIAMETER);

}

void drawPaddle() {

int x = xpos - PADDLE\_WIDTH / 2;

int y = height - (PADDLE\_HEIGHT + MARGIN);

strokeWeight(1);

fill(128);

rect(x, y, PADDLE\_WIDTH, PADDLE\_HEIGHT);

}

// END:game\_draw\_objects

// START:game\_messages

void printWinMessage() {

fill(255);

textSize(36);

textAlign(CENTER);

text("YOU WIN!", width / 2, height \* 2 / 3);

}

void printPauseMessage() {

fill(128);

textSize(16);

textAlign(CENTER);

text("Press Button to Continue", width / 2, height \* 5 / 6);

}

// END:game\_messages

// START:game\_update

void updateGame() {

if (ballDropped()) {

initBall();

paused = true;

} else {

checkBrickCollision();

checkWallCollision();

checkPaddleCollision();

px += vx;

py += vy;

}

}

// END:game\_update

// START:game\_collision

boolean ballDropped() {

return py + vy > height - BALL\_RADIUS;

}

boolean inXRange(final int row, final int v) {

return px + v > row \* BRICK\_WIDTH &&

px + v < (row + 1) \* BRICK\_WIDTH + BALL\_DIAMETER;

}

boolean inYRange(final int col, final int v) {

return py + v > col \* BRICK\_HEIGHT &&

py + v < (col + 1) \* BRICK\_HEIGHT + BALL\_DIAMETER;

}

void checkBrickCollision() {

for (int x = 0; x < COLUMNS; x++) {

for (int y = 0; y < ROWS; y++) {

if (bricks[x][y] > 0) {

if (inXRange(x, vx) && inYRange(y, vy)) {

bricks[x][y] = 0;

if (inXRange(x, 0)) // Hit top or bottom of brick.

vy = -vy;

if (inYRange(y, 0)) // Hit left or right side of brick.

vx = -vx;

}

}

}

}

}

void checkWallCollision() {

if (px + vx < BALL\_RADIUS || px + vx > width - BALL\_RADIUS)

vx = -vx;

if (py + vy < BALL\_RADIUS || py + vy > height - BALL\_RADIUS)

vy = -vy;

}

void checkPaddleCollision() {

final int cx = xpos;

if (py + vy >= height - (PADDLE\_HEIGHT + MARGIN + 6) &&

px >= cx - PADDLE\_WIDTH / 2 &&

px <= cx + PADDLE\_WIDTH / 2)

{

vy = -vy;

vx = int(

map(

px - cx,

-(PADDLE\_WIDTH / 2), PADDLE\_WIDTH / 2,

-MAX\_VELOCITY,

MAX\_VELOCITY

)

);

}

}

// END:game\_collision

// START:game\_controller

void serialEvent(Serial port) {

final String arduinoData = port.readStringUntil(LINE\_FEED);

if (arduinoData != null) {

final int[] data = int(split(trim(arduinoData), ' ')); // <label id="code.motion.controller\_data"/>

if (data.length == 4) {

buttonPressed = (data[3] == 1);

if (buttonPressed) {

paused = !paused;

if (done) {

done = false;

initGame();

}

}

if (!paused)

xpos = int(map(data[0], X\_AXIS\_MIN, X\_AXIS\_MAX, 0, WIDTH)); // <label id="code.motion.xpos"/>

}

}

}

// END:game\_controller

AXDL345.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Function: Header file for class ADXL345

// Hardware: 3-Axis Digital Accelerometer(¡À16g)

// Arduino IDE: Arduino-1.0

// Author: Frankie.Chu

// Date: Jan 11,2013

// Version: v1.0

// by www.seeedstudio.com

//

// This library is free software; you can redistribute it and/or

// modify it under the terms of the GNU Lesser General Public

// License as published by the Free Software Foundation; either

// version 2.1 of the License, or (at your option) any later version.

//

// This library is distributed in the hope that it will be useful,

// but WITHOUT ANY WARRANTY; without even the implied warranty of

// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

// Lesser General Public License for more details.

//

// You should have received a copy of the GNU Lesser General Public

// License along with this library; if not, write to the Free Software

// Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

//

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Arduino.h"

