A DEVICE TO RECORD NATURAL DAILY WRIST MOTION

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

by
Surya Prakash Sharma
December 2014

Accepted by:
Dr. Adam W. Hoover, Chair
Dr. Ian D. Walker
Dr. Jason O. Hallstrom

Abstract

We introduce a new device to record and store wrist motion activity data. The motivation to create this device was the fact that this data can be used to detect periods of eating or the number of bites consumed. There is no similar device available in the market. This device uses new components that have been recently introduced to the market, and newer techniques that can be used for low quantity production. The production cost for this device was \$52, similar to other fitness trackers on the market. The device was capable of recording wrist motion activity for 24 hours and was similar in weight to a wrist watch.

Acknowledgments

Being someone who wants to try everything out, I would not be where I am in life without the exceptional people around to guide me. These include professors, family members and friends.

I would like to thank Dr. Chelpa Lingam for encouraging me to continue studying and persevere in the hardest of situations. Without his encouragement I would probably not make it to graduate school.

The entire Clemson community has been a tremendous help. Dr. Adam Hoover understood my personality and tolerated my mistakes, for which I am grateful. I'd like to thank my committee members, Dr. Ian Walker and Dr. Jason Hallstrom for their help and guidance in creating our proposed device.

The staff members working for ECE have helped me over the last two years: John Hicks, Robert Teague, David Moline, Elizabeth Gibisch and Trish Nigro, you have been a great team. Linda Jennings would always help bring a smile to my face.

My colleagues and friends, Poornapragna Rao and Ranajeet Anand, you made the two years spent at Clemson well worth it. It was a memorable experience working here with you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis considers the problem of tracking human wrist movements during free living. The goal is to build a device that can track these movements continuously all day, storing the data so that it can be processed later. While many wrist-worn devices with tracking capabilities are manufactured for the commercial and research markets, none has specifically addressed the need of recording translational and rotational wrist motion for an extended period of time. Many devices do not include the necessary sensors to track rotational motion, or include unneeded features that make the device larger than necessary and require frequent recharging. Device cost is also a factor given the desire to record data for many subjects over extended periods of time. The work in this thesis was motivated by the goal of designing and prototyping a device that limits functionality to the goal of tracking and recording wrist motion all day while minimizing device size and cost.

The technical challenges of the problem are motivated by the size and power usage of the required components. The sensors necessary for tracking wrist motion are a 3-axis accelerometer and a 3-axis gyroscope. At the time of this writing, both types of sensors are available in small microelectricalmechanical systems (MEMS) packages with chip dimensions only a few millimeters. Thus, they can be comfortably worn in a wrist-mounted package. However, their power consumption varies by an order of magnitude. MEMS accelerometers typically draw a current in the range of 300-500 μA and MEMS gyroscopes typically draw 3-10 mA of current. To be worn on a wrist, these sensors must be powered by a battery. Common small batteries used in wrist based devices have a capacity of 20-200 mAh. These capacities allow for the continuous operation of a MEMS accelerometer for up to a week on a single charge. However, a MEMS gyroscope could typically

operate for only 5-15 hours on a single charge. The battery size is particularly important because it is usually the largest part in the device. Since the goal is to mount the device on the wrist, we are motivated to keep the battery size as small as possible. These technical challenges are confounded if other unnecessary parts are included in the device.

1.1 Motivation

The motivating vision for this work is a wrist mounted device that trackings wrist motion to detect when a person is eating, counting the number of bites consumed in each meal. In previous research our group has demonstrated algorithms where periods of eating can be detected with 81% accuracy [1], and the number of bites in these meals can be counted with 86% accuracy [2]. To further this research, a device is needed that will facilitate the recording of wrist motion data for a large number of subjects, preferably with each subject recording for several days or even weeks. This data could then be used to improve the detection and measurement algorithms.

For example, our previous work has suggested that the gyroscopes may not need to be powered continuously to detect periods of eating [1]. Instead, accelerometers could be powered continuously with gyroscopes only powered during periods of suspected eating to determine a final classification [1]. To test this idea, a large data set of continuous full wrist motion needs to be recorded. Algorithms that processed the gyroscope data at intermittent intervals could be compared to algorithms that processed all the gyroscope data.

As another example, algorithms could be custom calibrated to an individual, perhaps improving performance over algorithms that are only tuned to the group level. This is only feasible if sufficient data is recorded per person to enable custom tuning of thresholds or other algorithm parameters. To facilitate collecting this amount of data, a device that is comfortable to wear for many days is necessary in order to encourage enough subjects to collect the necessary data.

1.2 Existing devices

Table 1.1 shows a list of some devices on the market that can be used to track wrist motion and could potentially be used for the envisioned work. There are four main categories. The first category is fitness trackers. It includes devices like the Fitbit [3] and Jawbone [4] series of sensors.

Name of Device	Programmability	Battery (hrs)	Weight	Cost	Type	Body Position	Dimensions (mm)	Features
Fitbit Zip	Low	6 months	8.0	$49[4]	Fitness tracker	Waist / Torso / Wrist	36.0 x 28.0 x 10.0	Accelerometer
Fitbit Flex	Low	120	N/A	$99[4]	Fitness tracker	Wrist	140.0 x 161.0 x 13.9	Accelerometer
Jawbone Up Move	Low	168	6.8	$49[4]	Fitness tracker	Torso / Waist	27.6 x 27.6 x 27.6	Accelerometer
Jawbone Up 3	Low	168	29.0	N/A	Fitness tracker	Wrist	220.0 x 12.2 x 3.0	Accelerometer
Fitbit One	Low	240	8.0	$99[4]	Fitness tracker	Torso	48.0 x 20.0 x 10.0	Accelerometer
Nike FuelBand	Low	96	30.0	$149[4]	Fitness tracker	Wrist	147.0 (circumference)	Accelerometer
iPhone	High	40[1]	148.9	$239[4]	Mobile phone	Hand held	115.2 x 58.6 x 9.3	Gyroscope, Accelerometer Phone calls, LCD screen Media player
MetaWatch	High	80	81.0	$249[4]	Smartwatch	Wrist	51.0 x 38.0 x 13.0	Accelerometer Display Bluetooth connectivity
Samsung Gear 2	Medium	48	68.0	$275[4]	Smartwatch	Wrist	58.4 x 36.9 x 10.0	Gyroscope, Accelerometer Display, Phone calls Bluetooth connectivity
Thalmic Labs Myo	Medium	48	93.0[3]	$149	Research	Upper arm	N/A	Gyroscope, Accelerometer EMG Sensors
SHIMMER	High	24[2]	23.6	$249	Research	Wrist	50.0 x 25.0 x 12.5	Gyroscope, Accelerometer, Bluetooth module

Table 1.1: Comparison of select wrist based motion tracking devices in the market.

Notes:

1: Battery Life for iPhone 4 while listening to music and display is turned off [5].

2: Information provided by Shimmer Customer support over email.

3: No specification sheet is available, but the weight was available on a product discussion page [6].

4: Information from an Amazon product page [7].

Figure 1.1: The Fitbit flex activity tracker.

These devices allow for exercise and sleep monitoring. The Fibit activity monitor can be seen in figure 1.1 [8]. Its data is sent to a computer or a mobile phone to be viewed graphically by the user. Many of these devices can be worn the wrist, are small, relatively low-cost, and can be powered for a week or longer on a single battery charge. However, they do not include gyroscopes. Activity monitoring is accomplished solely through the use of accelerometers. In addition, accelerometer data is not stored continuously. Instead, 10-60 second windows of data are reduced to single measurements that are related to step count, physical activity, or sleep activity, by on board processing of the raw sensor data. Thus, none of these devices are designed to sense or record continuous wrist motion data.

Another relevant class of devices is smart phones. These devices commonly include gyroscopes and can be custom programmed to sense and record data continuously. They contain larger batteries than fitness monitors, and so can power gyroscopes for more than 10 hours. They also contain large memories that can store the raw sensor data for an extended period of time. In our group's preliminary work, an Apple iPhone was used to record data from 44 subjects for a single day each [1]. However, smart phones are large and not intended to be worn on the wrist. The comfort level of participants was lower than desired. Smart phones are also relatively expensive.

A third class of devices that has recently emerged is the smart watch. Examples include the Pebble and Samsung Gear. These devices include many parts common to smart phones and are

packaged in a wrist-worn assembly. The goal is to enable viewing of text messages and other data commonly viewed on a smart phone on a smaller screen worn on the wrist. A smart watch typically uses low power Bluetooth to communicate with a paired smart phone. Some of these devices include gyroscopes and most can be custom programmed. However, they are relatively larger than fitness devices due to their inclusion of a large display, wireless connectivity, and other features typically associated with smart phones. They are also relatively expensive.

A fourth class of relevant devices are those produced for research use. Examples include devices produced by XSens and Shimmer labs. These devices provide generic tracking capabilities. They are generally intended to be positioned or worn anywhere, so few specialize in wrist-mounted packaging. While they can include gyroscopes and facilitate custom programming, they also serve wide markets and thus tend to include parts and capabilities not necessary for the goal of this work. This tends to increase size and expense.

1.3   Novelty

The purpose of this work is to design and prototype a device that targets the goal of tracking and recording wrist motion continuously for a day. The challenges are to meet or surpass the existing devices listed in Table 1.1 in terms of the collective listed criteria. Towards that goal, this thesis describes an investigation of the parts currently available for use in building such a device, including sensors, memory, batteries, connectors, and materials and methods used for embedded device construction. After selecting parts and an overall design, a functioning prototype device was consructed. Its operating characteristics are compared to the devices listed in table 3.2 at the end of this thesis.

Chapter 2

A new device to record wrist movement in free living humans

Our planned device design for the all day wrist motion activity tracker revolves around a system of sensors that are polled at 15 Hz by a microcontroller. The data is logged throughout the day. Once the data for a day has been logged, the device is connected to a computer and data from the memory chip is dumped to a computer where it can be analyzed. Throughout the day, we need some kind of display to inform the user of the status of the device.

To create the prototype device, different prototype breakout boards were used (section 2.5). These boards allow rapid prototyping by hooking up components using wires. This reduces any time spent to solder an integrated circuit to a PCB or to understand its typical application circuit components. Next, we consider the different components used to create our wrist motion activity tracker. We need a power source in the form of a battery (section 2.12), sensors to measure acceleration (section 2.2) and angular velocity (section 2.3), a memory chip (section 2.7) to store data from the sensors, and a microcontroller (section 2.6) to control this system and act as a communication channel between the different components. To connect this device to a computer, we would need a translator. We use a USB to UART bridge (section 2.10) to accomplish this.

With all these components available, we have to connect them together correctly, which is explained in section 2.17. Once the design has been tested, it is laid out as a PCB design using EAGLE PCB, and then fabricated. This PCB would then need the IC's soldered to it, and we
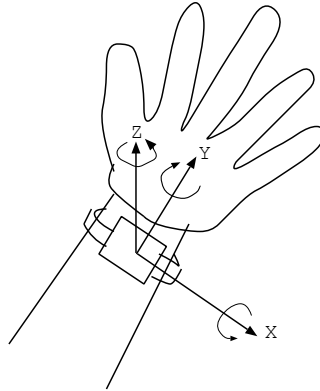
Figure 2.1: A wrist watch shaped device showing the different axis being recorded.

discuss the two methods used to do so (section 2.17).

We recap that we have two strong requirements for our device, and all hardware decisions must consider the device size and the power consumption. Throughout this document, we provide photographs of the different components used in creating our wrist activity monitor. These parts are very small in size, and to emphasize this, we show a coin, the standard US quarter next to the part being discussed for comparison. This will hopefully give the reader a better idea of the scale of devices we are working with. Remembering that device size and power consumption are very important, we look at the different hardware in our device in the next section.

2.1 Sensors

Since we want to log acceleration and rotation data, we consider the different sensors available in the market and their sizes. At the time of writing this thesis, the leader in this field was a category of sensors called Microelectromechanicalsystem sensors. MEMS sensors are small devices ranging from about 4 mm^2 to 25 mm^2. Our device design uses an MEMS gyroscope and accelerometer in a single electronic chip to sense movements. This chip may also be called the inertial measurement unit (IMU). The orientation and axes for this chip when it is worn on the wrist as a part of our final device is shown in figure 2.1.
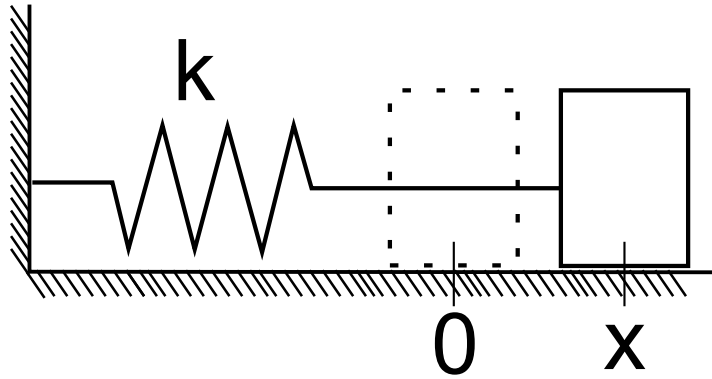
Figure 2.2: Example of a spring with spring constant k, displaced from its equilibrium 0 by X

2.2   Accelerometers

Accelerometers are devices that sense linear acceleration. In the case of our device we have to measure the linear acceleration in all three Cartesian coordinate axes: X, Y, and Z. This will allow us to track the movement of the wrist in free living humans as their hand moves around in free space. Figure 2.1 shows how the axis of the sensor are oriented for our device when worn on the wrist. To understand how an accelerometer works, we imagine a system devised of a spring connected to a fixed surface at one end, and an object with a mass M at the other end. This is shown in figure 2.2. The equilibrium point 0 and extended distance X is also shown. Since springs are governed by Hooke's Law, we can use this law to explain the behavior of our system. Hooke's law states that to extend or compress a spring by distance X, the force required is proportional to X, or mathematically as given in equation 2.1:

$$F = -kX \tag{2.1}$$

where F is the force required, X is the distance the spring is extended or compressed from its equilibrium position, and k is the spring constant for a particular spring.

We also have Newton's second law of motion which states that the force applied on a body with constant mass produces a proportional acceleration. Mathematically, Newton's second law of motion

can be expressed as given in equation 2.2:

$$F = ma \tag{2.2}$$

where F is the force, m is the mass of the body, and a is the acceleration produced. Combining the two equations, we can solve for acceleration using the result in equation 2.3.

$$a = \frac{-kX}{m} \tag{2.3}$$

However, consider the fact that the spring constant k and mass m are constant in the equation above, which means that acceleration a is inversely proportional to the displacement X. Because of this fact, new MEMS accelerometers are very simple systems consisting of small mechanical parts that measure the displacement and output the value of the acceleration measured based on this displacement. Something that we must note is that since the sensor is measuring force, that gravity is always acting on the device, and due to this, a reading of 1g will always be read on the Z axis in the "earth" system. If there is no motion, we can empirically know which direction gravitational force is acting in by looking at the sensed data, and can subtract this from later readings obtained from the accelerometer.

2.3 Gyroscopes

Gyroscopes are devices used to measure angular movement. In our case, we use MEMS gyroscopes to measure angular velocity in three axes, X, Y and Z (as shown in figure 2.1). How a gyroscope works is explained in a Sparkfun tutorial [9]. We will review the same here.

An MEMS gyroscope contains a small mass that oscillates inside it as shown in figure 2.3. When the gyroscope rotates, the oscillating mass is displaced between two oscillations. This change in location is converted into a very low current, that is then amplified (and converted to a voltage) by a microcontroller inside the gyroscope integrated chip.
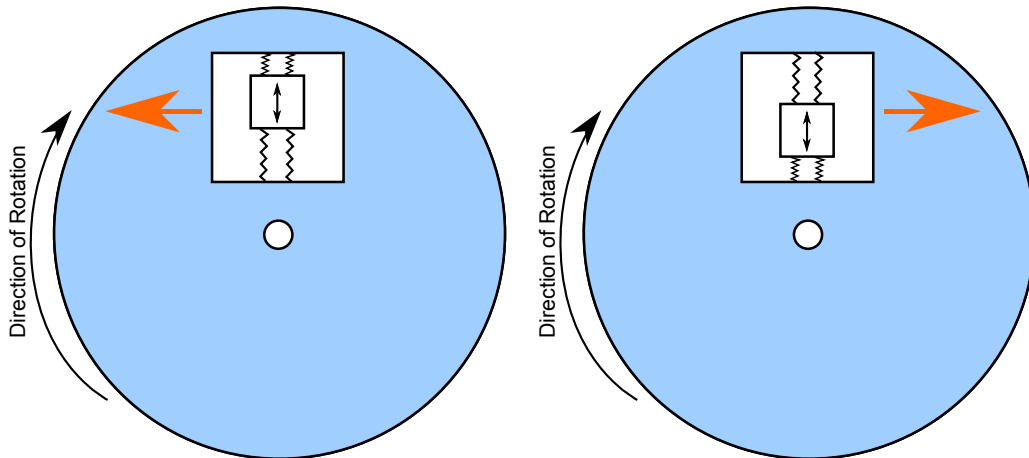
Figure 2.3: Internal working of a gyroscope

2.4 Accelerometer and gyroscope device selection

There are a variety of MEMS accelerometers and gyroscopes available in the market with different specifications. At the time of this writing, the MEMS accelerometer and gyroscope manufacturing industry is dominated by two companies: STMicroelectronics and InvenSense. Our purpose was to pick the most suitable device for logging wrist movement data. We will first look at the different parameter's that we have for MEMS devices. These parameters are referenced from Analog Devices' website[1] :

Output

The output of the sensor can be analog or digital. For an analog device, the output is a voltage corresponding to the detected measurement. This means that an acceleration between -A_{max} to +A_{max} would be output as a voltage between V_{dd} and V_{cc}. Each output would have its own pin to output a signal. A digital device on the other hand, would use some kind of communication protocol. The device usually has registers or a memory stack that can be read using a protocol like I^{2}C (also known as TWI) or SPI. Using this protocol, a master device (microcontroller) can poll the slave (sensor) to read its memory.

Output data rate

This defines the rate at which data is sampled and then output by the device. In analog accelerometers, this represents how often an accelerometer updates its output voltage in response

---

[1] Analog Devices - Accelerometer Specifications - Quick Definitions. Last accessed October 2014
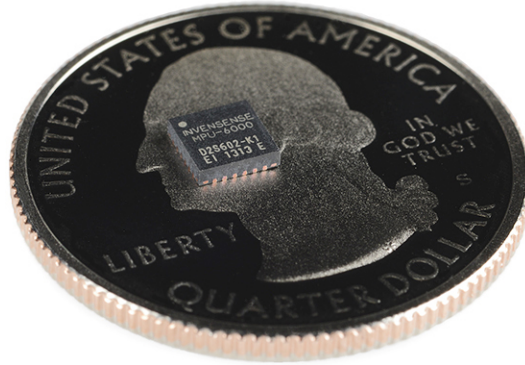
Figure 2.4: The MPU-6000 IMU, placed on an American Quarter to emphasize size.

to the measured acceleration. In digital accelerometers, this represents how often the register storing the value of the acceleration is updated. Our required data rate is 15Hz for sufficiently accurate monitoring of wrist movements.

