[REAL-TIME EMBEDDED SYSTEM FOR PEDOMETER APPLICATION]

Nijaporn Hotrabhavananda

**Abstract**

The compact choices make the big difference, and tracking your steps with a pedometer can motivate you to make more active choices by making a few changes in your routine to pursue a better life. This project illustrates an embedded system for a pedometer application. The system tracks steps with a MMA7455 3-axis accelerometer module.

**Introduction**

The better choices make the big difference, and tracking your steps with a pedometer can motivate every age ranges to make more active choices by making a few changes in your routine. By choosing the stairs over the elevator or walking to the train instead of driving to work. Pedometer boosts your awareness about your habits and activity level, and when it comes to being active, knowledge is power.

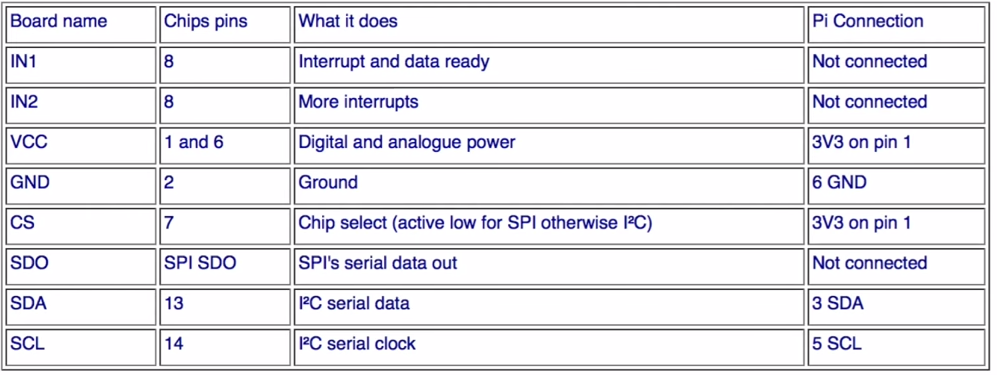
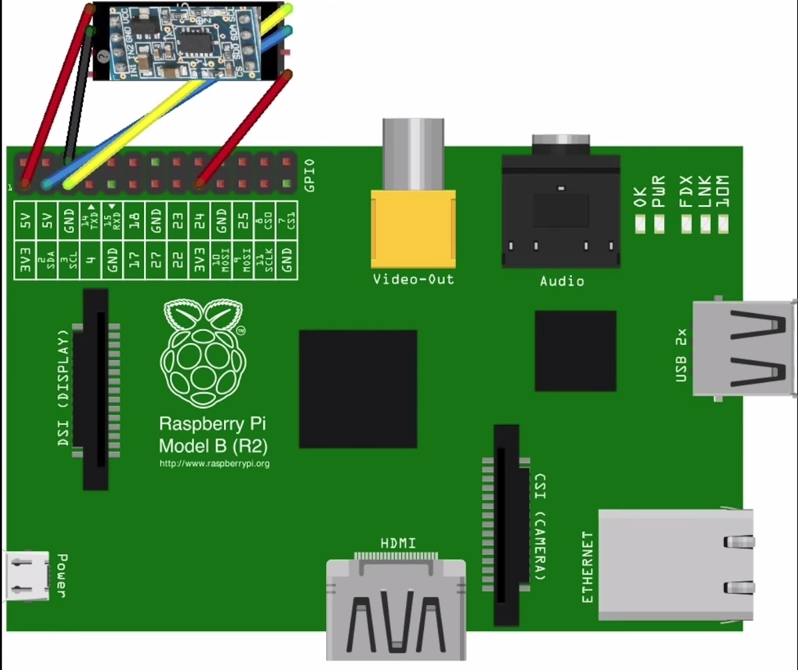
**Background**

This project will be developed on the Raspberry Pi 2 Model B, the second generation Raspberry Pi. It has 1GB RAM and a 900MHz quad-core ARM Cortex-A7 CPU.

Pedometers don’t actually count steps. They actually sense movement. In this project, MMA7455 accelerometer sensor with the Raspberry Pi via I2C is used to sense the movement and to classify which the activities in real time.