#ifndef ADXL345\_h

#define ADXL345\_h

/\* ------- Register names ------- \*/

#define ADXL345\_DEVID 0x00

#define ADXL345\_RESERVED1 0x01

#define ADXL345\_THRESH\_TAP 0x1d

#define ADXL345\_OFSX 0x1e

#define ADXL345\_OFSY 0x1f

#define ADXL345\_OFSZ 0x20

#define ADXL345\_DUR 0x21

#define ADXL345\_LATENT 0x22

#define ADXL345\_WINDOW 0x23

#define ADXL345\_THRESH\_ACT 0x24

#define ADXL345\_THRESH\_INACT 0x25

#define ADXL345\_TIME\_INACT 0x26

#define ADXL345\_ACT\_INACT\_CTL 0x27

#define ADXL345\_THRESH\_FF 0x28

#define ADXL345\_TIME\_FF 0x29

#define ADXL345\_TAP\_AXES 0x2a

#define ADXL345\_ACT\_TAP\_STATUS 0x2b

#define ADXL345\_BW\_RATE 0x2c

#define ADXL345\_POWER\_CTL 0x2d

#define ADXL345\_INT\_ENABLE 0x2e

#define ADXL345\_INT\_MAP 0x2f

#define ADXL345\_INT\_SOURCE 0x30

#define ADXL345\_DATA\_FORMAT 0x31

#define ADXL345\_DATAX0 0x32

#define ADXL345\_DATAX1 0x33

#define ADXL345\_DATAY0 0x34

#define ADXL345\_DATAY1 0x35

#define ADXL345\_DATAZ0 0x36

#define ADXL345\_DATAZ1 0x37

#define ADXL345\_FIFO\_CTL 0x38

#define ADXL345\_FIFO\_STATUS 0x39

#define ADXL345\_BW\_1600 0xF // 1111

#define ADXL345\_BW\_800 0xE // 1110

#define ADXL345\_BW\_400 0xD // 1101

#define ADXL345\_BW\_200 0xC // 1100

#define ADXL345\_BW\_100 0xB // 1011

#define ADXL345\_BW\_50 0xA // 1010

#define ADXL345\_BW\_25 0x9 // 1001

#define ADXL345\_BW\_12 0x8 // 1000

#define ADXL345\_BW\_6 0x7 // 0111

#define ADXL345\_BW\_3 0x6 // 0110

/\*

Interrupt PINs

INT1: 0

INT2: 1

\*/

#define ADXL345\_INT1\_PIN 0x00

#define ADXL345\_INT2\_PIN 0x01

/\*Interrupt bit position\*/

#define ADXL345\_INT\_DATA\_READY\_BIT 0x07

#define ADXL345\_INT\_SINGLE\_TAP\_BIT 0x06

#define ADXL345\_INT\_DOUBLE\_TAP\_BIT 0x05

#define ADXL345\_INT\_ACTIVITY\_BIT 0x04

#define ADXL345\_INT\_INACTIVITY\_BIT 0x03

#define ADXL345\_INT\_FREE\_FALL\_BIT 0x02

#define ADXL345\_INT\_WATERMARK\_BIT 0x01

#define ADXL345\_INT\_OVERRUNY\_BIT 0x00

#define ADXL345\_DATA\_READY 0x07

#define ADXL345\_SINGLE\_TAP 0x06

#define ADXL345\_DOUBLE\_TAP 0x05

#define ADXL345\_ACTIVITY 0x04

#define ADXL345\_INACTIVITY 0x03

#define ADXL345\_FREE\_FALL 0x02

#define ADXL345\_WATERMARK 0x01

#define ADXL345\_OVERRUNY 0x00

#define ADXL345\_OK 1 // no error

#define ADXL345\_ERROR 0 // indicates error is predent

#define ADXL345\_NO\_ERROR 0 // initial state

#define ADXL345\_READ\_ERROR 1 // problem reading accel

#define ADXL345\_BAD\_ARG 2 // bad method argument

class ADXL345

{

public:

bool status; // set when error occurs

// see error code for details

byte error\_code; // Initial state

double gains[3]; // counts to Gs

ADXL345();

void powerOn();

void readAccel(int\* xyx);

void readXYZ(int\* x, int\* y, int\* z);

void getAcceleration(double \*xyz);

void setTapThreshold(int tapThreshold);

int getTapThreshold();

void setAxisGains(double \*\_gains);

void getAxisGains(double \*\_gains);

void setAxisOffset(int x, int y, int z);

void getAxisOffset(int\* x, int\* y, int\*z);

void setTapDuration(int tapDuration);

int getTapDuration();

void setDoubleTapLatency(int doubleTapLatency);

int getDoubleTapLatency();

void setDoubleTapWindow(int doubleTapWindow);

int getDoubleTapWindow();

void setActivityThreshold(int activityThreshold);

int getActivityThreshold();

void setInactivityThreshold(int inactivityThreshold);

int getInactivityThreshold();

void setTimeInactivity(int timeInactivity);

int getTimeInactivity();

void setFreeFallThreshold(int freeFallthreshold);

int getFreeFallThreshold();

void setFreeFallDuration(int freeFallDuration);

int getFreeFallDuration();

bool isActivityXEnabled();

bool isActivityYEnabled();

bool isActivityZEnabled();

bool isInactivityXEnabled();

bool isInactivityYEnabled();

bool isInactivityZEnabled();

bool isActivityAc();

bool isInactivityAc();

void setActivityAc(bool state);

void setInactivityAc(bool state);

bool getSuppressBit();

void setSuppressBit(bool state);

bool isTapDetectionOnX();

void setTapDetectionOnX(bool state);

bool isTapDetectionOnY();