Package type and size

Since the device is mounted on the wrist, space and volume is at a premium. Most MEMS devices are catered to the consumer electronics market, thus used in small devices like mobile phones and smart watches. Due to this, there is constant innovation in the market with regards to the size of the integrated chip's footprint. Packages like LGA (land grid array) and QFN (quad flat no - lead) are used. These packages usually lack pins and are surface mounted to reduce the footprint on a PCB. An example of the QFN package is the IMU that we used in our device, the MPU 6000 produced by InvenSense Inc. An image from Sparkfun.com [10] (see figure 2.4) shows the size of the chip compared to a quarter.

Measurement range

This defines the range of measurement supported by an IMU's output specifications. For an accelerometer, the range is usually defined in g's, where $1g = 9.8$ m/s^2. This is the greatest amount of acceleration that can be measured and accurately represent as an output. A larger acceleration maybe incorrect or capped at the maximum value. For example, the MPU-6000

11

has four modes with measurement ranges from $\pm$2g to $\pm$16g. The maximum measurement range value is not the breaking point for the device. A higher acceleration of 17g may not break the accelerometer, but will not be reported correctly. The minimum value of acceleration that might harm the sensor is defined as the absolute maximum acceleration.

Human motions are typically contained within 2g of acceleration. With this fact, our options for accelerometers are fairly large considering that most MEMS accelerometers aimed at the consumer electronics market will have a minimum measurement range of 2g's or greater.

Current consumption

The power drained by the device is an important factor for this design. MEMS accelerometers are fairly low power devices. The LIS344ALH accelerometer used by Drennan [11] typically consumed 650 μA of current when operating and a maximum of 5 μA when in sleep mode. Gyroscopes on the other hand tend to consume a very high amount of current. Drennan used two gyroscopes [11], each consuming 6.8 mA. This means that on a standard coin cell battery that has a size of 90 mAh, the gyroscopes would run for about 7 hours, a very short lifetime.

Number of axes

As MEMS technology has improved, the sensors based on these technologies have improved. Early accelerometers would sense acceleration across a single axis. With current technology, most accelerometers and gyroscopes can measure motion across all three axes. Inertial measurement units are available that pack not only an accelerometer and a gyroscope, but also a magnetometer for more accurate motion tracking. This multiple sensor package greatly reduces the current consumption against a scenario where multiple devices are being used, each consuming its own current. We will not be using a magnetometer for our work as we require raw signals and not inertial movement data. Due to this we can ignore parts with magnetometers.

Name	Description	Output	Package	Current Consumption	Noise	Full Scale Range
STMicro LPR410AL	2 Axis Gyro (pitch, roll)	Analog	LGA 4X5X1.1 mm	6.8 mA	Gyro: 0.014 °/s/Hz	100 °/sec
STMicro LPY410AL	2 Axis Gyro (pitch, yaw)	Analog	LGA 4X5X1.1 mm	6.8 mA	Gyro: 0.014 °/s/Hz	100 °/sec
STMicro LPR344ALH	3 Axis Accelerometer (X,Y,Z)	Analog	LGA 4X4X1.5 mm	680 μA	50 μg/Hz	2/6 g
STMicro LGD20H	3 Axis Gyro (yaw,pitch,roll)	Digital	LGA 3X3X1.1 mm	6.2 mA	Gyro: 0.03 °/s/Hz	245/500/2000 °/sec
STMicro L3G462A	3 Axis Gyro (yaw,pitch,roll)	Analog	LGA 4X4X1.1 mm	6.9 mA	Gyro: 0.17 °/s/Hz	625 °/sec
STMicro LSM330	6 Axis (X,Y,Z,y,p,r)	Digital	TFLGA 3.5X3X1 mm	6.1 mA	Gyro: 0.03 °/s/Hz	[2 g/4 g/6 g/8 g/16 g] [250/500/2000 °/sec]
Invensense MPU-61NX	6 Axis (X,Y,Z,y,p,r)	Digital	QFN 4X4X0.9 mm	3.8 mA	Gyro: 0.005 °/s/Hz	[2g, 4g, 8g,16g] [56/113/225/450 °/sec]
Invensense MPU-6XX0	6 Axis (X,Y,Z,y,p,r)	Digital	QFN 4X4X0.9 mm	3.8 mA	Gyro: 0.005 °/s/Hz	[2g, 4g,8g,16g] [250/500/1000/2000 °/sec]
Invensense MPU-91NX	9 Axis (6 Axis + magnetometer)	Digital	QFN 4x4x1 mm	3.8 mA	Gyro: 0.005 °/s/Hz	[2g, 4g,8g,16g] [250/500/1000/2000 °/sec]

Table 2.1: Comparison of select motion sensors available in the market.

We compared a few of the options available in the market in September 2013 against the parameters discussed previously. Table 2.1 shows the different product names versus their important parameters. The first three products were used by work done by Drennan [11]. Based on these different parameters and comparing with previous work, we have selected the InvenSense MPU-6000. The device contains both, a 3-axis accelerometer and a 3-axis gyroscope. It features three 16-bit analog-to-digital converters each for both these devices to digitize the sensor outputs. This data is stored in internal registers that can be accessed using I^2C at 100 kHz or using SPI at 1 Mhz. An on-chip buffer allows storing bursts of readings, which can be read in bursts by a master device. The device operates at a voltage of 2.375 V to 3.46 V. An operating current of 3.8 mA means that the device can operate for roughly 24 hours on a standard coin cell battery. This makes the MPU 6000 very suitable for our requirements.

2.5 Breakouts

Newer integrated chips are packaged very densely, and have very small pins to connect to a PCB. With surface mount technology, it is possible to solder these devices to a PCB, however it is very difficult for hobbyists to do so because of the lack of expensive equipment. Electronic part retailers provide products in the form of breakout boards, which means that they provide a PCB with the integrated chip soldered. These PCBs have pins to connect to that can be used conveniently. In effect, a breakout board converts pin - less packages like BGA (ball grid array) and LGA (land grid array) to DIP (dual inline package).

These breakout boards are easier to prototype with, and allow modular creation of a complete circuit. Although we have selected the MPU 6000, we still have to work it into our circuit and ensure that it will work in our device. To do so, conventional methods would require us to first solder this IC to a PCB containing the other components, and then program a microcontroller to read from the sensor. However, PCB fabrication and soldering can be expensive, and are not a viable option during the prototype phase of product design. PCB manufacture can also take a considerable amount of time (up to a month), and is not a feasible strategy to work with. To prototype and test the sensor without fabricating a complete PCB, we decided to use a breakout board containing the sensor. Figure 2.5 shows an example of a breakout board. The image was sourced from Sparkfun [12].

Figure 2.5: A breakout board containing the MPU 6050 chip. Quarter shown for comparison.

2.6   Microcontroller

The sensors we discussed in the previous section cannot function on their own. They are primarily slave devices that can be requested for data. To get this data, we require a master device, which we have in the form of a microcontroller. Our microcontroller would need to support the communication protocol used by our sensors and memory chip. It would also need to have a sufficient memory to be able to temporarily buffer sensor data before it is flushed to the external memory chip.

Several manufacturers provide microcontrollers that we can use, however, our priority of low current is what will separate the different microcontrollers, and help us pick the right one for our device. We considered the different wrist based devices on the market and realized that the MSP430 family of microcontrollers greatly dominates this market segment [13, 14, 15]. The MSP430 Brochure [16] lists roughly three hundred parts that can be picked from, categorized as:

- Low power FRAM series

- Value line series with limited components

- F1xx Family with increased integration and performance

- F4xx Family with an integrated LCD driver

- F5xx Family with high frequency operation (up to 25 Mhz)

15
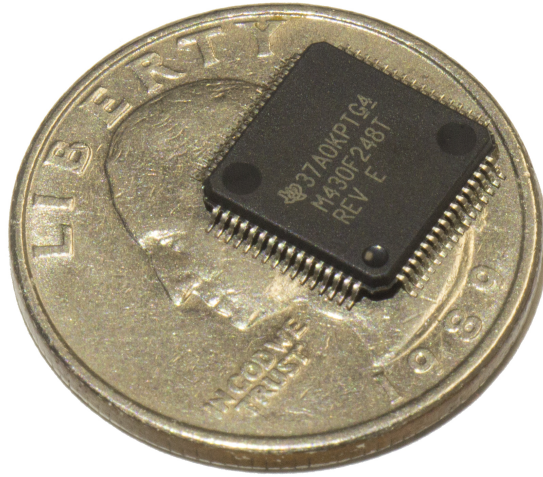
Figure 2.6: The MSP430F248 integrated chip, with a quarter to show scale.

- CC430 series (microcontroller with integrated RF chip)

When selecting our chip, we needed one that would allow future expansion with regards to the capabilities of the wearable monitor. The SHIMMER device uses the MSP430F1161 [17], which meets most of the requirements for our device. However, almost the entire MSP430 family is pretty close to what we might need. Our microcontroller would need SPI or I^{2}C support so that it can communicate with the memory chip and our sensor. The MSP430F1161 lacks any module that can communicate over the SPI protocol, which would mean we would have to implement a software based module, increasing the complexity of the device. The microcontroller would also need to have a buffer large enough to store a sizable amount of data before it would need to be flushed to the memory chip. We found parts with SPI support and a minimum SRAM of 2048 bytes in the MSP430F24X series. This sub-family of devices had the following characteristics [18]:

- Low supply-voltage range, 1.8 V to 3.6 V

- Active mode current consumption: 270 A at 1 MHz, 2.2 V

- Standby mode (VLO) current consumption: 0.3 A

- Four universal serial communication interfaces supporting SPI and I^{2}C.

- Two 16-bit timers with multiple compare registers.

16

We selected the MSP430F248 as our microcontroller. This chip comes with a 48 KB + 256 B Flash memory and 4 KB of RAM. Assuming our program size would not be more than 1 KB, this would allow us to buffer about 3KB of data from the sensors before flushing it to the external memory, allowing the memory to remain in sleep mode for a large amount of time.

2.7 Memory

The information sensed by the sensors has to be logged, and then processed by a computer. We could either store this information or transmit it wirelessly as soon as it is gathered. For a system designed to read sensors at 15 Hz, and logging data for 24 hours, we estimate the amount of data as shown in Equation 2.4

$$\text{data} = 24 \text{ hours} \times \frac{60 \text{ minutes}}{\text{hour}} \times \frac{60 \text{ seconds}}{\text{minute}} \times \frac{15 \text{ polls}}{\text{second}} \times \frac{6 \text{ sensors}}{\text{poll}} \times \frac{1 \text{ byte}}{\text{sensor poll}} \tag{2.4}$$

$$\text{data} = 7,776,000 \text{ bytes} \approx 7.5 \text{ MB} \tag{2.5}$$

We do not consider wireless transmission for our device. The device would have to transmit data to another device which it would be paired to. The paired device would need to remain in wireless connectivity range to store this data. This adds another device to our solution, increasing the power consumption, so we do not consider it.

To store the 7.5 megabytes of data, we need some kind of memory. We see in section 2.6 that our microcontroller has a flash memory of 48 KB. This would mean we need a memory to store the data during the interval that the user is wearing the device. This data can then be transferred to a computer from the memory once we have a connection between the two.

We compared the available options in the market. Most memory chip's are sold in powers of 2, so we would have to look at devices close to 8 megabytes or 64 megabits in design. A search on Digikey [19] while working on this thesis showed that there were very limited options in the flash memory market that would support such a high density of memory. The options were further limited by the fact that we required a small footprint that we could add to our wrist mountable device. Options typically had parallel input / output which meant they had 28 pins or more, which
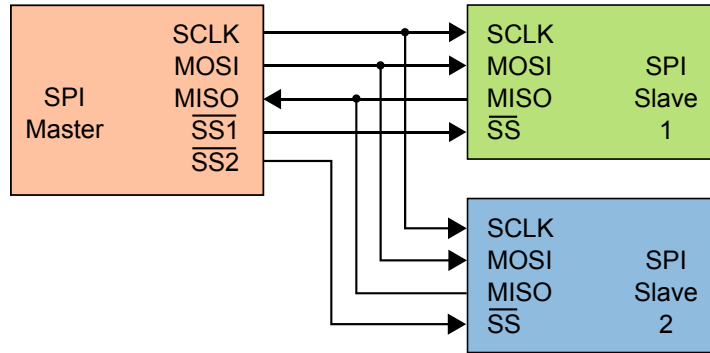
Figure 2.7: An SPI system with two slaves on the same bus.

required a larger footprint.

Atmel's memory division released AT45DB642D, a 64-megabit 2.7 V dual interface memory chip. This chip would support the amount of memory we required, and ran on a single 2.7 V - 3.6 V supply, similar to the operating voltage for common microcontrollers. This chip also supported SPI, the same protocol that was being used by the sensors in our circuit. SPI allows us to use the same bus for data communication and clock, while using one extra line for each device's slave select pin. Figure 2.7, courtesy of a Wikipedia user [20], shows how two slave devices can connect to an SPI master. This means that we could reduce the number of I/O pins used in our microcontroller, and also reduce the number of tracks created on our PCB for data communication.

The chip was made available in an unconventional CASON package, which is a surface mount package. This means we would have to solder the chip before it could communicate with a microcontroller. Since it was recently released to the market, we were not able to find a breakout board on the market for this chip. We would have to create our own breakout board for testing. To avoid the delay incurred because of PCB fabrication (which can take up to two weeks), we manually wired the chip for breakout. After flipping it upside down, so that its pins were exposed, we soldered strands of a multi strand wire to each of these pads. These strands were then soldered through a dot matrix PCB, allowing us to use the new unit as a PTH (pin through hole) device. This required careful handling of the chip with a dexterous hand, and was done under a microscope. Figure 2.8 shows the result of two attempts to do this.
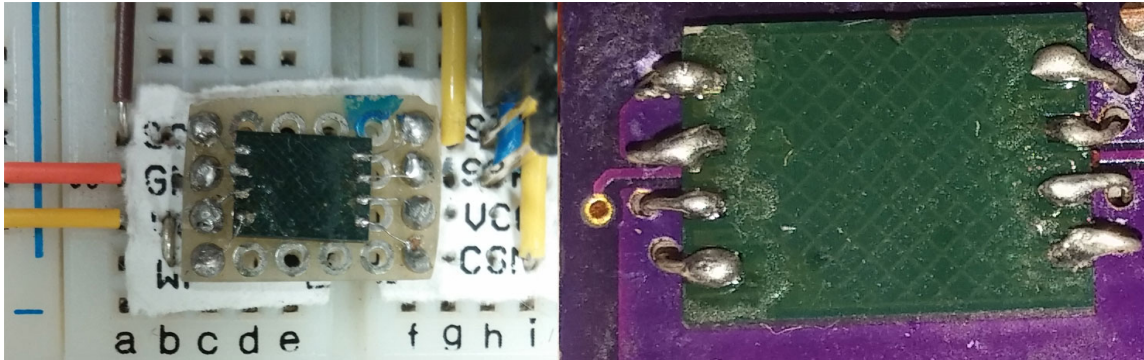
Figure 2.8: Two examples of flipping over a CASON chip and soldering wires to it's pads.

2.8 MSP 430 target board

Like our sensors, the MSP430F248 also comes in a surface mount package. The footprint is shown in Figure 2.6 and is known as the QFP[2] design. This package does have pins but they are not long enough to pass through a PCB. Instead they have to be mounted on the PCB. It is very hard to design a complete system without bugs; if we would design a PCB while writing microcontroller code at the same time, it would not be easy to debug an error because we would not know if the error is in the hardware design or the software code. Also, before we can use the microcontroller, any required external hardware needs to be connected. This includes capacitors and resistors, and a crystal oscillator to maintain accurate clocks. These parts would have to be soldered to the PCB with our microcontroller. Texas Instruments understands that this is not a fast process for someone who wants to dive into programming the microcontroller, so they provide a 64-Pin target board for its microcontroller. This target board has a ZIF[3] socket that accepts the 64-Pin QFP chip, and breaks out its pins. This means other devices can be connected to the board with ease allowing us to program the microcontroller without having soldered it.

Figure 2.9 shows us what the MSP430 target board [21] looks like, with the socket in the center. We can see a connector in the top-right of the figure, which is labeled JTAG. The socket is surrounded by holes on all sides which connect to the microcontrollers pins allowing for easy connections. The target board also allows for a crystal to be soldered next to the microcontroller if the programmer wants to use the low frequency clock based on an external crystal. We use this crystal for and accurate clock when polling our sensors and also when communicating with a host

---

[2]Quad Flat Package
[3]Zero Insertion Force
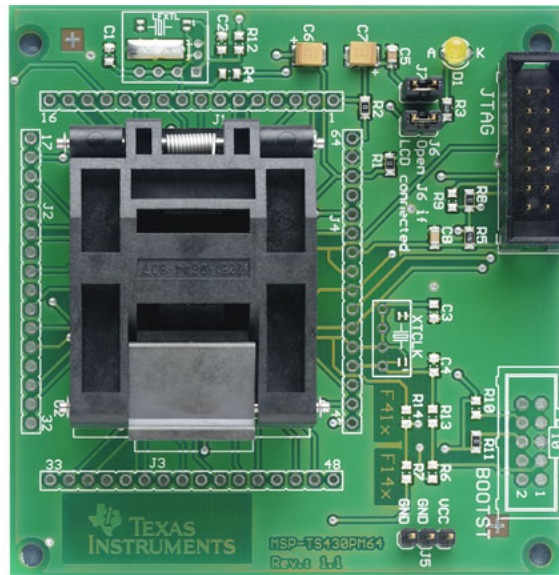
Figure 2.9: The MSP430 64-Pin target board.

computer as we will see in section 2.10.

2.9 MSP430 flash emulation tool

We need a way to program the MSP430 with with our code. The MSP430 series of micro-controllers are programmed by using JTAG using 4 wires. Modern computers usually lack a serial or parallel port for communication so Texas Instruments provides programmers with the MSP430 flash emulation tool. This tool (seen in Figure 2.10) connects to a desktop computer through USB, and the other end connects to a standardized JTAG connector as seen in figure 2.9 [22].

The tool supports powering the system it is programming so we were able to program our microcontroller without an external power source. Once we had our battery selected, a jumper on the MSP430 Target Board allowed us to disable the power supply of the flash emulation tool. The FET can be used by various software, paid or free. We used IAR's Code Composer Studio to program and flash the microcontroller. Through the MSP430FET, IAR supports not only programming and flashing the microcontroller, but also debugging the code while it is running. This feature enabled us to fix multiple bugs in a short amount of time, something that would be hard to do without the

Figure 2.10: The MSP430 flash emulation tool.

debugging capabilities. The software also allowed us to inspect different sections of the memory, allowing us to see live how the code is behaving.

2.10   USB to UART bridge

With the above hardware in place, we have a system which reads from sensors and stores this data in the memory chip. Once data for an entire day is logged, we would like to transfer the contents of this memory to a computer where the sensor data can be analyzed for different patterns and behaviors. Broadly speaking, there are two methods to transfer data, wireless and wired. Both these methods required parts that consume a current of about 18 mA when actively in use. However, in a wired connection, this power can be sourced from the host. Wireless transfer of data would consume power from the battery (unless a wire is connected to power the system), and assuming a speed of 115 Kbps, this would take about 30 minutes to transfer. With an active current draw of 18 mA for most wireless technologies, this would require a huge amount of power, something that the battery cannot sustain.

