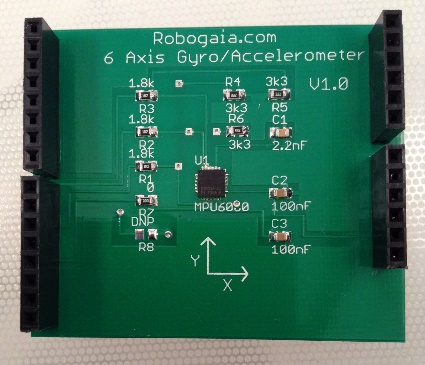
OSVR: Sensics latency-testing hardware

The Sensics latency testing hardware is constructed using an Arduino, an Arduino gyro/accelerometer shield, and one or more photosensor inputs.

## Arduino

We built the prototype unit using an Arduino Uno board. The particular board we used was the SainSmart UNO ATmega328P, which was purchased through Amazon.com at <http://smile.amazon.com/SainSmart-ATmega328P-CABLE-Included-Arduino/dp/B006GX8IAY/ref=sr_1_5?s=electronics&ie=UTF8&qid=1422119284&sr=1-5&keywords=arduino+uno> for $17.69.

## Gyro/accelerometer

We built the prototype unit using a 6-axis accelerometer gyro Arduino shield module made by Robogaia.com. It was purchased through Amazon.com at <http://smile.amazon.com/Axis-Accelerometer-Gyro-Arduino-Shield/dp/B00GLCEXDG/ref=lh_ni_t?ie=UTF8&psc=1&smid=A3CCYN6649JF9L> for $39.99 plus $4.49 shipping with an estimated delivery time of 6-25 days. It is designed to work directly with the Arduino Uno and can be adapted by adding wiring to work with the Arduino Mega 2560.

**Installation:** The shield plugs directly into the top of the Uno board.

## Photosensors

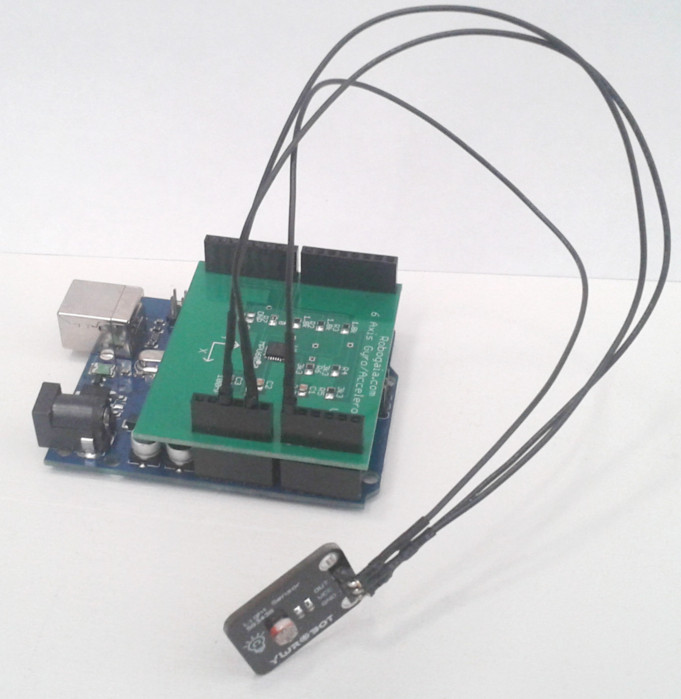
We built the prototype using a wRobot light sensor through Amazon.com at <http://smile.amazon.com/Phantom-YoYo-Arduino-compatible-Sensor/dp/B00AF278A8/ref=sr_1_1?ie=UTF8&qid=1422394751&sr=8-1&keywords=arduino+light+sensor> for $6.99.



You need to find some jumper cables to connect the pins on the photosensors to the pins on the Uno (actually on the shield’s pass-through connectors). For the prototype, we bought some at Radio Shack, but you can also purchase them online. You want make-to-female jumpers like the ones at <http://smile.amazon.com/Male-Female-Jumper-Cable-Wires/dp/B00D7SDDLU/ref=sr_1_7?ie=UTF8&qid=1422653254&sr=8-7&keywords=jumper+wires+arduino> from Amazon.