void setTapDetectionOnY(bool state);

bool isTapDetectionOnZ();

void setTapDetectionOnZ(bool state);

void setActivityX(bool state);

void setActivityY(bool state);

void setActivityZ(bool state);

void setInactivityX(bool state);

void setInactivityY(bool state);

void setInactivityZ(bool state);

bool isActivitySourceOnX();

bool isActivitySourceOnY();

bool isActivitySourceOnZ();

bool isTapSourceOnX();

bool isTapSourceOnY();

bool isTapSourceOnZ();

bool isAsleep();

bool isLowPower();

void setLowPower(bool state);

double getRate();

void setRate(double rate);

void set\_bw(byte bw\_code);

byte get\_bw\_code();

bool triggered(byte interrupts, int mask);

byte getInterruptSource();

bool getInterruptSource(byte interruptBit);

bool getInterruptMapping(byte interruptBit);

void setInterruptMapping(byte interruptBit, bool interruptPin);

bool isInterruptEnabled(byte interruptBit);

void setInterrupt(byte interruptBit, bool state);

void getRangeSetting(byte\* rangeSetting);

void setRangeSetting(int val);

bool getSelfTestBit();

void setSelfTestBit(bool selfTestBit);

bool getSpiBit();

void setSpiBit(bool spiBit);

bool getInterruptLevelBit();

void setInterruptLevelBit(bool interruptLevelBit);

bool getFullResBit();

void setFullResBit(bool fullResBit);

bool getJustifyBit();

void setJustifyBit(bool justifyBit);

void printAllRegister();

private:

void writeTo(byte address, byte val);

void readFrom(byte address, int num, byte buff[]);

void setRegisterBit(byte regAdress, int bitPos, bool state);

bool getRegisterBit(byte regAdress, int bitPos);

byte \_buff[6] ; //6 bytes buffer for saving data read from the device

};

void print\_byte(byte val);

#endif

AXDL345.cpp

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Function: Cpp file for class ADXL345

// Hardware: 3-Axis Digital Accelerometer(¡À16g)

// Arduino IDE: Arduino-1.0

// Author: Frankie.Chu

// Date: Jan 11,2013

// Version: v1.0

// by www.seeedstudio.com

//

// This library is free software; you can redistribute it and/or

// modify it under the terms of the GNU Lesser General Public

// License as published by the Free Software Foundation; either

// version 2.1 of the License, or (at your option) any later version.

//

// This library is distributed in the hope that it will be useful,

// but WITHOUT ANY WARRANTY; without even the implied warranty of

// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

// Lesser General Public License for more details.

//

// You should have received a copy of the GNU Lesser General Public

// License along with this library; if not, write to the Free Software

// Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

//

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "Arduino.h"

#include "ADXL345.h"

#include <Wire.h>

#define ADXL345\_DEVICE (0x53) // ADXL345 device address

#define ADXL345\_TO\_READ (6) // num of bytes we are going to read each time (two bytes for each axis)

ADXL345::ADXL345() {

status = ADXL345\_OK;

error\_code = ADXL345\_NO\_ERROR;

gains[0] = 0.00376390;

gains[1] = 0.00376009;

gains[2] = 0.00349265;

}

void ADXL345::powerOn() {

Wire.begin(); // join i2c bus (address optional for master)

//Turning on the ADXL345

writeTo(ADXL345\_POWER\_CTL, 0);

writeTo(ADXL345\_POWER\_CTL, 16);

writeTo(ADXL345\_POWER\_CTL, 8);

}

// Reads the acceleration into three variable x, y and z

void ADXL345::readAccel(int \*xyz){

readXYZ(xyz, xyz + 1, xyz + 2);

}

void ADXL345::readXYZ(int \*x, int \*y, int \*z) {

readFrom(ADXL345\_DATAX0, ADXL345\_TO\_READ, \_buff); //read the acceleration data from the ADXL345

// each axis reading comes in 10 bit resolution, ie 2 bytes. Least Significat Byte first!!

// thus we are converting both bytes in to one int

\*x = (((int)\_buff[1]) << 8) | \_buff[0];

\*y = (((int)\_buff[3]) << 8) | \_buff[2];

\*z = (((int)\_buff[5]) << 8) | \_buff[4];

}

void ADXL345::getAcceleration(double \*xyz){

int i;

int xyz\_int[3];

readAccel(xyz\_int);

for(i=0; i<3; i++){

xyz[i] = xyz\_int[i] \* gains[i];

}

}

// Writes val to address register on device

void ADXL345::writeTo(byte address, byte val) {

Wire.beginTransmission(ADXL345\_DEVICE); // start transmission to device

Wire.write(address); // send register address

Wire.write(val); // send value to write

Wire.endTransmission(); // end transmission

}

// Reads num bytes starting from address register on device in to \_buff array

void ADXL345::readFrom(byte address, int num, byte \_buff[]) {

Wire.beginTransmission(ADXL345\_DEVICE); // start transmission to device

Wire.write(address); // sends address to read from

Wire.endTransmission(); // end transmission

Wire.beginTransmission(ADXL345\_DEVICE); // start transmission to device

Wire.requestFrom(ADXL345\_DEVICE, num); // request 6 bytes from device

int i = 0;

while(Wire.available()) // device may send less than requested (abnormal)