We solve both these problems by using a wired connection to transfer data. As mentioned in section 2.9, modern computer systems only have the USB port as a viable way to communicate. Although some MSP430 microcontrollers have native USB support, those microcontrollers have an increased current consumption and are not viable to our purpose. Our selected microcontroller, the MSP430F248, does not have a native USB module, but does have other communication modules. The MSP430F248 has a UART, and a combined I^2C + SPI module that can be used for communication. Since we will be using SPI for communicating with the sensors and the memory chip, using UART
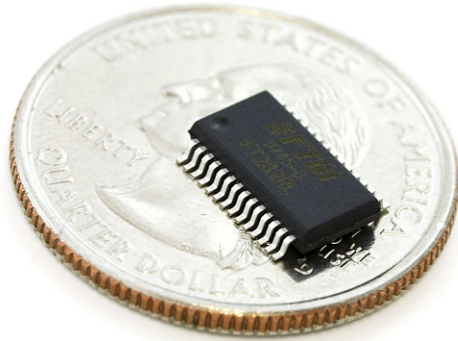
Figure 2.11: The FT232RL USB to UART bridge.

is a good way to isolate modules being used for different functions.

We looked for devices that would help convert communication in one protocol to another. Since SPI, I^{2}C and UART are all serial communication protocols, in theory, any device that can communicate in one of these protocols should be able to communicate in the other. However, a search on different devices in the market showed that USB to UART bridges are an easy way for microcontrollers to communicate with a computer. The device is detected as a virtual serial communication port to the operating system, and can be used as a regular serial port. Searching for USB to UART devices shows two primary contenders in the market segment, Silicon Labs and Future Technology Devices International (also known as FTDI). Both offer devices that have similar features and specifications.

We need a device that would communicate easily with our microcontroller, the MSP430F248 and also be easy to prototype with. All of Silicon Labs' offered parts are only available in the QFN package, which means it would be harder to prototype using them. For this reason, we select FTDI's chip, the FT232RL (shown in figure 2.11) [23]. FT232RL has the following specifications [24]:

- Single chip USB to asynchronous serial data transfer interface.

- Data transfer rates from 300 baud to 3 Mbaud (RS422, RS485, RS232) at TTL levels.

Figure 2.12: Ergo Minitec cases manufactured by OKW Enclosures.

- UART interface support for 7 or 8 data bits, 1 or 2 stop bits and odd / even / mark / space / no parity.

- Integrated +3.3V level converter for USB I/O.

- Available in compact lead-free 28 pin SSOP and QFN-32 packages.

The FT232 was also obtained in a breakout board for quick prototyping. USB 2.0 had a default maximum current of 0.5 mA. We add a fuse between the computer's USB module and the FT232 to make sure that voltage spikes or unwanted current draw does not harm the USB module of the computer connected to the FT232.

2.11   Case

All these parts would need to be housed inside a box or case of some kind. Previous work done by this group used an enclosure from OKW Enclosures. The family of enclosures can be a seen in figure 2.12 [25]. The image shows enclosures of three sizes, and table 2.2 shows the sizes of the three models, along with their type as defined by OKW Enclosures. Work done by Huang [26] used the Ergo Large case, which is the largest case in the Ergo Minitec family. Those devices were not designed to be worn all day, but our device is meant to be worn for a full day or multiple days. We ordered samples of all three sizes to experiment with, and try to make our device as small as possible.

Type	Length	Width	Height
Ergo Small	52 mm	32 mm	15 mm
Ergo Medium	68 mm	42 mm	18 mm
Ergo Large	78 mm	48 mm	20 mm

Table 2.2: Comparison of sizes the Ergo Minitec case is available in.

2.12 Battery

While the wrist activity monitor is mounted and active, it needs a power source to operate. The SHIMMER [17] uses a single 450mah lithium-ion polymer battery. Similar batteries with higher capacities are found in mobile phones. We also see other batteries available in the market. Our wrist motion activity monitor requires a small battery which will supply a burst current of $\approx$25mA when required. These bursts occur when memory has to be written to or during an erase operation for a memory page, since the memory requires higher current to process these operations. It should also hold enough power to supply current to the system for a minimum of 24 hours. Table 2.3 shows the different current battery types available in the market (as seen in [27]), and their parameters. From this table we see that lithium-ion and lithium-ion polymer batteries have identical characteristics. The difference between the two is the electrolyte used in creating these batteries. Lithium-ion batteries use a liquid, while lithium-ion polymer batteries use a gel based electrolyte. This allows lithium-ion polymer batteries to be smaller, and require less hardware in containing the electrolyte. For slim geometry, the only choice is a lithium-ion polymer battery [27]. After looking at work done by SHIMMER and our group earlier [17, 11] we decided to consider lithium-ion batteries. Another field where lithium-ion batteries are used is in flight. This field has requirements similar to our application, because the aircraft must be very light to allow for easy take-off and flight control. Since aerial vehicles need to have a low weight, batteries used in this application are high density and very efficient. However, these need a much larger battery, for example a 5000 mAh lithium polymer battery [28]. We considered the batteries used in the fields mentioned above and compared some of them. The comparison is given in table 2.4.
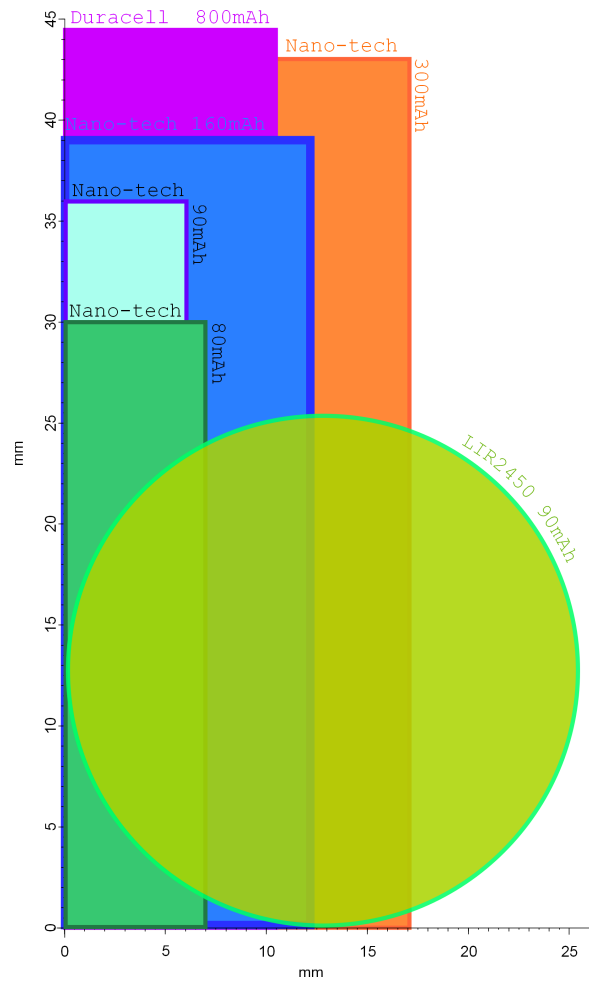
Figure 2.13: Top view profiles of the compared batteries.

Battery Type	NiCd	NiMH	Lead Acid	Li-ion	Li-ion polymer	Reusable Alkaline
Cycle Life (to 80% of initial capacity)	1500	300 to 500	200 to 300	500 to 1000	300 to 500	50
Fast Charge Time	1h typical	2-4h	8-16h	2-4h	2-4h	2-3h
Overcharge Tolerance	moderate	low	high	very low	low	moderate
Self-discharge/Month (room temperature)	20%	30%	5%	10%	10%	0.30%
Cell Voltage (nominal)	1.25V	1.25V	2V	3.6V	3.6V	1.5V
Peak Load Current	20C	5C	5C	>2C	>2C	0.5C
Preferred Load Current	1C	0.5C or lower	0.2C	1C or lower	1C or lower	0.2C or lower
Maintenance Requirement	30 to 60 days	60 to 90 days	3 to 6 months	not req.	not req.	not req.

Table 2.3: Comparison of different battery technologies.

Product Name	Category	Profile	Capacity	Length	Breadth	Thickness	Weight (g)	Discharge Rate (C)
Turnigy Nanotech	Microcopter	Cylinder	90 (mAh)	36 (mm)	N/A	6 (mm)	3	15
Turnigy Nanotech	Microcopter	Cylinder	80 (mAh)	30 (mm)	N/A	7 (mm)	2	15
LIR2450 (with Holder)	Wristwatches	Button Cell	90 (mAh)	22.6 (mm)	25.4 (mm)	8.9 (mm)	6	2
Turnigy Nanotech	Microcopter	Cube	160 (mAh)	39 (mm)	12 (mm)	8 (mm)	4	10
Turnigy Nanotech	Microcopter	Cube	300 (mAh)	43 (mm)	17 (mm)	8 (mm)	8	35
Duracell Staycharged (AAA)	Toys	Cylinder	800 (mAh)	44.5 (mm)	N/A	10.5 (mm)	11.5	2

Table 2.4: Comparison of selected Lithium-ion Polymer batteries available.
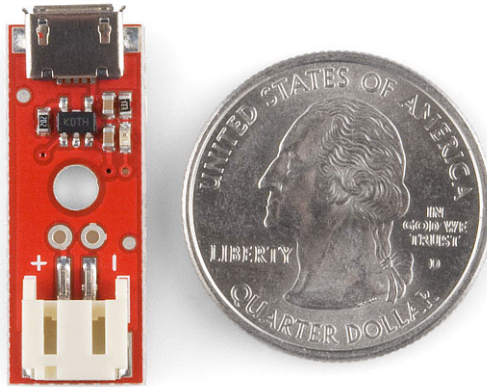
Figure 2.14: Breakout board with the MCP73831. Quarter shown for size comparison.

A visual comparison of the different battery sizes is shown in figure 2.13. As can be seen, apart from the LIR2450, the battery sizes are somewhat similar, so it may be possible to use a larger battery while still fitting our hardware in our case. The width of the batteries would be an issue too. As seen in section 2.11, the case is curved along its median, so a battery that was thicker could be accommodated if it was not too wide. Empirical analysis showed that the Turnigy Nano-tech 160mAh battery would fit with some modifications into the smallest Ergo case. These modifications included grinding the inside of the case to increase the space inside the case, allowing a slightly thicker battery.

2.13 Battery charger

If the device is being used similar to a wrist watch, it is not feasible to dis-assemble the device to change or recharge the battery. For this reason, we require a built-in battery charging system. This battery could be charged when the wrist monitor is connected to a computer for transferring data. Lithium polymer batteries require a differential charge which is based on the voltage held by the battery cell. These batteries should not be overcharged, as that can cause the battery to heat, decrease the life cycle of the battery, and also cause it to explode. Kioumars et al. demonstrate a sensor to monitor heart rate and body temperature [29] which uses the MCP73831,

a miniature lithium-polymer charge management controller. The MCP73831 is shown in figure 2.14 [30]. This part accepts a voltage between 3.75V to 6V, and outputs a charge current based on the status of the battery cell connected to it. When the MCP73831 detects a battery is charged to its maximum capacity, it stops supplying current to avoid harming the battery.

2.14   LED

When the wrist activity monitor is worn, the user would require some kind of feedback on its operation. Any form of display would consume a large amount of current, which would reduce the battery life of our device by a large magnitude. We decided that the only indicator the device would have would be on its status. Taking a cue from Bluetooth^{TM} headsets, we designed a system where a single light would blink to indicate its status. To indicate that it is currently operating, it would blink for a very short period roughly every 10 seconds. After 1KB of data was recorded to the microcontroller, it would be flushed to the memory chip, this would require using SPI commands to store this data to the memory chip's buffer, and issuing a special command to write this buffer to a page on the memory chip. We empirically determined that this activity would repeat every 11.33 seconds. Instead of blinking the LED every 10 seconds, we changed the process to turn on the LED every time the data flushing process would begin, and turn off at the end of this process. The result was a light that would blink every 11.33 seconds for 54 milliseconds. The hardware for this light was implemented using an LED. Since the LED is being turned on for very short periods of time, (the duty cycle for this operation being 0.37%) very little power is being consumed by the LED. For this reason, we used an LED found in our department lab. A photograph of the LED is seen in figure 2.15.

2.15   Button

As with any consumer electronic device, we require the option to turn our device on or off. We might also want to provide our user with the option of erasing the memory for the device. This would require some kind of button (or buttons) to allow such interaction. We consider a scheme that required one button, with a state machine behavior as given in table 2.5. Based on this state machine behavior, we recognize that a single pole single throw momentarily on switch is the most
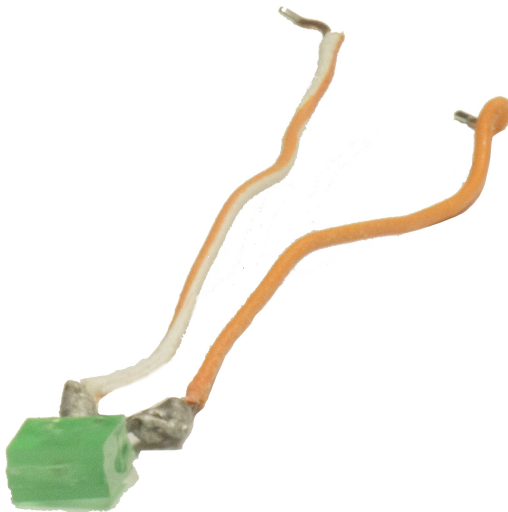
Figure 2.15: Photograph of the LED we used.

convenient device for our purpose. This button can be see in figure 2.16 [31]. A momentary on button is similar to those seen on a remote, where the button switches on momentarily when pressed, and then deactivates as soon as pressure on it is released.

Current State	Action	Result
Off	Quick button tap	Device turns on and starts recording.
Off	Hold button for 3 seconds	Memory is reset.
On	Quick button tap	No action.
On	Hold button for 3 seconds	Device stops recording and enters sleep mode.

Table 2.5: Behavior of the device as a state machine.

Figure 2.16: Example of a momentary switch.
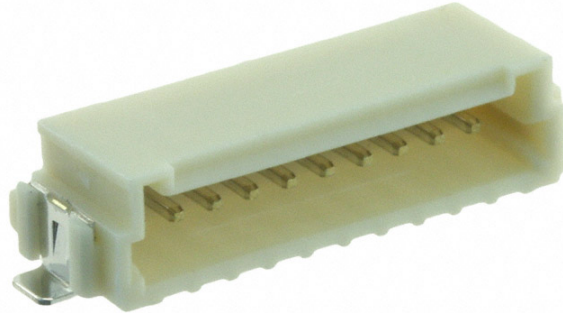
Figure 2.17: The 10 pin connector used to connect a JTAG cable to the microcontroller.

2.16   Connectors

Instead of soldering wires to our device (so that we can connect the JTAG cable or the USB to UART bridge to a computer), we prefer to use connectors. This allows us to disengage with ease without having to cut the cable or de-solder it from the device before it can be used. The JTAG connection required 7 pins to connect to our microcontroller, and a standard seven pin header is 17.78 mm X 4.85 mm in size. The pitch (distance between two pins) in a standard header is 0.1 inches or 2.54 mm. Since space on the PCB is at a premium, we use a smaller header instead, with a pitch of 0.059 inches or 1.50 mm. This allows us to conserve space on the PCB. Searching for connectors with a pitch of 1.50 mm showed us connectors that had housing around them. An example of one such connector with housing can be seen in figure 2.17. From the datasheet of this connector [32], we see that the housing increases the thickness of the connector 2.5 times, increases the length by 3 mm, and the width by 2.1 mm. The connector we selected was available only in a 10 pin configuration. To reduce this use of space, we cut the connector to its bare minimum. The housing was cut off using an Exacto knife, and three pins were cut off because they were not needed. The final connector is shown in figure 2.18. For the USB connection, we required only 4 pins. This would require a space of 9 mm X 2 mm, and could be squeezed between the other components on the PCB. We used IC hooks to connect to the holes in the PCB, with the hooks leading to a USB
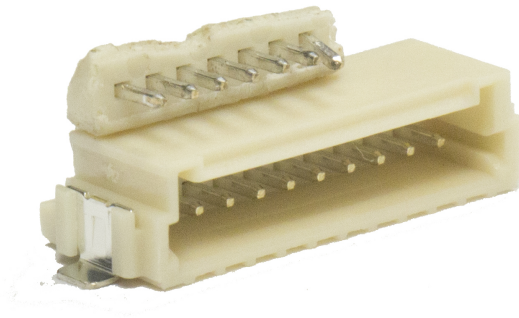
Figure 2.18: Photo showing the size difference after removing the enclosure and the original connector.

cable connected to a computer. These hooks allowed a good connection between the PCB and the

computer which would last for the 30 minutes it took to transfer memory contents.
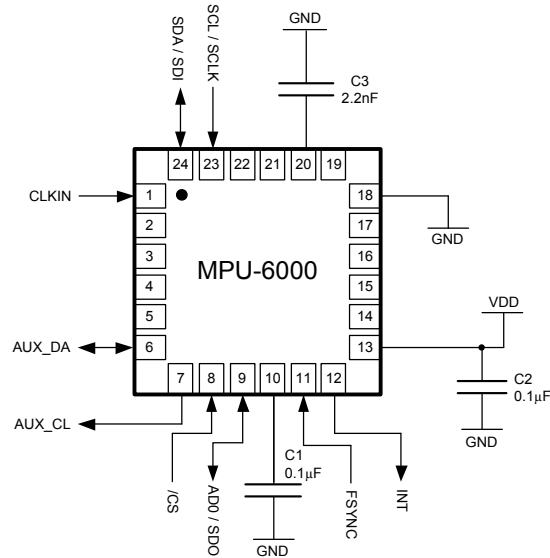
Figure 2.19: Typical operating circuit for the MPU-6000

2.17   Prototyping the device

We need to connect the parts previously mentioned correctly so that they can communicate with each other. Most electronic components are accompanied by a datasheet created by its manufacturer which details the behavior of the component, its electrical and non-electrical characteristics, its physical design, pin layout, and a typical application circuit. This typical application circuit allows us to connect the component in the way intended by its manufacturer. An example of this application circuit is a clipping shown in figure 2.19 from the datasheet [33] for the MPU-6000. As we can see, the typical application circuit for the sensor shows us the pin layout of the chip, along with any external components that need to be connected. In this case, it shows three capacitors that must be connected to the chip. Similar to the MPU-6000, all the components we have selected for our device design have their respective datasheets which contain information on how they should be operated. Using these datasheets, a master circuit was created for our device. This circuit was designed in the EAGLE PCB Design software.