**Installation:** Connect the GND pin on the light sensor to GND on the Arduino. Connect the VCC pin on the light sensor to 5V on the Arduino. Connect the OUT pin on the light sensor to ANALOG IN A0 on the Arduino.

## Hardware testing



The fully assembled unit is shown to the right.

To enable use viewing a head-mounted display, you will want to attach the photosensor to the back of the unit (the bottom in the picture) with the sensor facing away from the unit. **Important:** Place a non-conductive element between the back of the sensor and the back of the unit, perhaps some sort of adhesive putty. To enable viewing the latency on the screen, you will want to use long jumper cables and put the sensor on a stand that keeps it fixed in space fairly close to the display screen.

You can test the analog sensor by running the ***File/Examples/01.Basics/ReadAnalogVoltage*** Sketch program in the Arduino interface, and watching the output on the ***Tools/Serial Monitor*** that comes built in. Move the photosensor in front of bright and dark locations on the screen and you should see a measurable difference in voltage (perhaps a range of 3V to 4.12V).

You can test the accelerometers and gyroscopes by running the ***Accelerometer\_Gyro\_Shield\_test.ino*** program that you can download from the vendor site at <http://www.robogaia.com/6-axis-accelerometer-gyro-arduino-shield.html> in the ***Example Code*** section. Use the serial monitor to view output. As you flip the unit over, the accelerometer readings for the Z axis should change from negative several thousand to positive several thousand. As you rapidly rotate the unit, you should see the gyro values change.

## Programming the Arduino

When you plug the Arduino into Linux, it is immediately recognized as a virtual serial device and will show up in /dev/serial/by-id as a device with the name Arduino\_Uno in it (and may also appear as /dev/ttyACM0).

When you plug the Arduino into Windows 8, it does not immediately recognize it without a driver. You can download the Arduino software from the <http://Arduino.cc> web page. We used version 1.0.6 of the software to develop the latency-measuring prototype. Note that the software is GPL. Be sure to unplug your Arduino before installing the software (which includes a driver) and then plug it in again when the software has been installed. At this point, the device will show up as a COM port.

Sensics has developed an ***OSVR\_Latency\_hardware\_firmware.ino*** program to be downloaded and run on the prototype system. This program waits for a period of no motion of the inertial measurement unit followed by a sudden motion (acceleration and/or rotation). It then looks for a change in brightness in the photodetector signal and reports the duration in microseconds between the onset of motion and the change of brightness.

To load the program onto the Arduino, install the Arduino development environment and then open the .ino file. Press the run button (right arrow in version 1.0.6) to compile, load and run the program.

To run the program, open *Tools/Serial Monitor*. It should print a version number and give instructions. Rigidly attach the photosensor near a fixed location on the display screen or HMD. Rigidly attach the inertial measurement system (the body of the Arduino along with its shield) to the tracking device or mouse. Hold the tracking unit still and then move it suddenly in a way that will cause the part of the image in front of the photosensor to change its brightness dramatically in response to the motion. The program will report the number of microseconds between the detection of motion and the detection of a change it brightness. You probably want a situation where you can repeatedly rotate or move back and forth and have the brightness transition, so that you can get a number of measurements.

When the program runs, it first makes sure that the inertial measurement unit is held still for half a second; during the time it is waiting for the unit to become still, the LED will flash rapidly, looking like it is dim.

Once it has settled, it waits for a sudden motion. While it is waiting for this, the LED will be bright.

As soon as it detects the motion, the LED goes off while it waits for the brightness to change. If the brightness does not change for a second, it times out and goes back to looking for the unit to be held still. If it does detect a change, it reports the latency between the motion and the change in brightness and goes back for waiting for the unit to be held still.

Every 16 measurements, the program computes the average latency in microseconds for the previous 8 odd measurements, and also for the previous 8 even measurements (to support modes where there is an on/off or back/forth test being done). Timeouts are ignored when computing this average.

**Tweaks:** The motion and brightness thresholds are stored as constants in the program, with names *GYRO\_THRESHOLD*, *ACCEL\_CHANGE\_THRESHOLD*, and *BRIGHTNESS\_CHANGE\_THRESHOLD*. You can adjust these as needed to provide additional sensitivity (or more robustness in the presence of noise) and then re-run the program. The number of delays to wait before averaging is stored in *NUM\_DELAYS* and should be twice as many as will be averaged, since there is a separate even and odd measurement.