**Proposed Method**

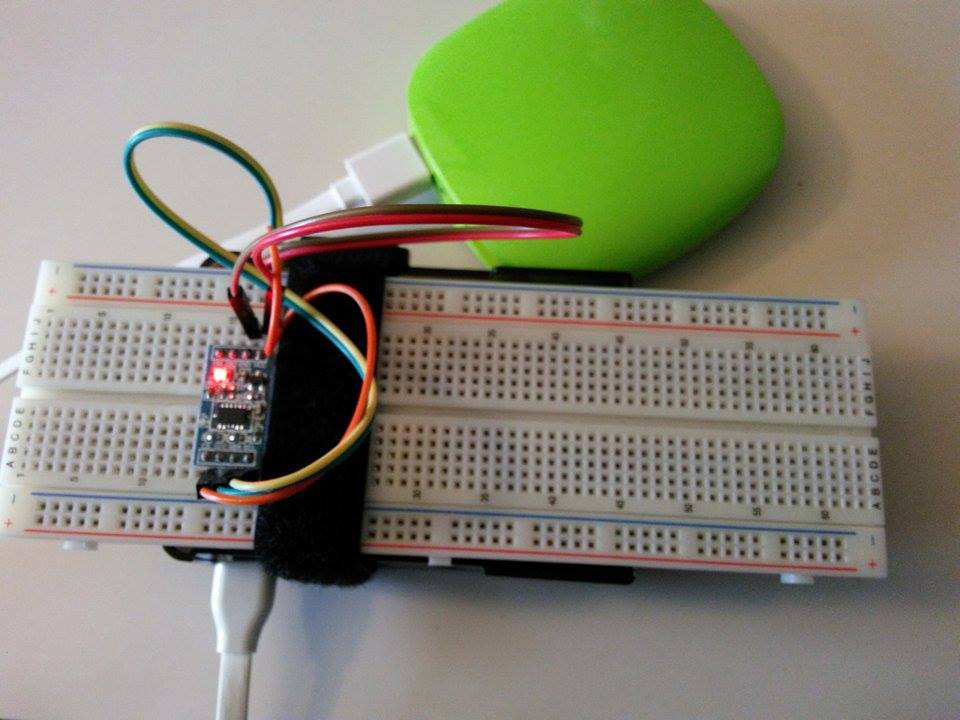
This project will be in user space. I use multiple threads to run in the system to make the system more efficient, running the program in parallel. The pins from accelerometer will be hooked up with the pins on the Raspberry pi board’s pins. As in the picture below:



**Experiments and Results**



The graph above is an example of the data I got from the sensor. Clearly, I can classify the steps by using x-axis of the peaks that below 50 to classify as steps. I also tried to use fitbit, to get a raw data, but the fitbit has a locked system that will only show how many steps a person walk.



This is how I set it up and then I attach to my left hip.

**Discussion**

As far as debugging, the hardest part for this project is the classify from the graph and to set up the port to connect with the sensor correctly. Since the sensor is not the consistence sensor, it’s harder to get a accurate walking step.

**Conclusion**

For the most part, the results are not what I expected. It’s very hard to figure a lot of parts. The sensor is not consistence as I expected. For further more experiment, I might need to try more number of trials and test with different sensors and with different method of how you attach the device to your body for counting steps.

**Appendices**

Code is in the final\_project.c

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <linux/i2c.h>

#include <linux/i2c-dev.h>

#include <unistd.h>

#include <sys/ioctl.h>

#include <pthread.h>

#include <sys/time.h>

/\* The 7-bit address \*/

#define MMA7455\_I2CADDR 0x1D

/\* The mode control register address \*/

#define MMA7455\_CTRLADDR 0x16

/\* The control value "0100 0101" \*/

#define MMA7455\_CTRLREG\_VAL 0x45

/\* The registers to read \*/

#define MMA7455\_XOUT8 0x6

#define MMA7455\_YOUT8 0x7

#define MMA7455\_ZOUT8 0x8

/\* The name of the file \*/

#define I2C\_FILE\_NAME "/dev/i2c-1"