2.17.1   Overview

All components in the final device are connected to each other using a printed circuit board. A PCB replaces wires between components by copper traces on the board, which are lines made of
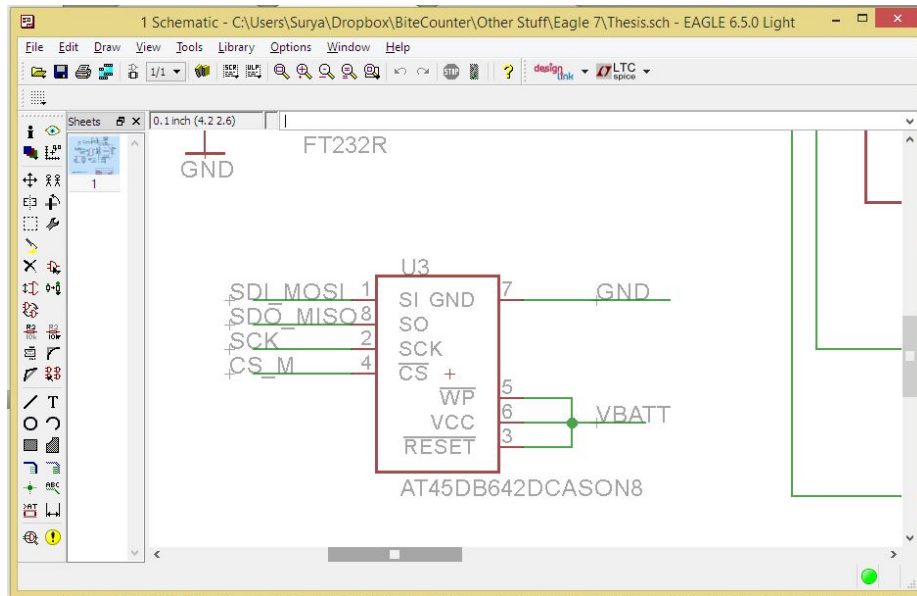
Figure 2.20: Screenshot of EAGLE PCB's schematic editor.

copper, so that electricity is conducted across them. Creating a PCB requires three steps, the first being circuit design. Once the circuit is designed, the components are laid out on a PCB, and traces are created based on the circuit design. The third process comprises of using this layout to create the actual PCB. This third step is completed by a PCB manufacturer after we provide them with our PCB layout design. The benefit of creating a schematic in EAGLE instead of directly creating a layout connected by traces is that the software uses the connections in the schematic to guide us while connecting the pins or pads of the different components. This reduces the chances of an incorrect connection between two components, reducing any mistakes that might creep into our final PCB design and cost us money as we re-manufacture our PCB.

2.17.2   Unit testing

We need to test each component separately and make sure they work as expected. Although most parts have a low failure rate, it is easier to work through with our devices knowing where a bug is if one shows up. Unit testing is a method of testing individual units of source code to check if they work as expected before introducing them into a larger piece of code. We extend this to our device creation process by testing each unit of hardware separately. We would have to test the microcontroller, the sensor, the memory chip, the USB to UART bridge and the battery charger

34
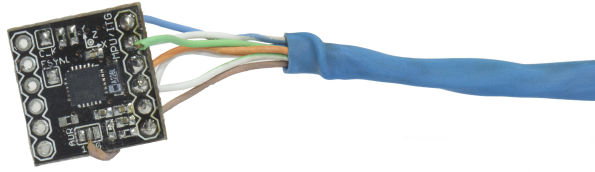
Figure 2.21: Photo of a MPU6000 breakout board connected to a CAT5 cable

separately. The microcontroller would have to be the first component to be tested because the other components require a master device that controls them.

2.17.2.1 Micrcontroller testing

To test the microcontroller we used a simple "Hello World" program. Since microcontrollers do not have a display, instead of displaying text, it is easier to just blink an LED using one of the pins. Pseudo code to blink the LED is show in in listing 2.1. We programmed our microcontroller using IAR's Code Composer Studio, and then flashed the microcontroller using the software and the MSP430 Flash Emulation Tool. If our connections were correct, the LED would turn on for one second, then turn off for another second. This cycle would repeat indefinitely, creating the illusion of a light blinking slowly.

Listing 2.1 :   Program to blink LED.

```
/* We want to blink forever */
while(1)
{
   /* Create a delay. Our clock is at 32768 Hz */
   __delay_cycles(32768);
   FlipLEDStatus();
}
```

2.17.2.2 Sensor testing

After confirming that the microcontorller chip is performing as expected, we tested our sensor. Breakout boards were previously discussed in section 2.5, which allow us to prototype sensors without worrying about soldering them. Breakout boards for the MPU-6000 were obtained from an Ebay seller. The breakout board has 6 pins that need to be connected for operation, which means we need 6 wires between the microcontroller and the breakout board. These pins on the breakout board were connected to the microcontroller target board using a CAT5 cable as can be seen in figure 2.21. This is the same cable used for Ethernet connections, and thus was easy to obtain. As shown in the same figure, a regular CAT5 cable consists of 4 pairs of twisted wires, so one pair in this cable was not needed. This pair was connected to the GND pin on the microcontroller to reduce noise.

The MPU6000 uses SPI for communication. The datasheet mentions that once the sensor is powered on and awake, the sensor will start recording what the acceleration and angular velocity are. This information is stored in an internal memory on the sensor which can be read using SPI instructions. We initially had trouble verifying if our sensor is working correctly and did not know if the sensor was faulty, or if there was a mistake in our connections or code. The datasheet for the sensor mentions a register called "WHO_AM_I" which contains he device's I^2C address. This address by default is set to 0x68. Since this register contains constant data, we can read it to check if the device is indeed correctly connected.

After some troubleshooting we learned that the length of the cable was too long for SPI communication and the signal would lose strength. Also, SPI is time sensitive, so if the cable length was too long, the delay created between two times would be too high, causing incorrect data to be received. We fixed this by reducing how long the cable was. In the final design the length the signal would have to travel would be very small since the chips would be laid out next to each other, so this was not an issue we needed to worry about. Once it was established that the sensors were working correctly and SPI communication between them was also as expected, we carried out simple tests where the acceleration on one axis would turn an LED on if positive, otherwise turn it off. Since gravity would show as -1 g on the Z Axis in the earth frame of reference, we could test all three sensor axes by rotating the sensor and observing the LED's behavior. These tests concluded that communication between the sensor and microcontroller was as expected.
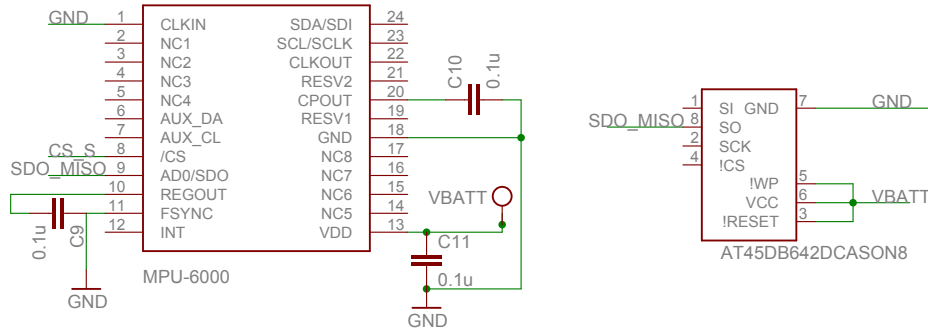
Figure 2.22: Circuits designed for the MPU 6000 (left) and the AT45DB642D (right).

2.17.2.3 Memory chip testing

Our memory chip was tested to check if SPI communication was up and running. As seen in Section 2.7, we created our own breakout board to break the pins out from the memory chip. The datasheet for our memory chip mentions a command known as "Manufacturer and Device ID". Issuing this command to the memory chip requests the memory chip to send 4 bytes of data to the master device. These bytes identify the device and contain other information about it. For example, the second byte contains the family code and the density code, telling us what the size of the memory is. We only need the first byte, which is 0x1F. Once we connected our memory chip to the microcontroller we sent this command from the master to the slave, and received 0x1F. This confirmed that we had connected the memory chip correctly.

Our next step was to store data from the sensor into the memory. Since both the devices use the SPI bus, it was possible to connect the devices to the same pins on the microcontroller. Before reading sensor values or storing them, we check if the device ID's are received correctly. This was done every time the microcontroller booted up. On our initial attempt, the memory chip did not respond with the correct device ID. This meant that even if we were sending correct data, it was not being received correctly with the new connection. After some testing we realized that this was because the cable to the sensor was too long for SPI, and timing errors were being created. Once we reduced the length of this cable, data transfer behaved as expected, and we were able to record data from the sensor, then flush it to the memory.
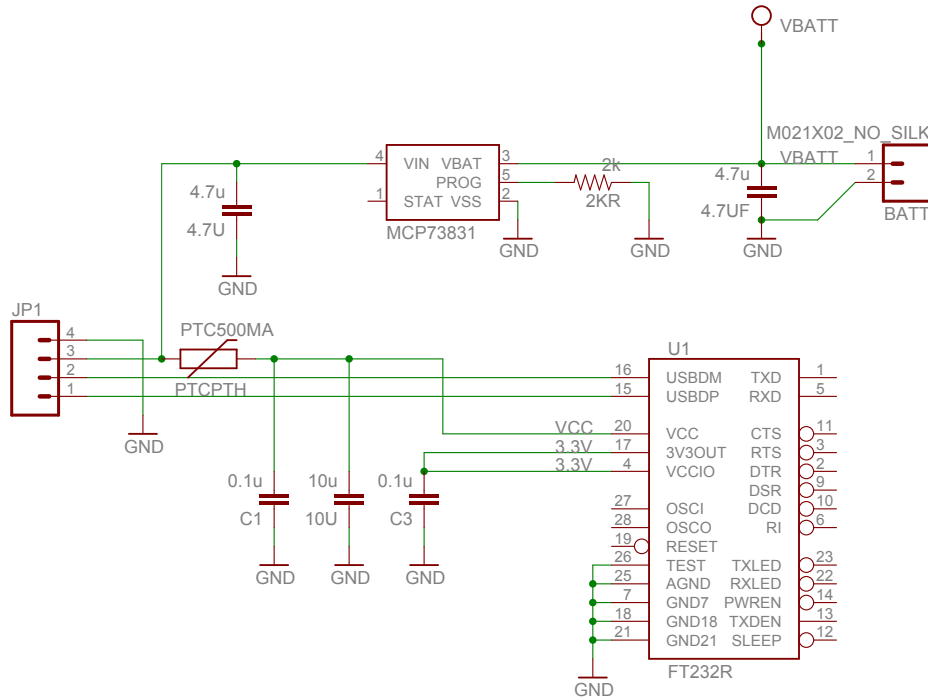
Figure 2.23: Circuits designed to power the circuit and connect the USB to UART bridge.

2.17.3 Circuit design

The circuits for different parts of the device that we designed are shown in figures 2.22, 2.23, and 2.24. The complete circuit we designed after consulting the different datasheets is shown in figure 2.25. This circuit was created using EAGLE PCB's schematic layout tool. Each component had its circuit created separately, and later these circuits were connected together to create the final device. A screenshot of Eagle PCB's schematic editor is shown in figure 2.20. We used a breadboard to prototype the device once we had a circuit design available. The breadboard allowed us to connect our components together and program the microcontroller with a reduced amount of soldering, and this can be seen in figure 2.26.

2.17.4 Programming and testing

With our components connected, it was possible to program our microcontroller and make it perform the required functions. Our program polls the sensors repeatedly. The data from these sensors is then stored in the microcontrollers memory until the total data crossed a threshold. Once this threshold was crossed, the data would be sent to the memory chip. A simplified version of the
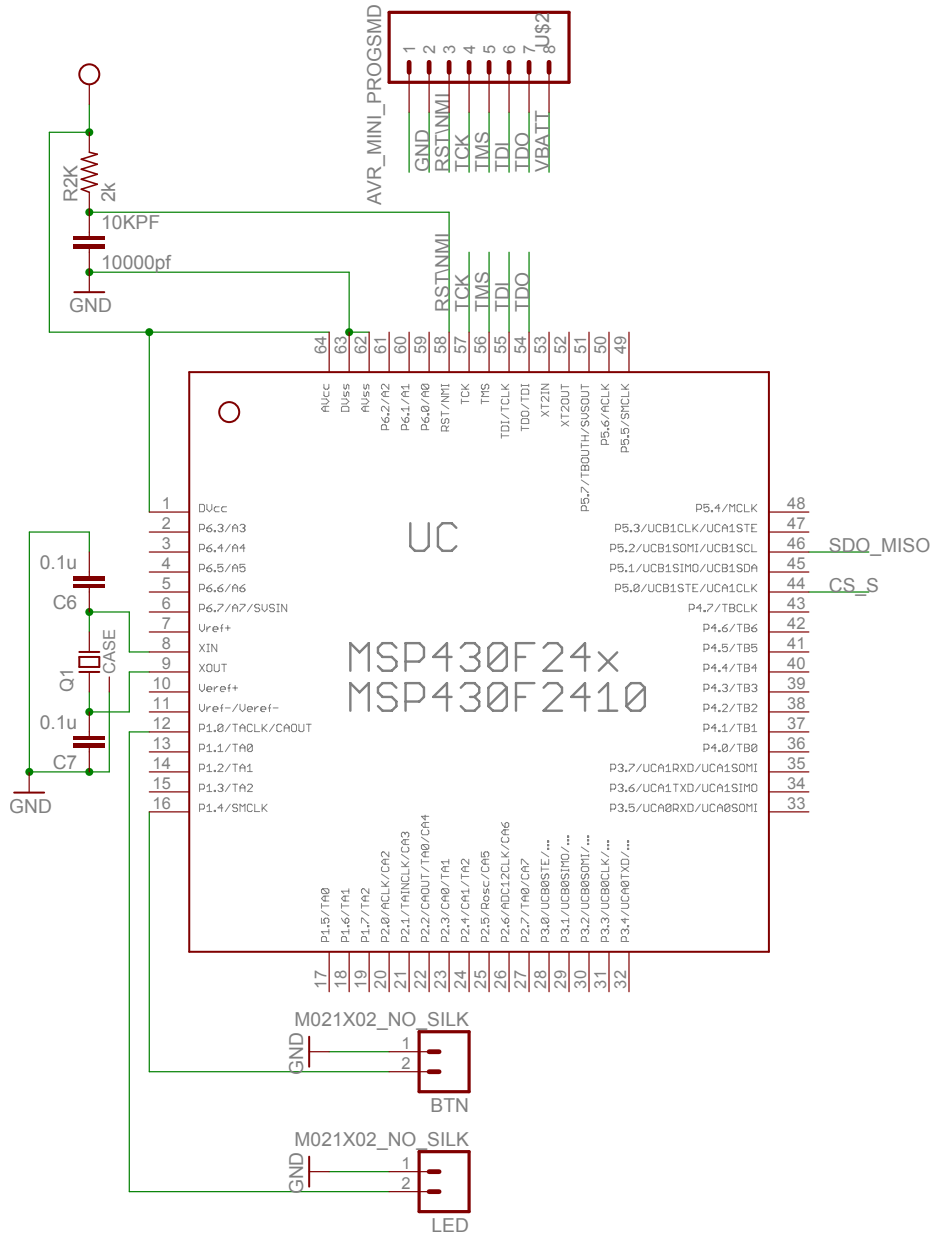
Figure 2.24: Circuits designed to operate the MSP430F248.
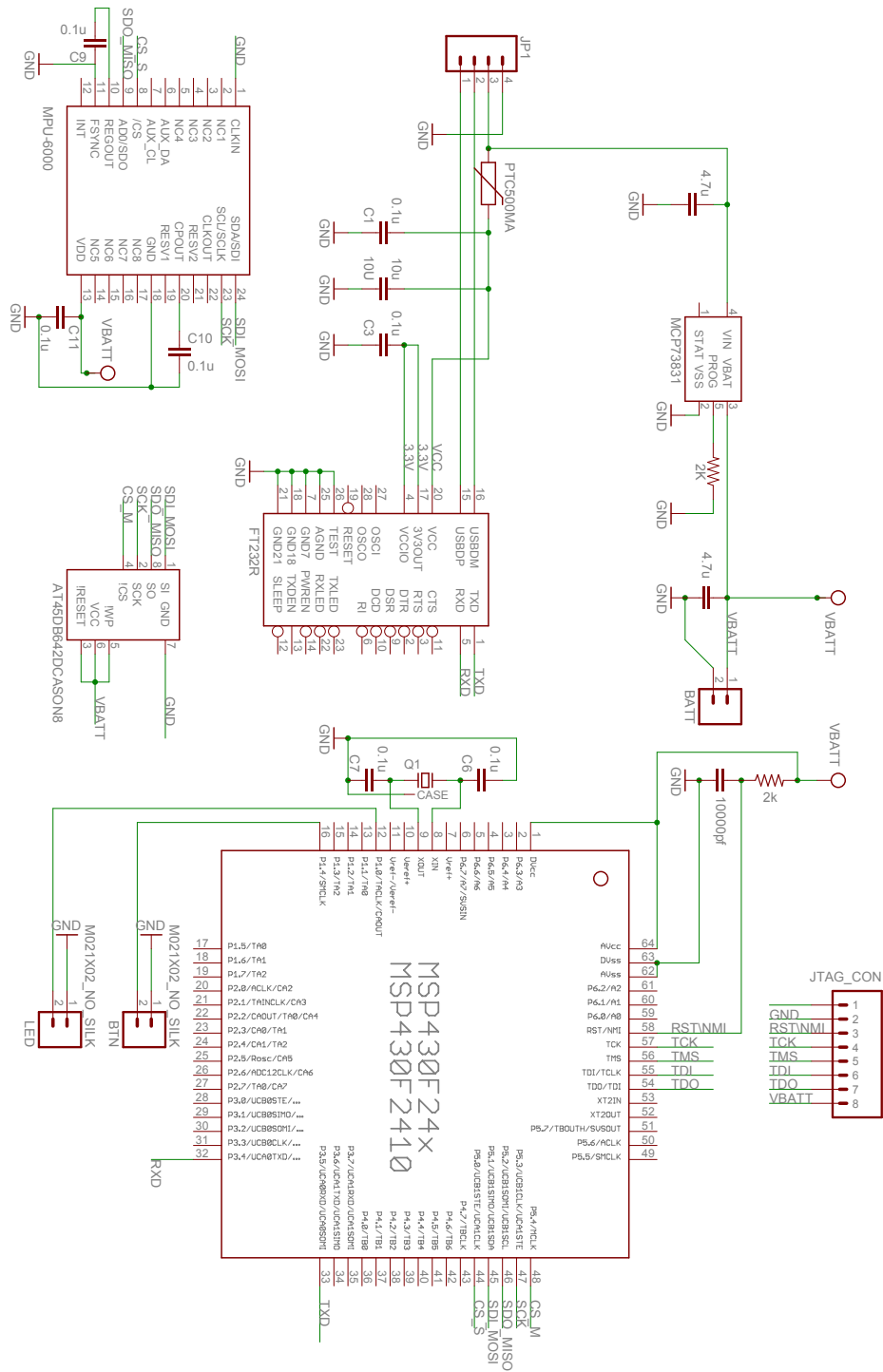
Figure 2.25: The complete circuit for the wrist motion activity tracker.
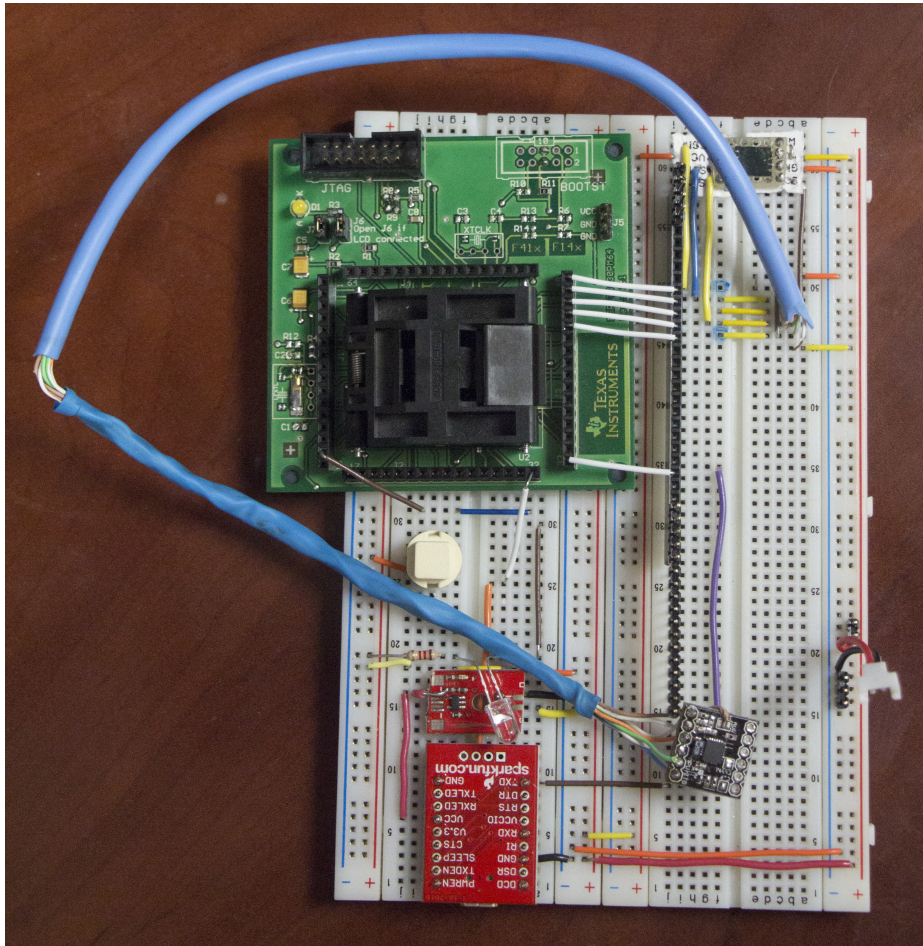
Figure 2.26: Photograph of prototype made using a breadboard. Not shown: USB cable for connection.
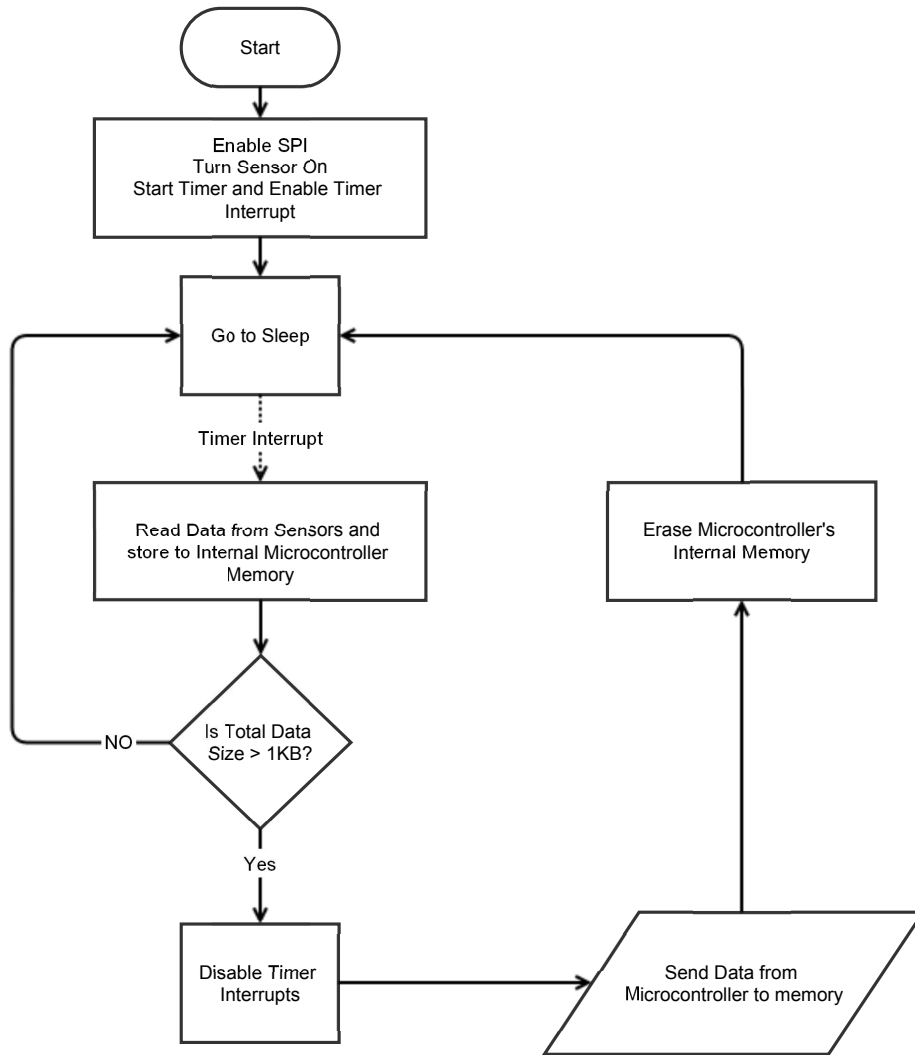
Figure 2.27: Flowchart showing simplified algorithm used for the device.