**Note:** The Arduino is programmed in a language called “Sketch”, which is converted into C/C++ code and then sent to the board. When you compile and upload a program to the unit, it will continue to run that program even if the reset button is pushed or the USB power is removed and restored. The Sensics app works fills in the *setup()* and *loop()* functions with the Sketch code to be run at boot time and repeatedly. Any variables that need to persist between loop runs must be declared static.

**Note:** The LED pin (pin 13) on the board is also used by the SPI communication protocol, so it is not available to be used for signaling when SPI communication is being used (to communicate to the gyro/accelerometer shield.

## Client programs

It is not necessary to run a separate client-side program to test the end-to-end latency of an application. The Arduino serial monitor can be used to report this latency as described above.

Sensics has also developed specialized client-side programs to test the latency of various subsystems. When used in conjunction with XXX.

XXX

The client-side program uses the virtual serial port on the USB connection to talk with the device.

**Note:** the Arduino resets itself when its USB port is plugged in, at least on a Mac and Linux. To avoid having command characters lost to the reset, the client code waits until it has been able to read at least one report from the device before attempting to send the first command to it.

XXX

## Appendix A: Photosensor latency testing

There is a time constant associated with the photosensor that depends on the R/C filter associated with the detector. To test this latency for the particular photosensor you are using, a *Photodiode\_latency\_test.ino* program has been developed. To use this program, point the photosensor at the LED on the Arduino board and then run the program. Open ***Tools/Serial Monitor*** on the Arduino development environment after running the program to see what these values are. You can then set these values in the client program based on the output of the test.

The particular unit used in the prototype had a fairly long time constant, so that operating the unit with a threshold halfway between the bright and dark value produced a latency of around 1-2ms for an LED-on event and of about 18-35ms for an LED-off event. This value depends on the ambient brightness and the difference between bright and dark levels for the photodiode. When operating with a threshold that is 1/10th of the way from the off value to the on value, the on latency is 0.4ms and the off latency is 1.8ms for a fairly bright ambient; they are 0.5ms and 6ms for a dark ambient. When operating with a threshold that is 1/20th of the way from on to off, the latencies are 0.4ms and 3ms for a dark ambient; they are 0.28 and 0.7ms for a bright ambient. All of these values include a 0.14ms loop delay (how long it takes for the Arduino to read the analog value and go once more through its loop function).

The algorithm used for latency testing uses the 1/20th of the full-scale threshold so that the photodiode latency will be a small fraction of the total system latency. It also includes parameters that can be set based on testing for a particular photosensor.

## Appendix B: End-to-end latency testing

We tested the prototype by holding the photodiode and the system board together in one hand and rapidly moving the entire unit so that the photosensor was moving out from behind an occluder and viewing a light. When moving as suddenly as possible by hand, latencies of as low as 1ms were observed. Latencies of as low as 2ms were observed when moving towards darker regions. As expected from by-hand testing, the measurements were variable and were sometimes much larger (as much as 30ms).

Adhesive putty was used to attach the photo sensor to the body of the unit and the unit was placed on a table top near the edge of an overhang that blocked the light. It was repeatedly tapped sharply with a fingernail to move it into the darkness, resulting in reliably low measurements (2.8ms, 3.1ms, 1.7ms, 2.4ms, 3.7ms). The variability in the measurements may still be attributable to variations in initial starting position and strength of tap.

When holding the unit and rotating it so that the photo sensor went into and out of the light, slightly higher latencies (3-6ms) were observed, probably due to the rotational inertial of the hand/Ardiuno/cord system being higher relative to muscle strength than the translational inertia.

With the system sitting above the photosensor between it and the light, when we tapped the system slightly so that the amount of shading increased a bit, we saw the following latencies (ms): 23, 20, 16, 14, 23, 11, 17, 11, 12, 6 (for varying-strength taps). For a series of strong taps with the unit positioned to partly shadow the sensor (ms): 8.5, 9.4, 8.4, 6.9, 8.8, 7.9, 8.3.