{

\_buff[i] = Wire.read(); // receive a byte

i++;

}

if(i != num){

status = ADXL345\_ERROR;

error\_code = ADXL345\_READ\_ERROR;

}

Wire.endTransmission(); // end transmission

}

// Gets the range setting and return it into rangeSetting

// it can be 2, 4, 8 or 16

void ADXL345::getRangeSetting(byte\* rangeSetting) {

byte \_b;

readFrom(ADXL345\_DATA\_FORMAT, 1, &\_b);

\*rangeSetting = \_b & B00000011;

}

// Sets the range setting, possible values are: 2, 4, 8, 16

void ADXL345::setRangeSetting(int val) {

byte \_s;

byte \_b;

switch (val) {

case 2:

\_s = B00000000;

break;

case 4:

\_s = B00000001;

break;

case 8:

\_s = B00000010;

break;

case 16:

\_s = B00000011;

break;

default:

\_s = B00000000;

}

readFrom(ADXL345\_DATA\_FORMAT, 1, &\_b);

\_s |= (\_b & B11101100);

writeTo(ADXL345\_DATA\_FORMAT, \_s);

}

// gets the state of the SELF\_TEST bit

bool ADXL345::getSelfTestBit() {

return getRegisterBit(ADXL345\_DATA\_FORMAT, 7);

}

// Sets the SELF-TEST bit

// if set to 1 it applies a self-test force to the sensor causing a shift in the output data

// if set to 0 it disables the self-test force

void ADXL345::setSelfTestBit(bool selfTestBit) {

setRegisterBit(ADXL345\_DATA\_FORMAT, 7, selfTestBit);

}

// Gets the state of the SPI bit

bool ADXL345::getSpiBit() {

return getRegisterBit(ADXL345\_DATA\_FORMAT, 6);

}

// Sets the SPI bit

// if set to 1 it sets the device to 3-wire mode

// if set to 0 it sets the device to 4-wire SPI mode

void ADXL345::setSpiBit(bool spiBit) {

setRegisterBit(ADXL345\_DATA\_FORMAT, 6, spiBit);

}

// Gets the state of the INT\_INVERT bit

bool ADXL345::getInterruptLevelBit() {

return getRegisterBit(ADXL345\_DATA\_FORMAT, 5);

}

// Sets the INT\_INVERT bit

// if set to 0 sets the interrupts to active high

// if set to 1 sets the interrupts to active low

void ADXL345::setInterruptLevelBit(bool interruptLevelBit) {

setRegisterBit(ADXL345\_DATA\_FORMAT, 5, interruptLevelBit);

}

// Gets the state of the FULL\_RES bit

bool ADXL345::getFullResBit() {

return getRegisterBit(ADXL345\_DATA\_FORMAT, 3);

}

// Sets the FULL\_RES bit

// if set to 1, the device is in full resolution mode, where the output resolution increases with the

// g range set by the range bits to maintain a 4mg/LSB scal factor

// if set to 0, the device is in 10-bit mode, and the range buts determine the maximum g range

// and scale factor

void ADXL345::setFullResBit(bool fullResBit) {

setRegisterBit(ADXL345\_DATA\_FORMAT, 3, fullResBit);

}

// Gets the state of the justify bit

bool ADXL345::getJustifyBit() {

return getRegisterBit(ADXL345\_DATA\_FORMAT, 2);

}

// Sets the JUSTIFY bit

// if sets to 1 selects the left justified mode

// if sets to 0 selects right justified mode with sign extension

void ADXL345::setJustifyBit(bool justifyBit) {

setRegisterBit(ADXL345\_DATA\_FORMAT, 2, justifyBit);

}

// Sets the THRESH\_TAP byte value

// it should be between 0 and 255

// the scale factor is 62.5 mg/LSB

// A value of 0 may result in undesirable behavior

void ADXL345::setTapThreshold(int tapThreshold) {

tapThreshold = constrain(tapThreshold,0,255);

byte \_b = byte (tapThreshold);

writeTo(ADXL345\_THRESH\_TAP, \_b);

}

// Gets the THRESH\_TAP byte value

// return value is comprised between 0 and 255

// the scale factor is 62.5 mg/LSB

int ADXL345::getTapThreshold() {

byte \_b;

readFrom(ADXL345\_THRESH\_TAP, 1, &\_b);

return int (\_b);

}

// set/get the gain for each axis in Gs / count

void ADXL345::setAxisGains(double \*\_gains){

int i;

for(i = 0; i < 3; i++){

gains[i] = \_gains[i];