microcontroller code is seen in figure 2.27. To test the battery life of our device, a video stream was setup which had a camera pointed at the device. We connected a multimeter measuring voltage to the battery, and had its display visible in the frame. The video stream also had a time stamp to show how the battery voltage dropped as the device operated. Since the device would blink an LED every 11.33 seconds, it was easy to monitor the device operation through this video. When the device stopped operating, the LED would stop blinking. We could watch the video to see the exact time when this happened, and have an accurate idea on the battery life of our device. These experiments were repeated three times each for the 90mAh and 130mAh Turnigy Nano-tech batteries.
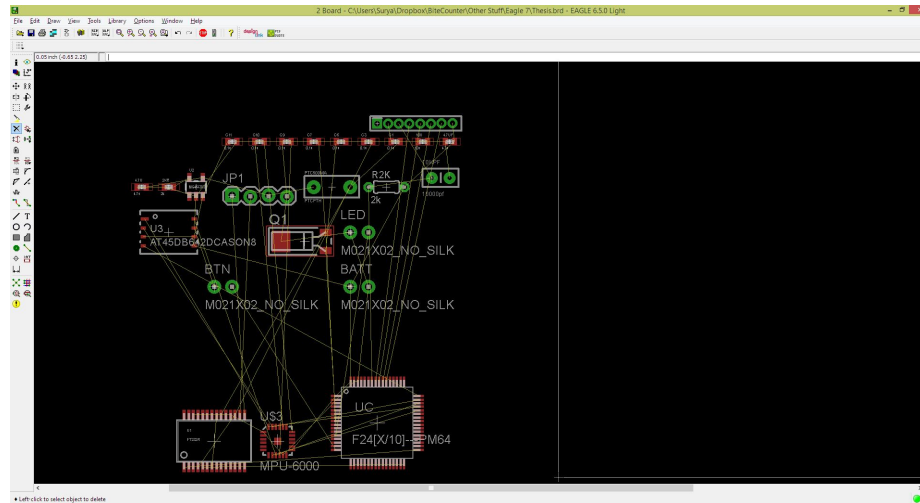
Figure 2.28: Screenshot of Eagle PCB's layout editor.

2.17.4.1   PCB layout

The PCB was laid out using EAGLE PCB's layout editor based on our tested circuit design. The software first generates parts and their footprints from the schematic we created. This can be seen in figure 2.28. The parts are placed in randomized locations by the software. The software connects these parts by direction lines (seen in yellow in figure 2.28) called signals. We sized the PCB to be smaller than the case so that it would fit easily, and moved the parts around so that pins that need to connect components are close to each other. This can be see in figure 2.29. The signals were routed through the copper board, using the signals as a guide, and this process is shown in figure 2.30. This process is like a puzzle where we are trying to draw lines through objects that we cannot cross. Seven revisions were required for our final PCB.

2.17.4.2   PCB manufacture

PCBs were manufactured by OSH Park[4]. Most PCB manufactures expect a high volume of production (100 - 1000 PCBs per order). We only required a few prototype PCBs so our order was only a few PCBs. OSH Park takes multiple orders from its clients and combines this into a large PCB. This is then cut into the requested PCBs and sent back to the client. This process takes time, but costs less than what most other PCB manufacturers charge. For example, Advanced Circuits LLC, a well known PCB manufacturer charges $120 for 4 bare bones PCBs, and has a turnaround

---

[4]http://oshpark.com
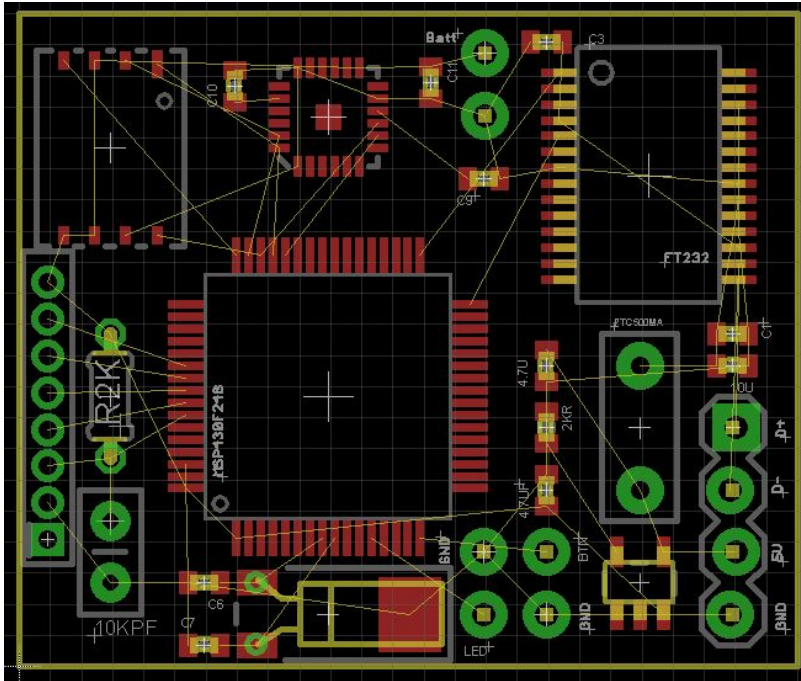
Figure 2.29: Parts to be routed after placing on the PCB in EAGLE PCB software.
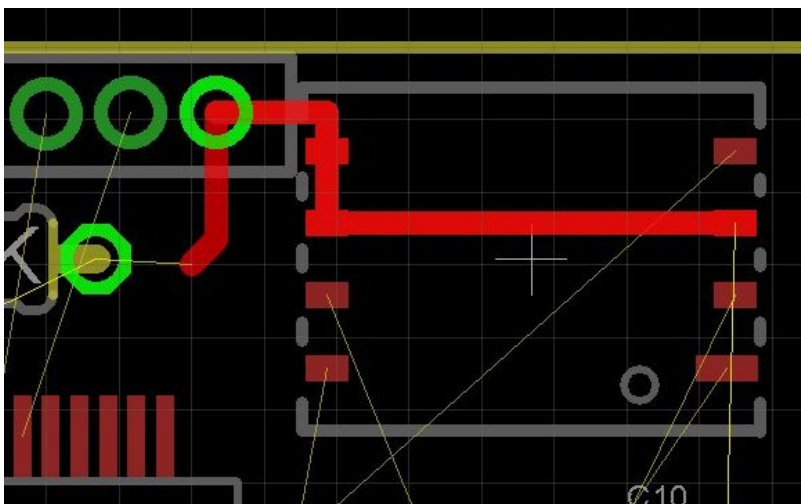
Figure 2.30: Routing in EAGLE PCB. The incomplete (thick) red line is guided by the (thin) yellow signal line.

time of a week. These PCB lack any kind of silkscreen, or printed text, which means it is hard to know where parts are located, or what their value is. Since there is no silkscreen, the copper traces on the PCB oxidize over time and lose conductivity. OSH Park, on the other hand offers high quality PCBs at a lower cost. The turn around time is much longer (approximately 21 days), however we were able to procure 6 PCBs for a total of $15, costing us approximately $2.5 per PCB.

To create our PCBs, we uploaded our PCB layout file generated by Eagle PCB to the OSH Park website. The website shows you an approximate render of how the PCB will look once it is delivered, and also notifies you if it thinks there could be issues with part placement (for example, parts may be too close to the border).

2.17.4.3   Software

Work done by Concha [34] shows PhoneView, a tool developed to analyze data collected by an iPhone. Instead of the wrist based activity monitor that we are developing, work done by the group previously used an iPhone mounted on the wrist. This data could be loaded into PhoneView, which would then plot the instantaneous values of each sensor versus time, allowing us to process this data as signals. PhoneView would accept data that was stored in ASCII. This means that each sensor reading would require three digits, each requiring 1 byte. Raw data that was 8 MB in size would expand to 24 MB if stored as ASCII. We modified the source code of PhoneView to allow for our data from the wrist motion activity tracker.

2.17.4.4   Soldering

A photograph of the bare PCB can be seen in figure 2.31. As can be seen, most components are going to be surface mounted to this PCB, and there are very few through hole components. Throughout this thesis we have emphasized on how small the components we are dealing with are. These components are so small that we lost a couple of the parts when trying to handle them because someone was breathing too hard. In the industry, pick and place machines are used to move these components around. Solder is first applied to all the pads. These machines then use vacuum to pick up the delicate components, and cameras to inspect them. Once the components are placed on a PCB, the PCB is sent to an oven where the solder melts, and the components are soldered once it cools down. We did not have access to an industrial facility with these features, so we would have to use a different technique to solder these components. Our first attempt was mentioned when we
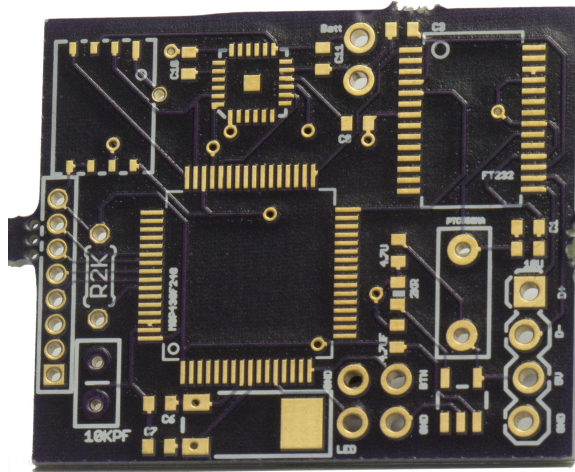
Figure 2.31: The bare PCB received after fabrication.

discussed our memory chip in section 2.7. As seen, we flipped the chip over to expose its pads, then soldered thin strands of a multi strand wire to the chip. These strands connected to thicker wire at the other end, which allowed us to use the chip as a through hole component. All this was done very carefully under a microscope. The MPU-6000 integrated chip has pads that are much smaller than what our memory chip did. It was not feasible to solder strands to this chip because of the same reason, so we researched for other techniques.

2.17.4.5   Z axis conductive tape

Sparkfun.com demoed a new product from 3M, called the Z-Axis Tape, just as we were looking for methods to solder our sensor [35]. The Z-Axis Conductive Tape is described by Sparkfun as "an easy to use, pressure sensitive double sided tape designed for connecting, bonding and grounding flex circuits and PCBs." In theory, we should be able to place this tape between the PCB and our integrated chip, and have conductivity between the pad on the PCB and those on the chip. The datasheet for this conductive tape mentions that it is filled with small conductive particles which allow it to conduct electricity through its thickness, however these particles are spaced far enough to maintain electrical insulation. The datasheet also mentions that the minimum distance between two adjacent conductors should be 0.4 mm or greater to ensure electrical insulation, however, the method used to bond the parts together and temperature would affect this number. The MPU-6000 datasheet mentions a pitch of 0.25 mm between two pads, which was smaller than the suggested dis-

tance of isolation by the tape. However given that the tape was expected to operate under extreme conditions of temperature (-40C − 70 C), we decided to try this tape to bond our sensor chip to the PCB. Our experiment showed that sufficient conductivity was attained after 24 hours to allow SPI communication between the sensor and the microcontroller, however this conductivity was lost over time, within seven days in our case.

This process concluded that the Z-Axis tape was not a permanent alternative solution to soldering the sensor to our PCB.

2.17.4.6   Reflow Skillet

As an incentive to its clients, Sparkfun offers tutorials on multiple topics, and has a tutorial on different soldering methods [36]. This tutorial mentions four different methods that a hobbyist electrical engineer can use to prototype a device based on SMT components:

Hand Soldering

This is the regular method of soldering with a soldering iron that we have already tried with our memory chip.

Toasting

This process involves placing solder paste and parts on the PCB, then heating them in a temperature regulated toaster oven to the temperature required by the solder to melt. This melts the solder, and when cooled provides us with a PCB with components connected. We did not have access to a toaster with an accurate temperature control, and so could not test this method.

Industrial Ovens

Similar to the method mentioned above, this technique uses industrial ovens specially constructed for SMD soldering. We did not have access to these industrial ovens.

Hot Air Rework

This method uses a hot air blower that increases the temperature of the PCB to that of the solder melting point. This method is used frequently to desolder or rework SMT components. We tried this method, but did not get a high degree of success. One of the reasons for failure was that our parts would fly away under the air pressure, and it was not feasible to hold down the part in place correctly because our hands would shake while doing so.

47
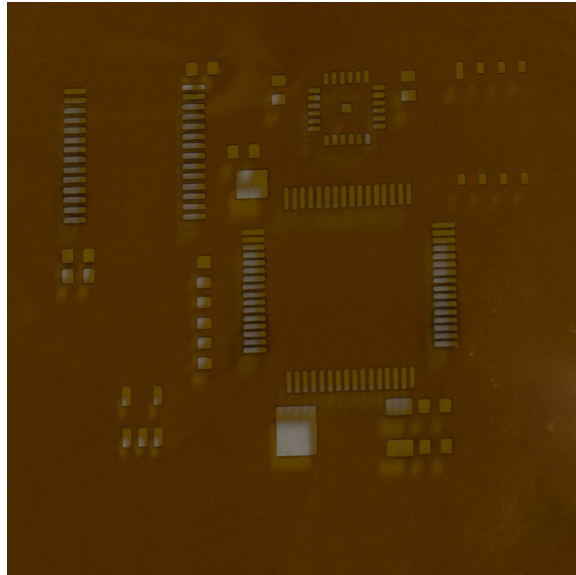
Figure 2.32: Photograph of the stencil used to apply solder paste to the PCB.

Hot Plate Reflowing

This method requires that pads on the PCB have solder paste applied to them. Once this is done, we place the components over their approximate locations on the PCB. After this is done, we place the PCB with the components over a skillet that is heated to the temperature of the solder. If everything goes right, the molten solder will behave like a liquid, and due to force of adhesion, parts will align themselves correctly in place.