Switching the photosensor threshold from 5 down to 3 (version 2.1.0 of the software) produced latencies in the unit positioned to partly shadow the sensor of (ms): 6.8, 6.2, 8.3, 7.3, 6.9, 6.9. For the fingernail-tap into the darkness under an overhang test, the latencies were (ms): 1.0, 1.0, 1.2, 1.2, 1.1, 1.1, 0.9, 1.3, 0.8, 0.8.

## Appendix C: Source code

The source code to version 3.0.0 of the latency tester is found below.

// OSRVR latency-testing hardware firmware source code.

// Author: Russell Taylor working for Sensics.com through ReliaSolve.com

// LICENSE: Apache License 2.0.

// This code is based on the "Accelerometer\_Gyro\_Shield\_test" example from

// Robogaia industries that ships along with their 6-axis Gyro/Accelerometer

// shield. That source is available from Robogaia.com.

// This code also uses information from the Blink example modified by Scott

// Fitzgerald that came with the Arduino.

// This code uses information from the AnalogReadSerial example that

// came with the Arduino.

//some other examples for MPU6000 code

//http://arduino.cc/en/Tutorial/BarometricPressureSensor

//http://code.google.com/p/ardu-imu/downloads/detail?name=ArduIMU\_1.9.zip&can=2&q=

/\*

Circuit:

MPU6000 sensor attached to the SPI communications circuit using pins 10 - 13:

NOTE: Pin 13 is also the pin that controls the on-board LED, so it cannot be

used for LED indication when SPI communication is happening.

CSB: pin 10

MOSI: pin 11

MISO: pin 12

SCK: pin 13

\*/

// the sensor communicates using SPI, so include the library:

#include <SPI.h>

//MPU 6000 register addresses

const int ACCEL\_XOUT\_H = 0x3B; //59 R ACCEL\_XOUT[15:8]

const int ACCEL\_XOUT\_L = 0x3C; //60 R ACCEL\_XOUT[7:0]

const int ACCEL\_YOUT\_H = 0x3D; //61 R ACCEL\_YOUT[15:8]

const int ACCEL\_YOUT\_L = 0x3E; //62 R ACCEL\_YOUT[7:0]

const int ACCEL\_ZOUT\_H = 0x3F; //63 R ACCEL\_ZOUT[15:8]

const int ACCEL\_ZOUT\_L = 0x40; //64 R ACCEL\_ZOUT[7:0]

const int TEMP\_OUT\_H = 0x41; //65 R TEMP\_OUT[15:8]

const int TEMP\_OUT\_L = 0x42; //66 R TEMP\_OUT[7:0]

const int GYRO\_XOUT\_H = 0x43; //67 R GYRO\_XOUT[15:8]

const int GYRO\_XOUT\_L = 0x44; //68 R GYRO\_XOUT[7:0]

const int GYRO\_YOUT\_H = 0x45; //69 R GYRO\_YOUT[15:8]

const int GYRO\_YOUT\_L = 0x46; //70 R GYRO\_YOUT[7:0]

const int GYRO\_ZOUT\_H = 0x47; //71 R GYRO\_ZOUT[15:8]

const int GYRO\_ZOUT\_L = 0x48; //72 RGYRO\_ZOUT[7:0]