/\* The readings are 8 bits and signed \*/

int8\_t x, y, z;

int i2c\_file;

static int set\_i2c\_register(int file,

unsigned char addr,

unsigned char reg,

unsigned char value) {

unsigned char outbuf[2];

struct i2c\_rdwr\_ioctl\_data packets;

struct i2c\_msg messages[1];

messages[0].addr = addr;

messages[0].flags = 0;

messages[0].len = sizeof(outbuf);

messages[0].buf = outbuf;

/\* The first byte indicates which register we'll write \*/

outbuf[0] = reg;

/\* The second byte indicates the value to write. Note that for many \*

\* devices, we can write multiple, sequential registers at once by \*

\* simply making outbuf bigger. \*/

outbuf[1] = value;

/\* Transfer the i2c packets to the kernel and verify it worked \*/

packets.msgs = messages;

packets.nmsgs = 1;

if(ioctl(file, I2C\_RDWR, &packets) < 0) {

/\* Error Indication \*/

perror("Unable to send data");

exit(1);

}

/\* Success Indication \*/

return 0;

}

static int get\_i2c\_register(int file,

unsigned char addr,

unsigned char reg,

unsigned char \*val) {

unsigned char inbuf, outbuf;

struct i2c\_rdwr\_ioctl\_data packets;

struct i2c\_msg messages[2];

/\* In order to read a register, we first do a "dummy write" by writing \*

\* 0 bytes to the register we want to read from. This is similar to \*

\* the packet in set\_i2c\_register, except it's 1 byte rather than 2. \*/

outbuf = reg;

messages[0].addr = addr;

messages[0].flags = 0;

messages[0].len = sizeof(outbuf);

messages[0].buf = &outbuf;

/\* The data will get returned in this structure \*/

messages[1].addr = addr;

messages[1].flags = I2C\_M\_RD;

messages[1].len = sizeof(inbuf);

messages[1].buf = &inbuf;

/\* Send the request to the kernel and get the result back \*/

packets.msgs = messages;

packets.nmsgs = 2;

if( ioctl(file, I2C\_RDWR, &packets) < 0 ) {

/\* Error Indication \*/

perror("Unable to send data");

exit(1);

}

/\* Success Indication \*/

\*val = inbuf;

return 0;

}

void \*counter\_step1(void \*parameter){

FILE \*outputarray = fopen("output\_xyz.csv","w+");

int step = 0;

int exit\_c = 0;

while (exit\_c < 100) {

/\* Read X, Y, and Z from the register \*/

if( get\_i2c\_register(i2c\_file, MMA7455\_I2CADDR, MMA7455\_XOUT8, &x) ||

get\_i2c\_register(i2c\_file, MMA7455\_I2CADDR, MMA7455\_YOUT8, &y) ||

get\_i2c\_register(i2c\_file, MMA7455\_I2CADDR, MMA7455\_ZOUT8, &z) ) {

/\* Error Indication \*/

printf("Unable to read register\n");

exit(1);

}

/\* Success Indication \*/

/\* Debug lifn e that prints out registers \*/

if (outputarray != NULL){

fprintf(outputarray,"%d,%d,%d\n", x, y, z);

}

printf("X = %d\tY = %d\tZ = %d\n", x, y, z);

if (abs(z) >10 || abs(x) < 20){

step++;

printf("step = %d\n",step);

}

usleep(750000);

exit\_c++;

}

}

int main(int argc, char \*\*argv) {

pthread\_t counter\_step;

/\* Open a connection to the I2C userspace control file \*/

if ((i2c\_file = open(I2C\_FILE\_NAME, O\_RDWR)) < 0) {

/\* Error Indication \*/

perror("Unable to open i2c control file");

exit(1);

}

/\* Success Indication \*/

printf("Initialized\n");

/\* Set control register \*/

if( set\_i2c\_register(i2c\_file, MMA7455\_I2CADDR, MMA7455\_CTRLADDR, 0x45) ) {

/\* Error Indication \*/

printf("Unable to set control register!\n");

return -1;

}

/\* Success Indication \*/

printf("Set control register successfully\n");

int ret = pthread\_create(&counter\_step, NULL, counter\_step1,NULL);

if (ret != 0) {

printf("Create thread failed! error: %d", ret);

}

pthread\_join(counter\_step, NULL);

/\* Cleanup and Exit \*/

close(i2c\_file);

return 0;

}