Based on the different techniques mentioned above, we used the hot plate reflowing method. Similar to OSH Park, a website, OSH Stencils accepts EAGLE PCB layout files, and provides laser cut stencils for the PCB. This stencil was used to apply solder paste only to the exposed pads of the PCB. A photo of the stencil is shown in figure 2.32. We placed some components: the microcontroller, memory chip, sensor and the USB to UART chip on the PCB by hand. This was done because it was very hard to place the smaller components like capacitors and resistors by hand and not have them move other parts in the process. After about 23 seconds on the skillet, we could see the solder melt and the parts fall into place. Through hole components and smaller components like the crystal oscillator and resistors were individually soldered later by hand under a microscope. Figure 2.33 shows the hand soldering process through the microscope's lens. The Final PCB can be seen in
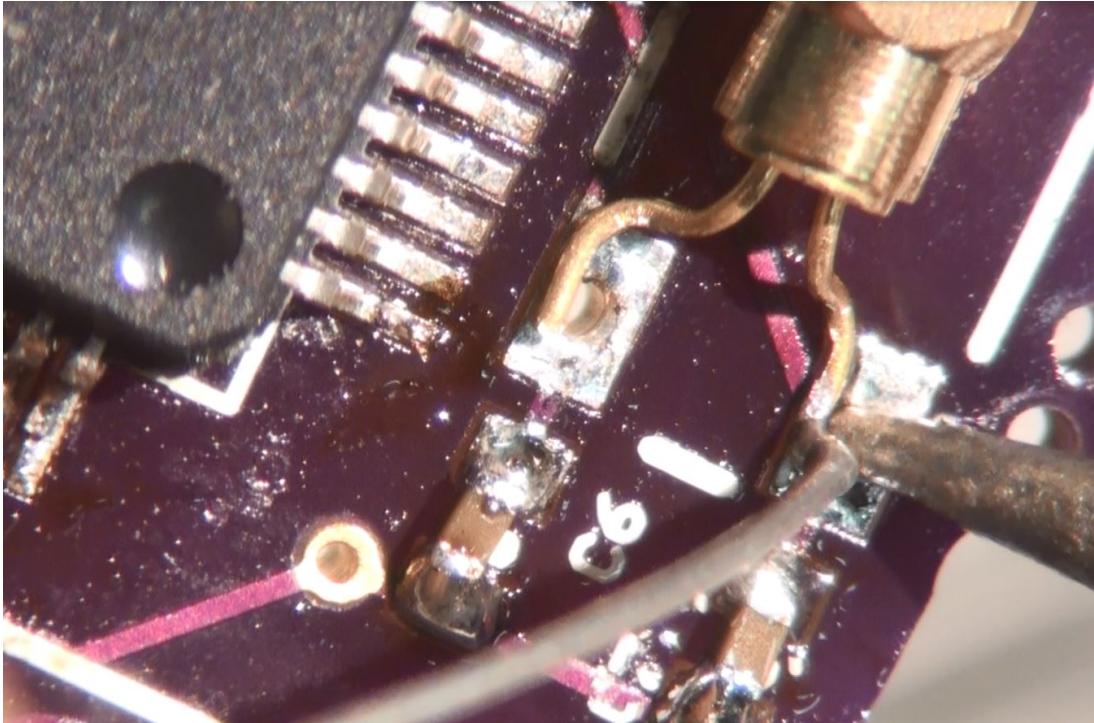
Figure 2.33: Photograph showing crystal oscillator being soldered under a microscope.

figure 2.34.

2.17.4.7 Final device

We programmed the microcontroller with the same code used in our breadboard based prototype. The case was modified to seat the LED and the button, with the LED and the button stuck to the upper side of the case using hot glue. The PCB was placed under this button, and the battery was the last part to enter the case. The case had a ring which allowed a wrist watch style strap to fit through, and we added this to the case.

2.17.5 Verification of logged data

Now that our device was ready to be worn on the wrist, we performed some experiments to verify that it works as expected. Our aim for this device was to create a wrist motion activity tracker that would track wrist movements all day, and also be comfortable when worn for extended intervals. We would have to test the comfort of the device, and also verify that the data it was reporting was

Figure 2.34: Photograph the PCB after soldering, along with top side of case.

correct. To check if the data being reported was correct, we setup an experiment where the user would wear both the wrist motion activity tracker, and the iPhone from the PhoneView experiment performed by our group earlier. The user would then make characteristic movements which could be identified easily by looking at the signals in PhoneView or WristView.

Chapter 3

Results

Our proposal for this thesis was to create an all day wrist motion activity monitor that was small, cost effective and comfortable. This device would also have to be able to record data for an entire day of activity, decided as 16 hours. The final device we have created can be worn on the wrist like a watch by using its strap. Figure 3.1 shows a photograph of this arrangement when the device is worn around a wrist. In this chapter we consider how we have compared against the requirements for the wrist activity monitor that we had set. Users are expected to wear the device at the start of the day, and press a button to start recording wrist motion movement data. At the end of the day, this device can be connected to a computer though a USB port and the data that has been logged can be transferred. This data can then be analyzed or processed on the computer as needed. We use WristView, a modified version of a software previously created by our group to visually display the data logged by our device.

3.1   Size and comfort

The final PCB and components fit comfortably inside an OKW Ergo Minitec Small enclosure. The dimensions of the device are the same as the enclosure, 52 mm X 32 mm X 15 mm. A strap that is 12 mm wide and 20 cm long allows the user to wear this device on their wrist. The device weighs 26.6g, including the strap. The device was tested by two volunteers for 10 hours each, and reported to be similar to a wrist watch with regards to comfort. Note that this size is limited to the fact that we used a case available off the shelf. If a custom design process was used which

Figure 3.1: Photograph of the our activity monitor mounted on a wrist.
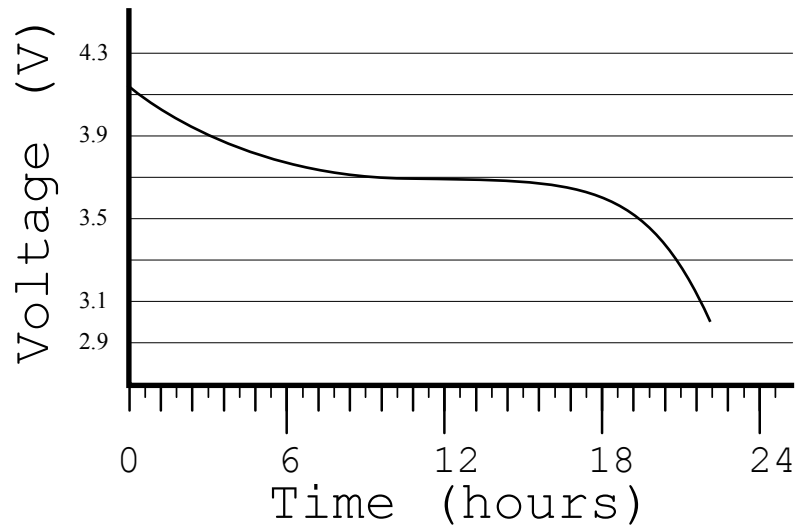
Figure 3.2: Graph showing battery life of the device.

would allow for a case to be molded, the size could be halved, or spread around like a strap.

3.2 Battery life

We needed to check if the battery life of the device met our specifications. Battery manufacturers are known to inflate the battery life on data sheets for their products. Current draw and how the battery is used also changes the output capacity of the battery. Section 2.17 describes the methods used to test our prototype, and the techniques used to measure the battery life of the product. After testing the battery three times, we concluded that the 130mAh battery would log wrist motion data for at least 24 hours, which was higher than the required battery life for our device, 16 hours. On standby mode (where the device is not recording any data), the battery life was up to one week. Figure 3.2 shows how the battery voltage drops versus time when it is actively recording motion data. Data is not shown once the voltage dropped lower than 2.8 V. The device was programmed to stop operating and enter standby mode once the voltage was lower then 2.8 V. This was to avoid recording incorrect data or corrupting the memory chip once supply voltage was near the minimum level for components on our circuit.
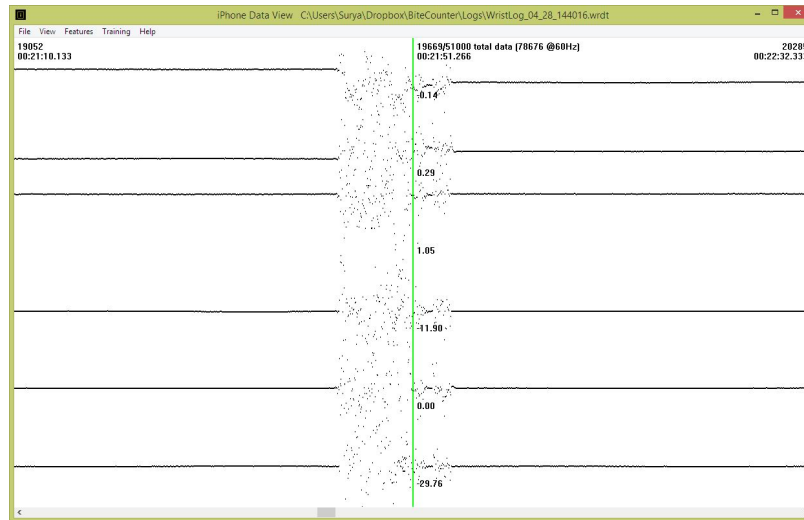
Figure 3.3: WristView displaying data captured by our prototype device. Top to bottom: Acceleration (X, Y, Z) and angular velocity (X, Y, Z)

3.3 Software

We had been able to log data from the sensors into a memory chip, and this data was then transferred to a computer in the form of a binary file. Each byte in the file represents data recorded from a sensor, with the time of the record being calculated by using the byte address. Work done by our group previously could display motion data collected by an iPhone in a visual format. We modified work done by Concha [34] and created our own software to display the data logged by the wrist motion activity tracker. This software, as mentioned in section 2.17, was named WristView and allowed us to visually compare data captured by the device. A screen shot of WristView can be seen in figure 3.3. The figure shows the data plotted as amplitude versus time. For acceleration data the amplitude was in g, where 1 g = 9.8 m/s^2. For gyroscopes, the data was in deg/sec. Raw data from sensors usually contains noise. WristView can smooth these signals based on code from PhoneView, which uses an algorithm used by the research group previously [34]. This can be seen in figure 3.4.

3.4 Device cost

Earlier, we had mentioned the different devices on the market marketed as research devices, and also the high costs attached to using them. The costs include expenditure on manufacturing

55
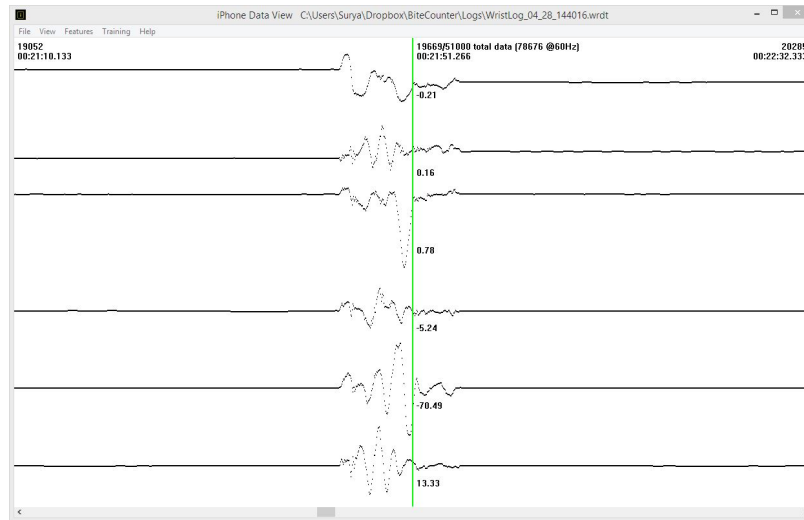
Figure 3.4: WristView's smoothed data feature.

and research. After our device was created, we tabulated the cost of the parts of the device, and this can be seen in table 3.1. The total cost for the parts used per device was US\$52.5. This number includes costs for PCB and stencil manufacture, but does not include any costs for the research of the device or hourly labor associated with assembly.

Description	Cost (US$)	Quantity
Capacitor, 0.1 uF	0.10	7
Capacitor, 4.7 uF	0.26	1
Capacitor, 10.0 uF	0.50	1
Capacitor, 10.0 nF	0.32	1
Resistor PTH, 2 kΩ	0.14	1
Resistor SMD, 2 kΩ	0.10	1
FT232RL	4.50	1
Fuse	0.35	1
MSP430F248	7.89	1
MPU-6000	14.9	1
MCP73831	0.67	1
Crystal, 32768 Hz	1.49	1
AT45DB642D	12.7	1
PCB Fabrication	2.99	1
Stencil	5.00	1
Total Cost	52.5	1

Table 3.1: Bill of materials for our device.

Name of Device	Programmability	Battery (hrs)	Weight	Cost	Type	Body Position	Dimensions (mm)	Features
Fitbit Zip	Low	6 months	8.0	$49[4]	Fitness tracker	Waist / Torso / Wrist	36.0 x 28.0 x 10.0	Accelerometer
Fitbit Flex	Low	120	N/A	$99[4]	Fitness tracker	Wrist	140.0 x 161.0 x 13.9	Accelerometer
Jawbone Up Move	Low	168	6.8	$49[4]	Fitness tracker	Torso / Waist	27.6 x 27.6 x 27.6	Accelerometer
Jawbone Up 3	Low	168	29.0	N/A	Fitness tracker	Wrist	220.0 x 12.2 x 3.0	Accelerometer
Fitbit One	Low	240	8.0	$99[4]	Fitness tracker	Torso	48.0 x 20.0 x 10.0	Accelerometer
Nike FuelBand	Low	96	30.0	$149[4]	Fitness tracker	Wrist	147.0 (circumference)	Accelerometer
iPhone	High	40[2]	148.9	$239[4]	Mobile phone	Hand held	115.2 x 58.6 x 9.3	Gyroscope, Accelerometer Phone calls, LCD screen Media player
MetaWatch	High	80	81.0	$249[4]	Smartwatch	Wrist	51.0 x 38.0 x 13.0	Accelerometer Display Bluetooth connectivity
Samsung Gear 2	Medium	48	68.0	$275[4]	Smartwatch	Wrist	58.4 x 36.9 x 10.0	Gyroscope, Accelerometer Display, Phone calls Bluetooth connectivity
Thalmic Labs Myo	Medium	48	93.0[5]	$149	Research	Upper arm	N/A	Gyroscope, Accelerometer EMG Sensors
SHIMMER	High	24[3]	23.6	$249	Research	Wrist	50.0 x 25.0 x 12.5	Gyroscope, Accelerometer, Bluetooth module
This Work	**High**	**24**	**26.6**	**$53**	**Research**	**Wrist**	**52.0 x 32.0 x 15.0[1]**	**Gyroscope, Accelerometer**

Table 3.2: Comparison of select wrist based motion tracking devices in the market.

Notes:

1: Size based off a pre-fabricated case available in the market. The device can be designed to be 40.0 mm x 20.0 mm x 10.0 mm.

2: Battery life for iPhone 4 while listening to music and display is turned off [5].

3: Information provided by Shimmer Customer support over email.

4: Information from an Amazon product page [7].

5: No specification sheet is available, but the weight was available on a product discussion page [6].

3.5 Comparison of devices

We see that our device cost is comparable to that of devices at the lower end of the spectrum. Fitness Trackers like the Fitbit Zip cost $49, while research oriented devices like the SHIMMER cost $249. When procured in a large quantity so that wrist motion data can be captured for 100 - 200 test subjects, devices like the SHIMMER prove to be very expensive. In comparison, our work costs $53 per device which allows us to obtain required quantities at lower costs. Most Fitness trackers that are inexpensive, however, do not allow custom programs or modifications. Our work allows a high degree of modification and programmability compared to other wrist mounted devices across the board. Table 3.2 shows only three devices which allow a significant level of programmability. Since our device can be programmed using our custom code it allows for a very high level of control. Due to this, we can capture a higher resolution of wrist movement data compared to other devices. Devices in table 3.2 either do not offer this data at all, offer it at a very low resolution (3 Hz for the Samsung Gear 2), or offer it for a short interval of time.

Devices that would suit our purpose, like the iPhone are not viable options due to their size or weight. Our device measures 52 mm X 32 mm X 15 mm and weighs 26.6 g, which makes it similar in size and weight to a watch. This size gives the wearer a good level of comfort compared to wearing a heavy device like the iPhone on the wrist. Most features of the device were obtained by limiting the components to what we required for our purpose of tracking wrist movement data. By not including components like a Bluetooth$^{\text{TM}}$ module, an LCD display or a speaker, we have managed to keep the device small in size. Note that this size was a limitation of the case we used, an off the shelf option. It is possible to reduce the size of the device much further by using industrial techniques and mass production methods to build a custom case. The same philosophy of minimal components has allowed us to have an active battery life of 24 hours, which is seen only in devices targeted towards research. These facts show that for the application of logging wrist movement data for extended periods of time, our customized device is the most economical option.

3.6 Prototype challenges

Throughout this work we have emphasized the small sizes of the parts we used. Integrating these parts into our device was a considerable challenge since we did not have access to industrial machines. Before any of our parts could be connected and used, they needed to be in a format that

was easy to connect to. The parts are manufactured to be used in an industrial setting, and provided in surface mount packages. An option available to us was to outsource the physical design of the device. Once we had the design ready, it could be provided to a manufacturer for soldering and assembling the device. However this would at least double the cost of our device in the prototype phase, and not an acceptable solution. Testing the different methods of soldering that could be used and then finally using the solder reflow method as explained in section 2.17 took a considerable amount of time.

Another issue we faced was devices overheating due to incorrect connection or a short circuit created during the soldering phase. To replace the burnt integrated chip, we had to cut off the burnt chip from the PCB using an Exacto knife, and then desolder the pins. This was done under a microscope.

Our first prototype had the device reporting incorrect motion data. This was due to the fact that the SPI lines were extended beyond the typical specifications ($\approx$150 mm), and the signal's power would drop rapidly as it traveled. We accommodated for this in our PCB based prototype, where the signal traveled very short distances ($\approx$5 mm).

Chapter 4

Conclusion and Future Work

In this thesis, we proposed to create a device that would record data on the movement of a human wrist during a day. This was motivated by the fact that wrist motion information can be used to count the number of bites taken by a user, or predict the tool being used to take a bite. Other devices in the market did not have the required sensors to track this motion, or had extra features which increased the size of the device and reduced its battery life. As mentioned in Chapter 3, we were able to create a wrist motion activity tracker that records this data at a good resolution, allows a high level of customization, has a sufficient battery life and has a size that is comfortable to wear for extended periods of time.

Future work for the device involves the addition of a time synchronization feature when the device is connected to a computer. This would store the local time of the user on the wrist motion activity tracker, and will allow the device to time stamp when it was stopped and started, allowing data analysis to be performed to detect periods of eating and non-eating.

In our wrist motion activity tracker, a large amount of power is utilized by the sensors which are always turned on. While we were working through this thesis, newer technology has allowed sensor manufacturers to improve the sensors they are producing. Some new sensors now come with microcontrollers in-built which can be programmed by the user. In our work, the sensors we used were operating based on an internal clock of 8 KHz. We would then read from the sensors at 15 Hz. Since we only need the sensors to be polled at 15 Hz, it should be possible to decrease the rate at which the sensors are operating. If the sensors could sleep between two readings, it would allow us to have a huge improvement in battery life as it would reduce the average current draw by the

sensors.

When we compared the different devices on the market, we noticed that the wrist band device MYO uses EMG sensors coupled with accelerometer and gyroscope data to increase the accuracy of detection. Thalmic Labs [37] claims that the EMG sensors help estimate the pose of a users hand with very high accuracy. We consider that this claim might be true, and we plan to incorporate these newer EMG sensors that are smaller in future work.

Creating a dataset of wrist motion data using this device is our next step forward. We plan to have 100 - 200 subjects wear the device for up to 2 weeks each to record their wrist movement data. This may require the production of 20 - 50 units of the device. This large data set will be used to improve the algorithms for detecting eating events and counting the bites during these events.