const int READ\_FLAG =0x80; //128 has to be added to the address register

// MPU 6000 registers

#define MPUREG\_WHOAMI 0x75 //

#define MPUREG\_SMPLRT\_DIV 0x19 //

#define MPUREG\_CONFIG 0x1A //

#define MPUREG\_GYRO\_CONFIG 0x1B

#define MPUREG\_ACCEL\_CONFIG 0x1C

#define MPUREG\_INT\_PIN\_CFG 0x37

#define MPUREG\_INT\_ENABLE 0x38

#define MPUREG\_ACCEL\_XOUT\_H 0x3B //

#define MPUREG\_ACCEL\_XOUT\_L 0x3C //

#define MPUREG\_ACCEL\_YOUT\_H 0x3D //

#define MPUREG\_ACCEL\_YOUT\_L 0x3E //

#define MPUREG\_ACCEL\_ZOUT\_H 0x3F //

#define MPUREG\_ACCEL\_ZOUT\_L 0x40 //

#define MPUREG\_TEMP\_OUT\_H 0x41//

#define MPUREG\_TEMP\_OUT\_L 0x42//

#define MPUREG\_GYRO\_XOUT\_H 0x43 //

#define MPUREG\_GYRO\_XOUT\_L 0x44 //

#define MPUREG\_GYRO\_YOUT\_H 0x45 //

#define MPUREG\_GYRO\_YOUT\_L 0x46 //

#define MPUREG\_GYRO\_ZOUT\_H 0x47 //

#define MPUREG\_GYRO\_ZOUT\_L 0x48 //

#define MPUREG\_USER\_CTRL 0x6A //

#define MPUREG\_PWR\_MGMT\_1 0x6B //

#define MPUREG\_PWR\_MGMT\_2 0x6C //

// Configuration bits MPU 6000

#define BIT\_SLEEP 0x40

#define BIT\_H\_RESET 0x80

#define BITS\_CLKSEL 0x07

#define MPU\_CLK\_SEL\_PLLGYROX 0x01

#define MPU\_CLK\_SEL\_PLLGYROZ 0x03

#define MPU\_EXT\_SYNC\_GYROX 0x02

#define BITS\_FS\_250DPS 0x00

#define BITS\_FS\_500DPS 0x08

#define BITS\_FS\_1000DPS 0x10

#define BITS\_FS\_2000DPS 0x18

#define BITS\_FS\_2G 0x00

#define BITS\_FS\_4G 0x08

#define BITS\_FS\_8G 0x10

#define BITS\_FS\_16G 0x18

#define BITS\_FS\_MASK 0x18

#define BITS\_DLPF\_CFG\_256HZ\_NOLPF2 0x00

#define BITS\_DLPF\_CFG\_188HZ 0x01

#define BITS\_DLPF\_CFG\_98HZ 0x02

#define BITS\_DLPF\_CFG\_42HZ 0x03

#define BITS\_DLPF\_CFG\_20HZ 0x04

#define BITS\_DLPF\_CFG\_10HZ 0x05

#define BITS\_DLPF\_CFG\_5HZ 0x06

#define BITS\_DLPF\_CFG\_2100HZ\_NOLPF 0x07

#define BITS\_DLPF\_CFG\_MASK 0x07

#define BIT\_INT\_ANYRD\_2CLEAR 0x10

#define BIT\_RAW\_RDY\_EN 0x01

#define BIT\_I2C\_IF\_DIS 0x10

#undef VERBOSE

#undef VERBOSE2

// pins used for the connection with the sensor

// the other you need are controlled by the SPI library):

const int chipSelectPin = 10;

// Thresholds

const int GYRO\_THRESHOLD = 300;

const int ACCEL\_CHANGE\_THRESHOLD = 500;

const int BRIGHTNESS\_CHANGE\_THRESHOLD = 3;

const unsigned long TIMEOUT\_USEC = 1000000L;

// Keeps track of delays so we can do an average.

const int NUM\_DELAYS = 16;

unsigned long delays[NUM\_DELAYS];

static int count = 0, odd\_count = 0, even\_count = 0;

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

void setup()

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

{

Serial.begin(9600);

Serial.print("OSVR\_latency\_hardware\_firmware v03.00.00\n");

Serial.print(" Mount the photosensor rigidly on the screen.\n");

Serial.print(" Move the interial sensor along with the tracking hardware.\n");

Serial.print(" Make the app change the brightness in front of the photosensor.\n");

Serial.print(" Latencies reported in microseconds, 1-second timeout\n");

MPU6000\_Init();

delay(100);

}

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

void loop()

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

{

// We run a finite-state machine that cycles through cases of waiting for

// a period of non motion, detecting a sudden motion, waiting until there

// is a change in brightness, and reporting the time passed in microseconds

// between the motion and the brightness change.

static enum { S\_CALM, S\_MOTION, S\_BRIGHTNESS } state = S\_CALM;

static unsigned long start;

static int initial\_brightness;

switch (state) {

case S\_CALM:

{

// Wait for a period of at least 100 cycles where there is no motion above

// the motion threshold.

static int calm\_cycles = 0;

if (moving2()) {

calm\_cycles = 0;