}

}

void ADXL345::getAxisGains(double \*\_gains){

int i;

for(i = 0; i < 3; i++){

\_gains[i] = gains[i];

}

}

// Sets the OFSX, OFSY and OFSZ bytes

// OFSX, OFSY and OFSZ are user offset adjustments in twos complement format with

// a scale factor of 15,6mg/LSB

// OFSX, OFSY and OFSZ should be comprised between

void ADXL345::setAxisOffset(int x, int y, int z) {

writeTo(ADXL345\_OFSX, byte (x));

writeTo(ADXL345\_OFSY, byte (y));

writeTo(ADXL345\_OFSZ, byte (z));

}

// Gets the OFSX, OFSY and OFSZ bytes

void ADXL345::getAxisOffset(int\* x, int\* y, int\*z) {

byte \_b;

readFrom(ADXL345\_OFSX, 1, &\_b);

\*x = int (\_b);

readFrom(ADXL345\_OFSY, 1, &\_b);

\*y = int (\_b);

readFrom(ADXL345\_OFSZ, 1, &\_b);

\*z = int (\_b);

}

// Sets the DUR byte

// The DUR byte contains an unsigned time value representing the maximum time

// that an event must be above THRESH\_TAP threshold to qualify as a tap event

// The scale factor is 625Âµs/LSB

// A value of 0 disables the tap/double tap funcitons. Max value is 255.

void ADXL345::setTapDuration(int tapDuration) {

tapDuration = constrain(tapDuration,0,255);

byte \_b = byte (tapDuration);

writeTo(ADXL345\_DUR, \_b);

}

// Gets the DUR byte

int ADXL345::getTapDuration() {

byte \_b;

readFrom(ADXL345\_DUR, 1, &\_b);

return int (\_b);

}

// Sets the latency (latent register) which contains an unsigned time value

// representing the wait time from the detection of a tap event to the start

// of the time window, during which a possible second tap can be detected.

// The scale factor is 1.25ms/LSB. A value of 0 disables the double tap function.

// It accepts a maximum value of 255.

void ADXL345::setDoubleTapLatency(int doubleTapLatency) {

byte \_b = byte (doubleTapLatency);

writeTo(ADXL345\_LATENT, \_b);

}

// Gets the Latent value

int ADXL345::getDoubleTapLatency() {

byte \_b;

readFrom(ADXL345\_LATENT, 1, &\_b);

return int (\_b);

}

// Sets the Window register, which contains an unsigned time value representing

// the amount of time after the expiration of the latency time (Latent register)

// during which a second valud tap can begin. The scale factor is 1.25ms/LSB. A

// value of 0 disables the double tap function. The maximum value is 255.

void ADXL345::setDoubleTapWindow(int doubleTapWindow) {

doubleTapWindow = constrain(doubleTapWindow,0,255);

byte \_b = byte (doubleTapWindow);

writeTo(ADXL345\_WINDOW, \_b);

}

// Gets the Window register

int ADXL345::getDoubleTapWindow() {

byte \_b;

readFrom(ADXL345\_WINDOW, 1, &\_b);

return int (\_b);

}

// Sets the THRESH\_ACT byte which holds the threshold value for detecting activity.

// The data format is unsigned, so the magnitude of the activity event is compared

// with the value is compared with the value in the THRESH\_ACT register. The scale

// factor is 62.5mg/LSB. A value of 0 may result in undesirable behavior if the

// activity interrupt is enabled. The maximum value is 255.

void ADXL345::setActivityThreshold(int activityThreshold) {

activityThreshold = constrain(activityThreshold,0,255);

byte \_b = byte (activityThreshold);

writeTo(ADXL345\_THRESH\_ACT, \_b);

}

// Gets the THRESH\_ACT byte

int ADXL345::getActivityThreshold() {

byte \_b;

readFrom(ADXL345\_THRESH\_ACT, 1, &\_b);

return int (\_b);

}

// Sets the THRESH\_INACT byte which holds the threshold value for detecting inactivity.

// The data format is unsigned, so the magnitude of the inactivity event is compared

// with the value is compared with the value in the THRESH\_INACT register. The scale

// factor is 62.5mg/LSB. A value of 0 may result in undesirable behavior if the

// inactivity interrupt is enabled. The maximum value is 255.

void ADXL345::setInactivityThreshold(int inactivityThreshold) {

inactivityThreshold = constrain(inactivityThreshold,0,255);

byte \_b = byte (inactivityThreshold);

writeTo(ADXL345\_THRESH\_INACT, \_b);

}

// Gets the THRESH\_INACT byte

int ADXL345::getInactivityThreshold() {

byte \_b;

readFrom(ADXL345\_THRESH\_INACT, 1, &\_b);

return int (\_b);

}

// Sets the TIME\_INACT register, which contains an unsigned time value representing the

// amount of time that acceleration must be less thant the value in the THRESH\_INACT

// register for inactivity to be declared. The scale factor is 1sec/LSB. The value must

// be between 0 and 255.