Appendices

Appendix A

C Code used to program the micrcontroller.

```c
/* Define functions */
#include <msp430f248.h>
#include "mpu6000.h"
#include "dataflash.h"

/* General Defines */
#define      LED            BIT0
#define      BTN            BIT4   /* Button on P1.4 */

/* SPI Pin defines */
#define      nSS            BIT0   /* Enable Pin for Sensor */
#define      SPI_SIMO       BIT1
#define      SPI_SOMI       BIT2
#define      SPI_CLK        BIT3
#define      mSS            BIT4   /* Enable Pin for Memory Chip */

/* SPI commands / vars */
#define      MPU_READ       0x80
/* PageSize for current Memory is set to 1024, but 6 * 170 = 1020*/
/* so we use this for all counters */
#define PAGESIZE 1020
/* UART Pins and vars */
#define RXPIN BIT5
#define TXPIN BIT4
#define RS232_ESC      27
#define ASCII0 0x30

/* Variables for Button Debouncing */
/* 1 Not waiting for Debounce Timer */
/* 0 Button was pressed very soon */
unsigned char debounce;

/* Variables for SPI */
unsigned char read = 0;
```

64

```
unsigned char PageAddress_H = 0;
unsigned char PageAddress_L = 0;

/* Variables for Main Program and to store Data */
/* ActionModes: 0 = LPM3, 1 = Switch between Blink / No Blink modes*/
/* 2 = Switch Blink Frequencies, 3 = SPI reading*/
unsigned char ActionMode = 0;
/* Stores data to output to Memory.*/
/* Size of Buffer and Block on Memory is PAGESIZE bytes by default */
signed char SensorData[PAGESIZE];
/* The current page number we are reading or writing */
unsigned int CurrentPage = 0;
/* Variable used for counters.*/
unsigned int ctr = 0;
/* 1 = Writing Sensor Data to Memory, 0 = Not Writing Data */
unsigned char WritingMode = 0;
/* 1 = Writing Sensor Data to Memory, 0 = Not Writing Data */
unsigned char SwitchOn = 0;
unsigned char FORCESTOP = 0;
void ONDANCE();
void OFFDANCE();

/* Timer for reading sensors */
 /* Blinking frequency / timer on startup 0x1000 is 1 second */
unsigned int BaseTime = 2184;
/* Flag is set if Timer interrupts an SPI operation */
unsigned char ReadingSensor = 0x00;

/* Variables to store Timer Counts to evaluate communication times */
unsigned int StartTime = 0;
unsigned int EndTime = 0;
unsigned int Time = 0;

/* Variables for logging to UART */
unsigned long ReadIndex = 0;
char PhoneViewStart[] = "START 2014-21-04 07:30:20";
char PhoneViewEnd[] = "END 2014-21-04 20:05:05";
unsigned char UART_Working = 0;
unsigned char UART_data[2];

/* Functions */

/* Sensor */
unsigned char Sensor_TXRX(unsigned char add, unsigned char val);
unsigned char _Sensor_write(unsigned char add, unsigned char val);
unsigned char _Sensor_read(unsigned char add);

/* General */
void JustDance();
void ReadPageNumberFromFlash();
void WritePageNumberToFlash();

/* Memory */
unsigned char MEM_TXRX(unsigned char data);
unsigned char Mem_ReadID();
```

```
void FlushMemory(void);
void Mem_WriteToBuffer();
void Mem_ReadFromBuffer();
void Mem_ReadFromMem();
void Mem_BufferToPage();

void Mem_ReadAllBinary();

/* UART Functions */

void ClearScreen();
void UART_SendValue(signed char);
void UART_SendChar(unsigned char);
void UART_SendIndex(unsigned long num);
void UART_SendTime(unsigned long index);
void UART_SendValue2(unsigned char num);

void UART_SendValue2(unsigned char num)
{
  unsigned char p = 0;
  p = (unsigned char) num / 10;
  UART_SendChar( p + ASCII0 );
  num = num - (p * 10);
  p = (unsigned char) num;
  UART_SendChar( p + ASCII0 );
}
```

```
/* Dataflash Commands. Source : http://playground.arduino.cc/Code/Dataflash */

#define FlashPageRead    0x52 // Main memory page read
#define FlashToBuf1Transfer   0x53 // Main memory page to buffer 1 transfer
#define Buf1Read              0xD1 // Buffer 1 read
#define FlashToBuf2Transfer  0x55 // Main memory page to buffer 2 transfer
#define Buf2Read       0x56 // Buffer 2 read
#define StatusReg      0x57 // Status register
#define AutoPageReWrBuf1 0x58 // Auto page rewrite through buffer 1
#define AutoPageReWrBuf2 0x59 // Auto page rewrite through buffer 2
#define FlashToBuf1Compare   0x60  // Main memory page to buffer 1 compare
#define FlashToBuf2Compare 0x61 // Main memory page to buffer 2 compare
#define ContArrayRead    0x68 // Continuous Array Read (Note : Only A/B-parts supported)
#define FlashProgBuf1    0x82 // Main memory page program through buffer 1
#define Buf1ToFlashWE       0x83 // Buffer 1 to main memory page program with built-in erase
#define Buf1Write     0x84 // Buffer 1 write
#define FlashProgBuf2    0x85 // Main memory page program through buffer 2
#define Buf2ToFlashWE       0x86 // Buffer 2 to main memory page program with built-in erase
#define Buf2Write     0x87 // Buffer 2 write
#define Buf1ToFlash         0x88 // Buffer 1 to main memory page program w/o built-in erase
#define Buf2ToFlash   0x89 // Buffer 2 to main memory page program w/o built-in erase
#define PageErase           0x81 // Page erase, added by Martin Thomas
```

```
/* Code from https://code.google.com/p/gentlenav/ */
#ifndef __MPU6000_H__
#define __MPU6000_H__


// MPU6000 registers
#define MPUREG_AUX_VDDIO 0x01
#define MPUREG_XG_OFFS_TC 0x00
#define MPUREG_YG_OFFS_TC 0x01
#define MPUREG_ZG_OFFS_TC 0x02
#define MPUREG_X_FINE_GAIN 0x03
#define MPUREG_Y_FINE_GAIN 0x04
#define MPUREG_Z_FINE_GAIN 0x05
#define MPUREG_XA_OFFS_H 0x06
#define MPUREG_XA_OFFS_L 0x07
#define MPUREG_YA_OFFS_H 0x08
#define MPUREG_YA_OFFS_L 0x09
#define MPUREG_ZA_OFFS_H 0x0A
#define MPUREG_ZA_OFFS_L 0x0B
#define MPUREG_PRODUCT_ID 0x0C
#define MPUREG_XG_OFFS_USRH 0x13
#define MPUREG_XG_OFFS_USRL 0x14
#define MPUREG_YG_OFFS_USRH 0x15
#define MPUREG_YG_OFFS_USRL 0x16
#define MPUREG_ZG_OFFS_USRH 0x17
#define MPUREG_ZG_OFFS_USRL 0x18
#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
#define MPUREG_BANK_SEL 0x6D
#define MPUREG_MEM_START_ADDR 0x6E
#define MPUREG_MEM_R_W 0x6F
#define MPUREG_DMP_CFG_1 0x70
#define MPUREG_DMP_CFG_2 0x71
```

```
#define MPUREG_FIFO_COUNTH 0x72
#define MPUREG_FIFO_COUNTL 0x73
#define MPUREG_FIFO_R_W 0x74
#define MPUREG_WHOAMI 0x75

// Configuration bits MPU6000
#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

// Register 55 - INT Pin / Bypass Enable Configuration (INT_PIN_CFG)
#define BIT_INT_LEVEL      0x80
#define BIT_INT_OPEN       0x40
#define BIT_LATCH_INT_EN   0x20
#define BIT_INT_RD_CLEAR   0x10
#define BIT_FSYNC_INT_LEVEL 0x08
#define BIT_FSYNC_INT_EN   0x04
#define BIT_I2C_BYPASS_EN 0x02
#define BIT_CLKOUT_EN      0x01

// Register 56 - Interrupt Enable (INT_ENABLE)
#define BIT_FF_EN          0x80
#define BIT_MOT_EN         0x40
#define BIT_ZMOT_EN        0x20
#define BIT_FIFO_OFLOW_EN 0x10
#define BIT_I2C_MST_INT_EN 0x08
#define BIT_DATA_RDY_EN    0x01

// Register 58 - Interrupt Status (INT_STATUS)
#define BIT_FF_INT         0x80
#define BIT_MOT_INT        0x40
```

```
#define BIT_ZMOT_INT      0x20
#define BIT_FIFO_OFLOW_INT 0x10
#define BIT_I2C_MST_INT    0x08
#define BIT_DATA_RDY_INT   0x01


// DMP output rate constants
#define MPU6000_200HZ 0 // default value
#define MPU6000_100HZ 1
#define MPU6000_66HZ 2
#define MPU6000_50HZ 3

#endif // __MPU6000_H__
```

```c
#include "main.h"

#define TIMETOWRITE 60
#define TIMETOREAD 60

#define PAGESTOWRITE 8189 // (TIMETOWRITE * 5.5)
#define PAGESTOREAD 8189 //(TIMETOREAD * 5.5)



/*
** Main Function. We setup the clocks
** and SPI, then leave the device into
** sleep mode to be activated by the Timer Interrupt
*/
void main(void)
{

  /* Turn off Watchdog Timer */
  WDTCTL = WDTPW+WDTHOLD;

  BCSCTL3 |= XCAP_3;         /* Set the capacitance of the external crystal to 12.4pf. */

   /* Initialize DCO */
  BCSCTL1 = CALBC1_1MHZ;                /* Set DCO to 1MHz */
  DCOCTL = CALDCO_1MHZ;                 /* Set DCO to 1MHz */
  __delay_cycles(200000);
  debounce = 1;
  // MCLK/3 for Flash Timing Generator (0x0040u)
  /* Flash clock select: 1 - MCLK Divided by 2. */
  FCTL2 = FWKEY + FSSEL0 + FN1;
  SVSCTL = 0x80;                        /* SVS to set at 2.8V */

  /* Initialize Port 1 For LED and Button */
  P1DIR |= LED;              /* Set LED as output */
  P1REN |= BTN;                 /* Enable Pullup / Pulldown on BTN pin */
  P1OUT |= BTN;           /* Turn LED OFF Till the BTN is pressed, Pull up BTN pin */
  P1OUT &= ~LED;
  P1IFG = 0;                /* Clear any interrupts on P1 */
  P1IE  |= BTN;                 /* Set BTN as intterupt (default P1IES sets low to high )*/
  P1IES |= BTN;                 /* Sensitive to (H->L) */

  /* Enable SPI */
  P5SEL |= (BIT1 + BIT2 + BIT3);    /* Peripheral function instead of I/O */
  /* SPI Polarity = 1, MSB First, Master*/
  /* 3 Pin Mode, Synchronous Comm. UCCKPH = 0 is ~SPIPhase*/
  UCB1CTL0 = UCCKPL | UCMSB | UCMST | UCMODE_0 | UCSYNC;
  UCB1CTL1 = UCSSEL_2 | UCSWRST;      /* Clock from SMCLK; hold SPI in reset */

  UCB1BR1 = 0x0;                         /* Upper byte of divider word */
  UCB1BR0 = 0x1;                         /* Clock = SMCLK / 10 = 100 KHz */

  UCB1CTL1 &= ~UCSWRST;             /* Remove SPI reset to enable it*/

  /* Configure Pins for SPI. These commands might not be needed */
```

```
    P5DIR |= nSS | mSS | SPI_SOMI | SPI_CLK; /* Output on the pins */
    P5OUT |= nSS | mSS | SPI_SOMI | SPI_CLK; /* Set pins to high */
    __delay_cycles(0x80);

    /* Read the Memory Device ID. Should be 1F */
    read = Mem_ReadID();

// Mem_ReadAllBinary();
    /* Initialize UART */
    P3DIR = TXPIN;
    P3OUT = TXPIN;
    P3SEL = TXPIN | RXPIN;

    /* Values from MSP430x24x Demo - USCI_A0, 115200 UART Echo ISR, DCO SMCLK */

    UCA0CTL1 |= UCSSEL_2;                  // CLK = SMCLK
    UCA0BR0 = 8;                           // 32kHz/9600 = 3.41
    UCA0BR1 = 0x00;
    UCA0MCTL = UCBRS2 + UCBRS0;            // Modulation UCBRSx = 3
    UCA0CTL1 &= ~UCSWRST;                  // **Initialize USCI state machine**
    IE2 |= UCA0RXIE;                       // Enable USCI_A0 RX interrupt
    /* End UART init */

    /* Disable I2C on the sensor */
    _Sensor_write(MPUREG_USER_CTRL, BIT_I2C_IF_DIS);

    /* Check the Sensor Device ID */
    read = _Sensor_read(MPUREG_WHOAMI);

    /* Trap uC if Sensor doesn't ID correctly */
    /* Check WHOAMI Hopefully it is 0x68. Otherwise freak out. */
    if(read != 0x68) JustDance();


    /* Sensor might be sleeping, read Register with Sleep Bit */
    /* reset the bit and write it back */
    read = _Sensor_read(MPUREG_PWR_MGMT_1);
    /* Reset the Sleep Bit to wake it up */
    _Sensor_write(MPUREG_PWR_MGMT_1, (read & ~BIT_SLEEP));

    /* Erase all data SensorData */
    for(ctr = 0; ctr < PAGESIZE; ctr++)
      SensorData[ctr] = 0;

    /* Reset Counter to 0 after previous for loop */
    ctr = 0;

    /*
    ** TimerA for Blinking, Button Debouncing, Long Press wait
    ** TACTL = Timer A Control
    ** TASSEL_1 => Set Timer source to ACLK (ACLK is from LFXT) | MC_2 = MODE => Continuous
    ** TACLR => Clear Timer | ID_3 Divide Clock by 8 = > 4096 = 0x1000
    ** This does not start the timer (Set mode to not 0, and have not zero value in TACCR0)).
    */
    TACTL = TACLR;
```

```
__delay_cycles(8254); /* Delay by 0.2 seconds DELAY CYCLES USES DCO @ 1MHZ*/
TACTL = TASSEL_1 | MC_2; /* Run Timer on ACLK */

  /* Other Housekeepigng */
__delay_cycles(8234);                  /* Delay 0.2s at 1Mhz to let clock settle */
__enable_interrupt();

OFFDANCE();

/** Main Program **/

while(1)
{
  __low_power_mode_3();

  if(ActionMode == 1)              /* Long Buttton Press from old Timer Code */
  {
    if(((WritingMode == 1) && ((P1IN & BTN) == 0)) || (FORCESTOP == 1))
    {
      /* Disable Sensor Reading */
      /* Set timer to disable */
      FORCESTOP = 0;
      TACCR0 = 0;
      TACCTL0 &= ~CCIE;
      OFFDANCE();
      WritingMode = 0;
      SwitchOn = 0;
      __enable_interrupt();
      __delay_cycles(600);
    }
    else if((WritingMode == 0) && ((P1IN & BTN) == 0))
    {
     /* Reset Memory count since device is off and LongPress */
      CurrentPage = 0;
      WritePageNumberToFlash();
      ctr = 0;
      JustDance();
      SwitchOn = 0;

    }
    else if (SwitchOn == 1)
    {
      TACCR0 += (BaseTime);
      TACCTL0 |= CCIE;
      ONDANCE();

      P1OUT &= ~LED;
      WritingMode = 1;
      SwitchOn = 0;
    }
    ActionMode = 0;
  }

  if(ActionMode == 2)              /* Short Button Press from old Timer Code */
  {
```

```
    if(WritingMode == 0)            /* Device is off and shortpress */
    {
    SwitchOn = 1;
    }
    ActionMode = 0;
  }

  if(ActionMode == 3)               /* Timer A Freq at 15hz */
  {
    ReadingSensor = 1;
    /* 2nd to Last data should be -128, Last Should be 0 */
    if(CurrentPage == PAGESTOWRITE && ctr > 1007)
    {
    SensorData[ctr++] = -128;
    SensorData[ctr++] = -128;
    SensorData[ctr++] = -128;
    SensorData[ctr++] = -128;
    SensorData[ctr++] = -128;
    SensorData[ctr++] = -128;
    JustDance();
    }
    else
    {
    /* Read X Acc */
    SensorData[ctr++] = _Sensor_read(MPUREG_ACCEL_XOUT_H); // rand--;//
    /* Read Y Acc */
    SensorData[ctr++] = _Sensor_read(MPUREG_ACCEL_YOUT_H);
    /* Read Z Acc */
    SensorData[ctr++] = _Sensor_read(MPUREG_ACCEL_ZOUT_H);
    /* Read X Gyro */
    SensorData[ctr++] = _Sensor_read(MPUREG_GYRO_XOUT_H);
    /* Read Y Gyro */
    SensorData[ctr++] = _Sensor_read(MPUREG_GYRO_YOUT_H);
    /* Read Z Gyro */
    SensorData[ctr++] = _Sensor_read(MPUREG_GYRO_ZOUT_H);
    }


        if(ctr >= PAGESIZE) /* The Buffer is full */
        {
/* Disable interrupts b/c the timer might interrupt otherwise */
        __disable_interrupt();
        P1OUT |= LED;
        ReadPageNumberFromFlash();
        StartTime = TAR;
        /* Write the stored SensorData to the Memory chip Buffer */
        Mem_WriteToBuffer();
        /* Write the Buffer data to Page. Page Number is stored in Global CurrentPage */
        Mem_BufferToPage();
        CurrentPage++;      /* Increment current page for next write */
        WritePageNumberToFlash();
        ctr=0;              /* Reset ctr so its starts at 0 for next page */
        EndTime = TAR;
        Time = EndTime - StartTime;
        /* Erase SensorData. Using new ctr variable b/c ctr is used in loop above */
```

```
            for(int actr = 0; actr < PAGESIZE; actr++)
                SensorData[actr] = 0;
            ReadingSensor = 0; /* Flag set to 0 b/c we're done with communication */
            ActionMode = 0;
            P1OUT &= ~LED;
            /* Enable Interrupts for the timer to start logging data again */
            __enable_interrupt();


        }

      ReadingSensor = 0; /* Flag set to 0 b/c we're done with communication */
      ActionMode = 0;
    }
  }

}

/*********************/
/* Code for MPU 6000 */
/*********************/

unsigned char _Sensor_write(unsigned char add, unsigned char val)
{
  P5OUT |= mSS;                      /* Deselect Select Memory as SPI slave */
  P5OUT &= ~nSS;                     /* Select Sensor as SPI Slave */
  unsigned char RXCHAR = 0x00;

  RXCHAR = Sensor_TXRX(add, val);

  P5OUT |= nSS;                      /* Deselect Sensor as SPI slave */
  return RXCHAR;
}

unsigned char _Sensor_read(unsigned char add)
{

  unsigned char RXCHAR = 0x00;
  P5OUT |= mSS;                      /* Deselect Select Memory as SPI slave */
  P5OUT &= ~nSS;                     /* Select Sensor as SPI Slave */

  RXCHAR = Sensor_TXRX(add | MPU_READ, 0x00);

  P5OUT |= nSS;                      /* Deselect Sensor as SPI slave */
  return RXCHAR;
}

unsigned char Sensor_TXRX(unsigned char add, unsigned char val)
{
  unsigned char RXCHAR = 0x00;

  while (!(UC1IFG & UCB1TXIFG));     /* Wait for TXBUF to be empty */

  UCB1TXBUF = add;                   /* Send Address of Register */
  /* Wait for TXBUF to be empty (TXBUF data moves to the shift register) */
  while(!(UC1IFG & UCB1TXIFG));
```

```c
  while(!(UC1IFG & UCB1RXIFG));          /* Wait for RXBUF to be full */
  /* Wait for SPI to complete communication. Shouldn't be needed */
 // while(UCB1STAT & UCBUSY);
  RXCHAR = UCB1RXBUF;                     /* Read what is RX to clear buffer / flags*/


  UCB1TXBUF = val;                        /* Write the val */
  /* Wait for TXBUF to be empty (TXBUF data moves to the shift register) */
  while(!(UC1IFG & UCB1TXIFG));
  while(!(UC1IFG & UCB1RXIFG));          /* Wait for RXBUF to be full */

  RXCHAR = UCB1RXBUF;                     /* Read what is RX to clear buffer / flags*/

  return RXCHAR;
}


/**********************/
/* Code for Dataflash */
/**********************/
unsigned char MEM_TXRX(unsigned char data)
{
  unsigned char RXCHAR = 0x00;
  while (!(UC1IFG & UCB1TXIFG));          /* Wait for TXBUF to be empty */
  /* Wait for SPI to complete communication. Shouldn't be needed */
// while(UCB1STAT & UCBUSY);
  UCB1TXBUF = data;                       /* Send Address of Register */
  /* Wait for TXBUF to be empty (TXBUF data moves to the shift register) */
  while(!(UC1IFG & UCB1TXIFG));
  while(!(UC1IFG & UCB1RXIFG));          /* Wait for RXBUF to be full */
  /* Wait for SPI to complete communication. Shouldn't be needed */
// while(UCB1STAT & UCBUSY);
  RXCHAR = UCB1RXBUF;                     /* Read what is RX to clear buffer / flags*/
// while(UCB1STAT & UCBUSY);
  return RXCHAR;

}

unsigned char Mem_ReadID()
{
  unsigned char RXCHAR = 0x00;
  P5OUT |= nSS;                           /* Deselect Sensor as SPI slave */
  P5OUT |= mSS;                           /* Deselect memory as SPI slave */
  P5OUT &= ~mSS;                          /* Select Memory as SPI Slave */

  MEM_TXRX(0x9F);                         /* Send the Buffer OpCode to the Memory */
  /* Write 3 bytes of address. We starts at byte 0, so this is always 0 */
  RXCHAR = MEM_TXRX(0x00);
  RXCHAR = RXCHAR;

  P5OUT |= mSS;                           /* Deselect memory as SPI slave */
  return RXCHAR;
}

void Mem_WriteToBuffer()
```

```
{
  P5OUT |= nSS;                      /* Deselect Sensor as SPI slave */
  P5OUT |= mSS;                      /* Deselect memory as SPI slave */
  P5OUT &= ~mSS;                     /* Select Memory as SPI Slave */

  MEM_TXRX(Buf1Write);               /* Send the Buffer OpCode to the Memory */
  /* Write 3 bytes of address. We starts at byte 0, so this is always 0 */
  MEM_TXRX(0x00);
  MEM_TXRX(0x00);
  MEM_TXRX(0x00);

  for(ctr = 0; ctr < PAGESIZE; ctr++)
    MEM_TXRX(SensorData[ctr]);

   /* Wait for SPI to complete communication. Shouldn't be needed */
  while(UCB1STAT & UCBUSY);
  P5OUT |= mSS;                      /* Deselect memory as SPI slave */
  __delay_cycles(100);
}

void Mem_ReadFromBuffer()
{
  P5OUT |= nSS;                      /* Deselect Sensor as SPI slave */
  P5OUT |= mSS;                      /* Deselect memory as SPI slave */
  P5OUT &= ~mSS;                     /* Select Memory as SPI Slave */

  MEM_TXRX(Buf1Read);                /* Send the Buffer OpCode to the Memory */
  MEM_TXRX(0x00);                    /* Don't Care Bytes */
   /* Upper and lower Byter. We starts at byte 0, so this is always 0 */
  MEM_TXRX(0x00);
  MEM_TXRX(0x00);                    /* Lower Byte */

  for(ctr = 0; ctr < PAGESIZE; ctr++)
      SensorData[ctr] = MEM_TXRX(0x00);
/* Wait for SPI to complete communication. Shouldn't be needed */
  while(UCB1STAT & UCBUSY);
  P5OUT |= mSS;                      /* Deselect memory as SPI slave */

}

void Mem_BufferToPage()
{
  PageAddress_H = (unsigned char) (((CurrentPage<<2) & 0xFF00)>>8);
  PageAddress_L = (unsigned char) (((CurrentPage<<2) & 0xFF));

  P5OUT |= nSS;                      /* Deselect Sensor as SPI slave */
  P5OUT |= mSS;                      /* Deselect memory as SPI slave */
  P5OUT &= ~mSS;                     /* Select Memory as SPI Slave */

  MEM_TXRX(Buf1ToFlashWE);               /* Send the Buffer OpCode to the Memory */
  MEM_TXRX(PageAddress_H);                /* Don't Care Bytes */
  /* Upper and lower Byter. We starts at byte 0, so this is always 0 */
  MEM_TXRX(PageAddress_L);
  MEM_TXRX(0x00);                    /* Lower Byte */
```

```c
  P5OUT |= mSS;                        /* Deselect memory as SPI slave */
}

void Mem_ReadFromMem(unsigned int PageToRead)
{
  PageAddress_H = (unsigned char) (((PageToRead<<2) & 0xFF00)>>8);
  PageAddress_L = (unsigned char) (((PageToRead<<2) & 0xFF));

  P5OUT |= nSS;                        /* Deselect Sensor as SPI slave */
  P5OUT |= mSS;                        /* Deselect memory as SPI slave */
  P5OUT &= ~mSS;                       /* Select Memory as SPI Slave */

  MEM_TXRX(0xD2);                   /* Send the Buffer OpCode to the Memory */
  MEM_TXRX(PageAddress_H);                  /* Don't Care Bytes */
  /* Upper and lower Byter. We starts at byte 0, so this is always 0 */
  MEM_TXRX(PageAddress_L);
  MEM_TXRX(0x00);                      /* Lower Byte and Add*/

  MEM_TXRX(0x00);                    /* Dummy Byte */
  MEM_TXRX(0x00);                    /* Dummy Byte */
  MEM_TXRX(0x00);                    /* Dummy Byte */
  MEM_TXRX(0x00);                    /* Dummy Byte */

  for(ctr = 0; ctr < PAGESIZE; ctr++)
      SensorData[ctr] = MEM_TXRX(0x00);
  /* Wait for SPI to complete communication. Shouldn't be needed */
  while(UCB1STAT & UCBUSY);
  P5OUT |= mSS;                        /* Deselect memory as SPI slave */
}


/* Old function used when READMEM */
void Mem_ReadAllBinary()
{
    for(int i = 0; i < CurrentPage ; i++)
    {

      for(ctr = 0; ctr < PAGESIZE; ctr++)
          SensorData[ctr] = 0;

      Mem_ReadFromMem(i);

      for(ctr = 0; ctr < 170; ctr++)
        for(int ctr2 = 0; ctr2 < 6; ctr2++)
      UART_SendChar(SensorData[(ctr*6)+ctr2]);
    }

    while(UCA0STAT & UCBUSY);
}



void UART_SendChar(unsigned char data)
{
  while(!(IFG2 & UCA0TXIFG));
```

```
  UCA0TXBUF = data;
}




/* Interrupt for TimerA Channel 0, runs short periods*/
#pragma vector = TIMERA0_VECTOR
__interrupt void TA0V_ISR(void)
{
  __disable_interrupt();
  TACCR0 += (BaseTime); /* Next Interrupt at given time */
  BaseTime ^= 0x01;          /* Flip between 2184 and 2185 to average at 2184.5 */

  ActionMode = 3;

  if(ReadingSensor == 1)
   JustDance();

  if( (SVSCTL & SVSFG) == SVSFG)
  {
    ActionMode = 1;
    FORCESTOP = 1;
  }
  else
  {
    FORCESTOP = 0;
  }
  __low_power_mode_off_on_exit();
  P1IFG = 0;                 /* Changing P1 can change P1IFG */
  __enable_interrupt();
}



/* Interrupt for Port 1 Button */
#pragma vector = PORT1_VECTOR
__interrupt void P1IV_ISR(void)
{
__disable_interrupt();

  /* Temporarily disable the interrupt (This is enabled after timer compares) */
  P1IE &= ~BTN;
  switch(P1IFG)
  {
  case BTN:
        if(debounce == 1)
        {
            /* Set debounce to 0 to indicate that timer needs*/
            /* to overflow before button can be read again */
            debounce = 0;
            TACCR1 = TAR + 0x1800; /* Timer 1 for half second later */
            TACCTL1 = CCIE;        /* Enable Interrupt */
            TACCR2 = TAR+0x9000;   /* Timer 2 for long press */
            TACCTL2 = CCIE;        /* Enable Timer 2 interrupt */
```

```
        }
        break;

  default: break;
  }

  P1IFG = 0;                /* Clear any interrupts on P1 */
  P1IE  |= BTN;             /* Set BTN as intterupt (default P1IES sets low to high )*/
  P1IES |= BTN;             /* Sensitive to (H->L) */
  P1IFG = 0;                /* Clear any interrupts on P1 */
  __enable_interrupt();
}

/* Interrupt for TimerA Channel 1 */
#pragma vector = TIMERA1_VECTOR
__interrupt void TAIV_ISR(void)
{
__disable_interrupt();
switch(TAIV)
  {
  /* half a second has passed, so process the button command */
  case TAIV_TACCR1:
        TACCR1 = 0;
        TACCTL1 &= ~CCIE;
        /* half Seconds later, so we can say button has been debounced */
        debounce = 1;
        ActionMode = 2;          /* Action - Short Button Press */
        __low_power_mode_off_on_exit();

        break;
  case TAIV_TACCR2:
        /* 3 Seconds later after button press. */
        /* Turn Channel 1 to off. */

        TACCTL2 &= ~CCIE;
        TACCR2 = 0;
        ActionMode = 1;          /* Action = Long Button Press */

        debounce = 1; /* half Seconds later, so we can say button has been debounced */
        __low_power_mode_off_on_exit();
          break;
  default: break;
  }

  P1IFG = 0;                /* Clear any interrupts on P1 */
  P1IE  |= BTN;             /* Set BTN as intterupt (default P1IES sets low to high )*/
  P1IES |= BTN;             /* Sensitive to (H->L) */
  P1IFG = 0;                /* Clear any interrupts on P1 */
}

void JustDance()
{
 /* Turn LED off if the WRONG WHO_AM_I response is received */
 P1OUT &= ~LED;
     for(int j = 0; j<0x05; j++) for(int i = 0; i<0xFFFF; i++); /* Long LED off delay */
```

```
        P1OUT ^= LED;
        for(int j = 0; j<0x05; j++) for(int i = 0; i<0xFFFF; i++); /* Long LED on */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Short LED off */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Short LED on */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Short LED off */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Short LED on */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Short LED off */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Short LED on */
        P1OUT ^= LED;
        for(int j = 0; j<0x05; j++) for(int i = 0; i<0xFFFF; i++); /* Long LED on delay */
//      P1OUT ^= LED;
  P1IFG = 0;                 /* Clear any interrupts on P1 */
  P1IE  |= BTN;              /* Set BTN as intterupt (default P1IES sets low to high )*/
  P1IES |= BTN;              /* Sensitive to (H->L) */
  P1IFG = 0;                 /* Clear any interrupts on P1 */
}


// USCI A0/B0 Receive ISR
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
__disable_interrupt();
ReadPageNumberFromFlash();
//UC1IE &= ~UCA1RXIE;                    // Disable RX interrupt
UART_data[0] = UART_data[1];
UART_data[1] = UCA0RXBUF;

if(UART_data[0] == 's' && UART_data[1] == 'd')
{
  UART_data[0] = 0x00;
  UART_data[1] = 0x00;
  Mem_ReadAllBinary();
}
__enable_interrupt();
//UC1IE |= UCA1RXIE;                     // Enable RX interrupt
}

void ReadPageNumberFromFlash()
{
  char *Flash_ptr;                       // Flash pointer
  unsigned char LSB,MSB;

  Flash_ptr = (char *)0x1040;        // Initialize Flash segment C ptr
  MSB = *Flash_ptr++;             // Get Current Page number from the Flash location 0x1040
  LSB = *Flash_ptr;
  CurrentPage = (((unsigned int)MSB)<<8)+(unsigned int)LSB;
  __no_operation();                      // SET BREAKPOINT HERE
}
```

```
void WritePageNumberToFlash()
{
  char *Flash_ptr;                      // Flash pointer
  unsigned char LSB,MSB;
  Flash_ptr = (char *)0x1040;           // Initialize Flash segment C ptr

  FCTL3 = FWKEY;                        // Clear Lock bit
  FCTL1 = FWKEY + ERASE;                // Set Erase bit
  *Flash_ptr = 0;                       // Dummy write to erase Flash seg D
  FCTL1 = FWKEY + WRT;                  // Set WRT bit for write operation

  MSB = (unsigned char)((CurrentPage & 0xFF00u)>>8);
  LSB = (unsigned char)(CurrentPage & 0xFF);
  *Flash_ptr++ = MSB;
  *Flash_ptr = LSB;

  __no_operation();                     // SET BREAKPOINT HERE

  FCTL1 = FWKEY;                        // Clear WRT bit
  FCTL3 = FWKEY + LOCK;                 // Set LOCK bit
}


void OFFDANCE()
{
        P1OUT &= ~LED;
        for(int j = 0; j<0x05; j++) for(int i = 0; i<0xFFFF; i++); /* Long LED off delay */
        P1OUT ^= LED;
        for(int j = 0; j<0x05; j++) for(int i = 0; i<0xFFFF; i++); /* Long LED on */
        P1OUT ^= LED;
        for(int j = 0; j<0x05; j++) for(int i = 0; i<0xFFFF; i++); /* Long LED off */
        P1OUT ^= LED;
        for(int j = 0; j<0x05; j++) for(int i = 0; i<0xFFFF; i++); /* Long LED on */
        P1OUT ^= LED;
        for(int j = 0; j<0x05; j++) for(int i = 0; i<0xFFFF; i++); /* Long LED off */
}


void ONDANCE()
{
        P1OUT &= ~LED;
        for(int i = 0; i<0x8FFF; i++);     /* Long LED off delay */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Long LED on */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Long LED off */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Long LED on */
        P1OUT ^= LED;
        for(int i = 0; i<0x8FFF; i++);     /* Long LED off */
}
```