} if (++calm\_cycles >= 50) {

state = S\_MOTION;

calm\_cycles = 0;

} else {

// Make sure we stay calm for a while (half a second)

delay(10);

}

}

break;

case S\_MOTION:

{

// Wait for a sudden motion. When we find it, record the time in microseconds

// so we can compare it to when the brightness changes. Also record the brightness

// so we can look for changes.

if (moving2()) {

start = micros();

initial\_brightness = analogRead(A0);

state = S\_BRIGHTNESS;

#ifdef VERBOSE

Serial.print("Moving\n");

#endif

}

}

break;

case S\_BRIGHTNESS:

{

// Wait for a change in brightness compared to the original value that

// passes a threshold. When we get it, report the latency.

// If it takes too long, then we time out and start over.

int brightness = analogRead(A0);

unsigned long now = micros();

// Keep track of how many values we got for odd and even rows and

// compute a running average when we get a full complement for each.

// Ignore timeout values

if (abs(brightness - initial\_brightness) > BRIGHTNESS\_CHANGE\_THRESHOLD) {

// Print the result for this time

Serial.print(now - start);

Serial.print("\n");

if (count % 2 == 0) {

odd\_count++; // This is the first one (zero indexed) or off by twos

} else {

even\_count++;

}

delays[count++] = now - start;

state = S\_CALM;

} else if (now - start > TIMEOUT\_USEC) {

Serial.print("Timeout: no brightness change after motion, restarting\n");

// We don't increment the counter and we set the reading to 0 so we ignore it.

delays[count++] = 0;

state = S\_CALM;

}

// See if it is time to print the average result.

if (count == NUM\_DELAYS) {

unsigned long even\_average = 0;

unsigned long odd\_average = 0;

for (int i = 0; i < NUM\_DELAYS/2; i++) {

odd\_average += delays[2\*i];

even\_average += delays[2\*i+1];

}

odd\_average /= odd\_count;

Serial.print("Average of last ");

Serial.print(NUM\_DELAYS/2);

Serial.print(" odd counts (ignoring timeouts) = ");

Serial.print(odd\_average);

Serial.print("\n");

even\_average /= even\_count;

Serial.print("Average of last ");

Serial.print(NUM\_DELAYS/2);

Serial.print(" even counts (ignoring timeouts) = ");

Serial.print(even\_average);

Serial.print("\n");

count = 0;

odd\_count = 0;

even\_count = 0;

}

}

break;

default:

Serial.print("Error: Unrecognized state; restarting\n");

break;

}

}

// Determine whether we're undergoing acceleration or rotation.

// Does this by comparing the gyros to a threshold and

// comparing accelerations to see if they are changing.

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

inline bool moving(void)

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

{

byte accX, accY, accZ;

byte gyroX, gyroY, gyroZ;

getAccelerometerAndGyroHighCounts(accX, accY, accZ, gyroX, gyroY, gyroZ);

// If the gyros are above threshold, then we're moving.

if (gyroX > GYRO\_THRESHOLD) { return true; }

if (gyroY > GYRO\_THRESHOLD) { return true; }

if (gyroZ > GYRO\_THRESHOLD) { return true; }

// See if the accelerations are sufficiently different from the

// ones we read last time. If so, we're moving.

static int prevX = 0;

static int prevY = 0;

static int prevZ = 0;

bool ret = false;

if ( abs(accX - prevX) > ACCEL\_CHANGE\_THRESHOLD ) {

ret = true;

}

if ( abs(accY - prevY) > ACCEL\_CHANGE\_THRESHOLD ) {

ret = true;

}

if ( abs(accZ - prevZ) > ACCEL\_CHANGE\_THRESHOLD ) {

ret = true;

}

prevX = accX;

prevY = accY;

prevZ = accZ;

return ret;

}

// Determine whether we're undergoing acceleration or rotation.

// Does this by comparing the gyros to a threshold and

// comparing accelerations to see if they are changing.