void ADXL345::setTimeInactivity(int timeInactivity) {

timeInactivity = constrain(timeInactivity,0,255);

byte \_b = byte (timeInactivity);

writeTo(ADXL345\_TIME\_INACT, \_b);

}

// Gets the TIME\_INACT register

int ADXL345::getTimeInactivity() {

byte \_b;

readFrom(ADXL345\_TIME\_INACT, 1, &\_b);

return int (\_b);

}

// Sets the THRESH\_FF register which holds the threshold value, in an unsigned format, for

// free-fall detection. The root-sum-square (RSS) value of all axes is calculated and

// compared whith the value in THRESH\_FF to determine if a free-fall event occured. The

// scale factor is 62.5mg/LSB. A value of 0 may result in undesirable behavior if the free-fall

// interrupt is enabled. The maximum value is 255.

void ADXL345::setFreeFallThreshold(int freeFallThreshold) {

freeFallThreshold = constrain(freeFallThreshold,0,255);

byte \_b = byte (freeFallThreshold);

writeTo(ADXL345\_THRESH\_FF, \_b);

}

// Gets the THRESH\_FF register.

int ADXL345::getFreeFallThreshold() {

byte \_b;

readFrom(ADXL345\_THRESH\_FF, 1, &\_b);

return int (\_b);

}

// Sets the TIME\_FF register, which holds an unsigned time value representing the minimum

// time that the RSS value of all axes must be less than THRESH\_FF to generate a free-fall

// interrupt. The scale factor is 5ms/LSB. A value of 0 may result in undesirable behavior if

// the free-fall interrupt is enabled. The maximum value is 255.

void ADXL345::setFreeFallDuration(int freeFallDuration) {

freeFallDuration = constrain(freeFallDuration,0,255);

byte \_b = byte (freeFallDuration);

writeTo(ADXL345\_TIME\_FF, \_b);

}

// Gets the TIME\_FF register.

int ADXL345::getFreeFallDuration() {

byte \_b;

readFrom(ADXL345\_TIME\_FF, 1, &\_b);

return int (\_b);

}

bool ADXL345::isActivityXEnabled() {

return getRegisterBit(ADXL345\_ACT\_INACT\_CTL, 6);

}

bool ADXL345::isActivityYEnabled() {

return getRegisterBit(ADXL345\_ACT\_INACT\_CTL, 5);

}

bool ADXL345::isActivityZEnabled() {

return getRegisterBit(ADXL345\_ACT\_INACT\_CTL, 4);

}

bool ADXL345::isInactivityXEnabled() {

return getRegisterBit(ADXL345\_ACT\_INACT\_CTL, 2);

}

bool ADXL345::isInactivityYEnabled() {

return getRegisterBit(ADXL345\_ACT\_INACT\_CTL, 1);

}

bool ADXL345::isInactivityZEnabled() {

return getRegisterBit(ADXL345\_ACT\_INACT\_CTL, 0);

}

void ADXL345::setActivityX(bool state) {

setRegisterBit(ADXL345\_ACT\_INACT\_CTL, 6, state);

}

void ADXL345::setActivityY(bool state) {

setRegisterBit(ADXL345\_ACT\_INACT\_CTL, 5, state);

}

void ADXL345::setActivityZ(bool state) {

setRegisterBit(ADXL345\_ACT\_INACT\_CTL, 4, state);

}

void ADXL345::setInactivityX(bool state) {

setRegisterBit(ADXL345\_ACT\_INACT\_CTL, 2, state);

}

void ADXL345::setInactivityY(bool state) {

setRegisterBit(ADXL345\_ACT\_INACT\_CTL, 1, state);

}

void ADXL345::setInactivityZ(bool state) {

setRegisterBit(ADXL345\_ACT\_INACT\_CTL, 0, state);

}

bool ADXL345::isActivityAc() {

return getRegisterBit(ADXL345\_ACT\_INACT\_CTL, 7);

}

bool ADXL345::isInactivityAc(){

return getRegisterBit(ADXL345\_ACT\_INACT\_CTL, 3);

}

void ADXL345::setActivityAc(bool state) {

setRegisterBit(ADXL345\_ACT\_INACT\_CTL, 7, state);

}

void ADXL345::setInactivityAc(bool state) {

setRegisterBit(ADXL345\_ACT\_INACT\_CTL, 3, state);

}

bool ADXL345::getSuppressBit(){

return getRegisterBit(ADXL345\_TAP\_AXES, 3);

}

void ADXL345::setSuppressBit(bool state) {

setRegisterBit(ADXL345\_TAP\_AXES, 3, state);

}

bool ADXL345::isTapDetectionOnX(){

return getRegisterBit(ADXL345\_TAP\_AXES, 2);

}

void ADXL345::setTapDetectionOnX(bool state) {

setRegisterBit(ADXL345\_TAP\_AXES, 2, state);

}

bool ADXL345::isTapDetectionOnY(){

return getRegisterBit(ADXL345\_TAP\_AXES, 1);