Bibliography

[1] Y. Dong, J. Scisco, M. Wilson, E. Muth, and A. Hoover, "Detecting periods of eating during free-living by tracking wrist motion," *Biomedical and Health Informatics, IEEE Journal of*, vol. 18, no. 4, pp. 1253–1260, July 2014.

[2] Y. Dong, A. Hoover, J. Scisco, and E. Muth, "A new method for measuring meal intake in humans via automated wrist motion tracking," *Applied psychophysiology and biofeedback*, vol. 37, no. 3, pp. 205–215, 2012.

[3] F. Inc., "Fitbit Official Website for Activity Trackers," Last Accessed: October 2014.

[4] Jawbone, "UP24 Fitness Tracker," Last Accessed: October 2014.

[5] Apple, "iPhone 4 - Technical Specifications," Last Accessed: October 2014.

[6] T. Labs, "Recap of Myo Hangout," Last Accessed: October 2014.

[7] Amazon, "Amazon: Samsung Gear 2 Product Description," Last Accessed: October 2014.

[8] U. C. P. S. Commision, "Fitbit Recalls Force Activity-Tracking Wristband," Last Accessed: October 2014.

[9] S. Electronics, "Sparkfun Tutorials: How a Gyroscope Works," Last Accessed: October 2014.

[10] ——, "3-Axis Gyro/Accelerometer IC - MPU-6050," Last Accessed: October 2014.

[11] M. Drennan, "An assessment of linear wrist motion during the taking of a bite of food," Master's thesis, Clemson University, 2010.

[12] S. Electronics, "Triple Axis Accelerometer and Gyro Breakout - MPU-6050," Last Accessed: October 2014.

[13] B. P. Lo, S. Thiemjarus, R. King, and G.-Z. Yang, "Body sensor network–a wireless sensor platform for pervasive healthcare monitoring," pp. 077–080, 2005.

[14] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*. IEEE, 2005, pp. 364–369.

[15] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh, "Deploying a wireless sensor network on an active volcano," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 18–25, 2006.

[16] T. Instruments, "Msp430 ultra-low-power microcontrollers," *MSP Product Brochure*, 2009.

[17] S. Research, "Shimmer Sensing," Last Accessed: October 2014.

[18] T. Instruments, "Mixed signal microcontroller," *MSP430F248 — MSP430F2x/4x — Ultra-low Power — Description and parametrics*, 2009.

[19] Digikey, "Digikey Electronics - Electronics Component Distributer," Last Accessed: October 2014.

[20] C. Burnett, "SPI Three Slaves," Last Accessed: October 2014.

[21] T. Instruments, "MSP430 64-Pin Target board," Last Accessed: October 2014.

[22] ——, "MSP430 Flash Emulation Tool," Last Accessed: October 2014.

[23] S. Electronics, "https://www.sparkfun.com/products/650," Last Accessed: October 2014.

[24] F. T. D. I. Ltd, "Ft232r usb uart ic," *FT232R USB UART IC*, Last Accessed: October 2014.

[25] O. Enclsoures, "Tactile multi-colored remote control enclosures," Last Accessed: October 2014.

[26] Z. Huang, "An assessment of the accuracy of an automated bite counting method in a cafeteria setting," Ph.D. dissertation, Clemson University, 2013.

[27] I. Buchmann, "Advantages and Limitations of Batteries," Last Accessed: October 2014.

[28] F. Neitzel and J. Klonowski, "Mobile 3d mapping with a low-cost uav system," *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci*, vol. 38, pp. 1–6, 2011.

[29] A. H. Kioumars and L. Tang, "Atmega and xbee-based wireless sensing," in *Automation, Robotics and Applications (ICARA), 2011 5th International Conference on*. IEEE, 2011, pp. 351–356.

[30] S. Electronics, "LiPo Charger Basic - Micro-USB," Last Accessed: October 2014.

[31] ——, "Momentary Pushbutton Switch - 12mm Square," Last Accessed: October 2014.

[32] *AMP Mini CT Connector 1.5mm PITCH*, TE Connectivity, 3 2011, rev. C3.

[33] *MPU-6000 and MPU-6050 Product Specification*, Invensense Inc., 8 2013, rev. 3.4.

[34] J. L. R. Concha, "A study of time-based features and regularity of manipulation to improve the detection of eating activity periods during free living," Ph.D. dissertation, Clemson University, 2014.

[35] S. Electronics, "SparkFun 10-25-13 Product Showcase: Sticky Situation," Last Accessed: October 2014.

[36] ——, "Reflow Skillet," Last Accessed: October 2014.

[37] T. Labs, "Gesture Control Armband," Last Accessed: October 2014.