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

inline bool moving2(void)

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

{

bool ret = false;

int accX, accY, accZ;

int gyroX, gyroY, gyroZ;

accX = getXAccelerometerCounts();

accY = getYAccelerometerCounts();

accZ = getZAccelerometerCounts();

gyroX = getXGyroCounts();

gyroY = getYGyroCounts();

gyroZ = getZGyroCounts();

#ifdef VERBOSE2

/\* Debugging info to help figure out what the thresholds should be \*/

Serial.print(" "); Serial.print(accX);

Serial.print(" "); Serial.print(accY);

Serial.print(" "); Serial.print(accZ);

Serial.print(", "); Serial.print(gyroX);

Serial.print(" "); Serial.print(gyroY);

Serial.print(" "); Serial.print(gyroZ);

Serial.print(", "); Serial.print(analogRead(A0));

Serial.print("\n");

delay(1000);

#endif

// If the gyros are above threshold, then we're moving.

if (gyroX > GYRO\_THRESHOLD) {

#ifdef VERBOSE

Serial.print("gX\n");

#endif

ret = true;

}

if (gyroY > GYRO\_THRESHOLD) {

#ifdef VERBOSE

Serial.print("gY\n");

#endif

ret = true;

}

if (gyroZ > GYRO\_THRESHOLD) {

#ifdef VERBOSE

Serial.print("gZ\n");

#endif

ret = true;

}

// See if the accelerations are sufficiently different from the

// ones we read last time. If so, we're moving.

static int prevX = 0;

static int prevY = 0;

static int prevZ = 0;

if ( abs(accX - prevX) > ACCEL\_CHANGE\_THRESHOLD ) {

#ifdef VERBOSE

Serial.print("aX\n");

#endif

ret = true;

}

if ( abs(accY - prevY) > ACCEL\_CHANGE\_THRESHOLD ) {

#ifdef VERBOSE

Serial.print("aY\n");

#endif

ret = true;

}

if ( abs(accZ - prevZ) > ACCEL\_CHANGE\_THRESHOLD ) {

#ifdef VERBOSE

Serial.print("aZ\n");

#endif

ret = true;

}

prevX = accX;

prevY = accY;

prevZ = accZ;

#undef VERBOSE

return ret;

}

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

// We only read the high-order bytes for two reasons:

// 1) Speed.

// 2) To avoid getting a measurement whose upper and lower bytes are

// inconsisent.

// XXX However, this routine did not work. It is not clear why it

// did not, but switching back to using the original read routines

// through the moving2() call.

inline void getAccelerometerAndGyroHighCounts(byte &accX, byte &accY, byte &accZ,

byte &gyroX, byte &gyroY, byte &gyroZ)

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

{

accX = readRegister(ACCEL\_XOUT\_H);

accY = readRegister(ACCEL\_YOUT\_H);

accZ = readRegister(ACCEL\_ZOUT\_H);

gyroX = readRegister(GYRO\_XOUT\_H);

gyroY = readRegister(GYRO\_YOUT\_H);

gyroZ = readRegister(GYRO\_ZOUT\_H);

}

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

int getXAccelerometerCounts(void)

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

{

int tempData\_HI,tempData\_LO;

//Read the data

tempData\_HI = readRegister(ACCEL\_XOUT\_H);

tempData\_LO = readRegister(ACCEL\_XOUT\_L );

return ((tempData\_HI << 8) | tempData\_LO);

}//end func

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

int getYAccelerometerCounts(void)

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

{

int tempData\_HI,tempData\_LO;

tempData\_HI = readRegister(ACCEL\_YOUT\_H);

tempData\_LO = readRegister(ACCEL\_YOUT\_L );

return ((tempData\_HI << 8) + tempData\_LO);

}//end func

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

int getZAccelerometerCounts(void)

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

{

int tempData\_HI,tempData\_LO;

tempData\_HI = readRegister(ACCEL\_ZOUT\_H );

tempData\_LO = readRegister(ACCEL\_ZOUT\_L );

return ((tempData\_HI << 8) + tempData\_LO);

}//end func

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

int getXGyroCounts(void)

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

{

int tempData\_HI,tempData\_LO;

tempData\_HI = readRegister(GYRO\_XOUT\_H );

tempData\_LO = readRegister(GYRO\_XOUT\_L );

return ((tempData\_HI << 8) + tempData\_LO);

}//end func

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

int getYGyroCounts(void)