}

void ADXL345::setTapDetectionOnY(bool state) {

setRegisterBit(ADXL345\_TAP\_AXES, 1, state);

}

bool ADXL345::isTapDetectionOnZ(){

return getRegisterBit(ADXL345\_TAP\_AXES, 0);

}

void ADXL345::setTapDetectionOnZ(bool state) {

setRegisterBit(ADXL345\_TAP\_AXES, 0, state);

}

bool ADXL345::isActivitySourceOnX(){

return getRegisterBit(ADXL345\_ACT\_TAP\_STATUS, 6);

}

bool ADXL345::isActivitySourceOnY(){

return getRegisterBit(ADXL345\_ACT\_TAP\_STATUS, 5);

}

bool ADXL345::isActivitySourceOnZ(){

return getRegisterBit(ADXL345\_ACT\_TAP\_STATUS, 4);

}

bool ADXL345::isTapSourceOnX(){

return getRegisterBit(ADXL345\_ACT\_TAP\_STATUS, 2);

}

bool ADXL345::isTapSourceOnY(){

return getRegisterBit(ADXL345\_ACT\_TAP\_STATUS, 1);

}

bool ADXL345::isTapSourceOnZ(){

return getRegisterBit(ADXL345\_ACT\_TAP\_STATUS, 0);

}

bool ADXL345::isAsleep(){

return getRegisterBit(ADXL345\_ACT\_TAP\_STATUS, 3);

}

bool ADXL345::isLowPower(){

return getRegisterBit(ADXL345\_BW\_RATE, 4);

}

void ADXL345::setLowPower(bool state) {

setRegisterBit(ADXL345\_BW\_RATE, 4, state);

}

double ADXL345::getRate(){

byte \_b;

readFrom(ADXL345\_BW\_RATE, 1, &\_b);

\_b &= B00001111;

return (pow(2,((int) \_b)-6)) \* 6.25;

}

void ADXL345::setRate(double rate){

byte \_b,\_s;

int v = (int) (rate / 6.25);

int r = 0;

while (v >>= 1)

{

r++;

}

if (r <= 9) {

readFrom(ADXL345\_BW\_RATE, 1, &\_b);

\_s = (byte) (r + 6) | (\_b & B11110000);

writeTo(ADXL345\_BW\_RATE, \_s);

}

}

void ADXL345::set\_bw(byte bw\_code){

if((bw\_code < ADXL345\_BW\_3) || (bw\_code > ADXL345\_BW\_1600)){

status = false;

error\_code = ADXL345\_BAD\_ARG;

}

else{

writeTo(ADXL345\_BW\_RATE, bw\_code);

}

}

byte ADXL345::get\_bw\_code(){

byte bw\_code;

readFrom(ADXL345\_BW\_RATE, 1, &bw\_code);

return bw\_code;

}

//Used to check if action was triggered in interrupts

//Example triggered(interrupts, ADXL345\_SINGLE\_TAP);

bool ADXL345::triggered(byte interrupts, int mask){

return ((interrupts >> mask) & 1);

}

/\*

ADXL345\_DATA\_READY

ADXL345\_SINGLE\_TAP

ADXL345\_DOUBLE\_TAP

ADXL345\_ACTIVITY

ADXL345\_INACTIVITY

ADXL345\_FREE\_FALL

ADXL345\_WATERMARK

ADXL345\_OVERRUNY

\*/

byte ADXL345::getInterruptSource() {

byte \_b;

readFrom(ADXL345\_INT\_SOURCE, 1, &\_b);

return \_b;

}

bool ADXL345::getInterruptSource(byte interruptBit) {

return getRegisterBit(ADXL345\_INT\_SOURCE,interruptBit);

}

bool ADXL345::getInterruptMapping(byte interruptBit) {

return getRegisterBit(ADXL345\_INT\_MAP,interruptBit);

}

// Set the mapping of an interrupt to pin1 or pin2

// eg: setInterruptMapping(ADXL345\_INT\_DOUBLE\_TAP\_BIT,ADXL345\_INT2\_PIN);

void ADXL345::setInterruptMapping(byte interruptBit, bool interruptPin) {

setRegisterBit(ADXL345\_INT\_MAP, interruptBit, interruptPin);

}

bool ADXL345::isInterruptEnabled(byte interruptBit) {

return getRegisterBit(ADXL345\_INT\_ENABLE,interruptBit);

}

void ADXL345::setInterrupt(byte interruptBit, bool state) {

setRegisterBit(ADXL345\_INT\_ENABLE, interruptBit, state);

}

void ADXL345::setRegisterBit(byte regAdress, int bitPos, bool state) {

byte \_b;

readFrom(regAdress, 1, &\_b);

if (state) {

\_b |= (1 << bitPos); // forces nth bit of \_b to be 1. all other bits left alone.

}

else {

\_b &= ~(1 << bitPos); // forces nth bit of \_b to be 0. all other bits left alone.