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

{

int tempData\_HI,tempData\_LO;

tempData\_HI = readRegister(GYRO\_YOUT\_H);

tempData\_LO = readRegister(GYRO\_YOUT\_L );

return ((tempData\_HI << 8) + tempData\_LO);

}//end func

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

int getZGyroCounts(void)

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

{

int tempData\_HI,tempData\_LO;

tempData\_HI = readRegister(GYRO\_ZOUT\_H);

tempData\_LO = readRegister(GYRO\_ZOUT\_L);

return ((tempData\_HI << 8) + tempData\_LO);

}//end func

//Read from register

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

unsigned int readRegister(byte thisRegister)

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

{

unsigned int result = 0; // result to return

byte addr = thisRegister + 0x80;

// Serial.print(thisRegister, BIN);

// byte dataToSend = thisRegister ;

//Serial.println(thisRegister, BIN);

// take the chip select low to select the device:

digitalWrite(chipSelectPin, LOW);

// send the device the register you want to read:

SPI.transfer(addr);

// send a value of 0 to read the first byte returned:

result = SPI.transfer(0x00);

// take the chip select high to de-select:

digitalWrite(chipSelectPin, HIGH);

// return the result:

return(result);

}

//Write to register

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

void writeRegister(byte thisRegister, byte thisValue)

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

{

thisRegister = thisRegister ;

// now combine the register address and the command into one byte:

byte dataToSend = thisRegister;

// take the chip select low to select the device:

digitalWrite(chipSelectPin, LOW);

SPI.transfer(dataToSend); //Send register location

SPI.transfer(thisValue); //Send value to record into register

// take the chip select high to de-select:

digitalWrite(chipSelectPin, HIGH);

}//

// MPU6000 Initialization and configuration

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

void MPU6000\_Init(void)

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

{

// MPU6000 chip select setup

pinMode(chipSelectPin, OUTPUT);

digitalWrite(chipSelectPin, HIGH);

// SPI initialization

SPI.begin();

SPI.setClockDivider(SPI\_CLOCK\_DIV16); // SPI at 1Mhz (on 16Mhz clock)

delay(10);

// Chip reset

writeRegister(MPUREG\_PWR\_MGMT\_1, BIT\_H\_RESET);

delay(100);

// Wake up device and select GyroZ clock (better performance)

writeRegister(MPUREG\_PWR\_MGMT\_1, MPU\_CLK\_SEL\_PLLGYROZ);

delay(1);

// Disable I2C bus (recommended on datasheet)

writeRegister(MPUREG\_USER\_CTRL, BIT\_I2C\_IF\_DIS);

delay(1);

// Set the sampling rate. We do this by setting the divisor on the rate,

// which has a base update rate of 1kHz. The number we use here is added

// to 1 and then used as a divisor: Rate = 1Khz / (value + 1). So a value

// of 0 is 1 kHz, a value of 4 is 200Hz, and a value of 19 is 50Hz.

// We want to sample as rapidly as possible, so that we have the minimum

// latency between motion and its detection. Even at 1kHz, we're going to

// have some fraction of a second of latency on our reads.

writeRegister(MPUREG\_SMPLRT\_DIV,0);

delay(1);

// Set the filter pass frequency on the low-pass filter to the maximum

// (no filter, corresponding to a 2.1Khz cutoff).

writeRegister(MPUREG\_CONFIG, BITS\_DLPF\_CFG\_2100HZ\_NOLPF);

delay(1);

// Set the measurement scale on the gyros and the accelerometers.

// From the data sheet: For precision tracking of both fast and slow

// motions, the parts feature a user-programmable gyroscope full-scale range

// of ±250, ±500, ±1000, and ±2000°/sec (dps) and a user-programmable accelerometer

// full-scale range of ±2g, ±4g, ±8g, and ±16g.

// We set the sensitivity to match the example app, which got good full-scale results.

writeRegister(MPUREG\_GYRO\_CONFIG,BITS\_FS\_2000DPS);

delay(1);

writeRegister(MPUREG\_ACCEL\_CONFIG,BITS\_FS\_4G);

delay(1);

// Oscillator set

writeRegister(MPUREG\_PWR\_MGMT\_1,MPU\_CLK\_SEL\_PLLGYROZ);

delay(1);

}