}

writeTo(regAdress, \_b);

}

bool ADXL345::getRegisterBit(byte regAdress, int bitPos) {

byte \_b;

readFrom(regAdress, 1, &\_b);

return ((\_b >> bitPos) & 1);

}

// print all register value to the serial ouptut, which requires it to be setup

// this can be used to manually to check the current configuration of the device

void ADXL345::printAllRegister() {

byte \_b;

Serial.print("0x00: ");

readFrom(0x00, 1, &\_b);

print\_byte(\_b);

Serial.println("");

int i;

for (i=29;i<=57;i++){

Serial.print("0x");

Serial.print(i, HEX);

Serial.print(": ");

readFrom(i, 1, &\_b);

print\_byte(\_b);

Serial.println("");

}

}

void print\_byte(byte val){

int i;

Serial.print("B");

for(i=7; i>=0; i--){

Serial.print(val >> i & 1, BIN);

}

}

Breakout2.cpp

#include <Wire.h>

#include "ADXL345.h"

#include <Bounce2.h>

ADXL345 adxl; //variable adxl is an instance of the ADXL345 library

int x, y, z;//added with v 2.0

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 = 9600;

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);

//added with v2.0

adxl.powerOn();

//set activity/ inactivity thresholds (0-255)

adxl.setActivityThreshold(75); //62.5mg per increment

adxl.setInactivityThreshold(75); //62.5mg per increment

adxl.setTimeInactivity(10); // how many seconds of no activity is inactive?

//look of activity movement on this axes - 1 == on; 0 == off

adxl.setActivityX(1);

adxl.setActivityY(1);

adxl.setActivityZ(1);

//look of inactivity movement on this axes - 1 == on; 0 == off

adxl.setInactivityX(1);

adxl.setInactivityY(1);

adxl.setInactivityZ(1);

//look of tap movement on this axes - 1 == on; 0 == off

adxl.setTapDetectionOnX(0);

adxl.setTapDetectionOnY(0);

adxl.setTapDetectionOnZ(1);

//set values for what is a tap, and what is a double tap (0-255)

adxl.setTapThreshold(50); //62.5mg per increment

adxl.setTapDuration(15); //625us per increment

adxl.setDoubleTapLatency(80); //1.25ms per increment

adxl.setDoubleTapWindow(200); //1.25ms per increment

//set values for what is considered freefall (0-255)

adxl.setFreeFallThreshold(7); //(5 - 9) recommended - 62.5mg per increment

adxl.setFreeFallDuration(45); //(20 - 70) recommended - 5ms per increment

//setting all interrupts to take place on int pin 1

//I had issues with int pin 2, was unable to reset it

adxl.setInterruptMapping( ADXL345\_INT\_SINGLE\_TAP\_BIT, ADXL345\_INT1\_PIN );

adxl.setInterruptMapping( ADXL345\_INT\_DOUBLE\_TAP\_BIT, ADXL345\_INT1\_PIN );

adxl.setInterruptMapping( ADXL345\_INT\_FREE\_FALL\_BIT, ADXL345\_INT1\_PIN );

adxl.setInterruptMapping( ADXL345\_INT\_ACTIVITY\_BIT, ADXL345\_INT1\_PIN );

adxl.setInterruptMapping( ADXL345\_INT\_INACTIVITY\_BIT, ADXL345\_INT1\_PIN );

//register interrupt actions - 1 == on; 0 == off

adxl.setInterrupt( ADXL345\_INT\_SINGLE\_TAP\_BIT, 1);

adxl.setInterrupt( ADXL345\_INT\_DOUBLE\_TAP\_BIT, 1);

adxl.setInterrupt( ADXL345\_INT\_FREE\_FALL\_BIT, 1);

adxl.setInterrupt( ADXL345\_INT\_ACTIVITY\_BIT, 1);

adxl.setInterrupt( ADXL345\_INT\_INACTIVITY\_BIT, 1);

//end v 2.0 additions

}

int get\_axis(const int axis) {

delay(1);

//buffer[axis][buffer\_pos[axis]] = analogRead(PINS[axis]); //superceded v2.0

buffer\_pos[axis] = (buffer\_pos[axis] + 1) % BUFFER\_SIZE;

long sum = 0;

for (int i = 0; i < BUFFER\_SIZE; i++)

sum += buffer[axis][i];

return round(sum / BUFFER\_SIZE);

}

//superceded by v2.0

//int get\_x() { return get\_axis(0); }

//int get\_y() { return get\_axis(1); }

//int get\_z() { return get\_axis(2); }

void loop() {

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//New code for v 2.0

adxl.readXYZ(&x, &y, &z);

//v2.0: we read them all at once, and assign them to x, y, z

Serial.print(x);

Serial.print(" ");

Serial.print(y);

Serial.print(" ");

Serial.print(z);

Serial.print(" ");

/\* //Superceded by v 2.0

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

Serial.println("0");

}