UNIVERSITY OF TECHNOLOGY, SYDNEY

Master Thesis

The Firmware Development of a Portable Inertial Measurement Unit (IMU)

Faculty of Engineering and Information Technology
School of Electrical, Mechanical and Mechatronic Systems

Student: Xiwei Cui

Supervisers: Dr. Steven Su
Prof. Hung Nguyen

November 3, 2014

CERTIFICATE OF ORIGINAL AUTHORSHIP

I certify that the work in this thesis has not been submitted for a degree previously nor submitted as part of requirements for a degree.

I also certify that the thesis is all written by me. Any help that I have received in my research work has been acknowledged. Finally, I certify that all reference sources and literature used in the thesis are indicated.

Signature of Student:

Date:

ACKNOWLEDGEMENT

First of all, I would like to thank my supervisors Dr. Steven Su and Prof. Hung Nguyen for giving me opportunity to pursue master study in this project at University of Technology, Sydney.

Secondly, I particularly appreciate Dr. Steven Su for helping me overcome several difficulties in my research. He gives me guidance and assistance immediately when I encounter obstacles in designing software and understanding TI hardware. He also gives me strong supports for data operation when I encounter confused problems. I greatly appreciate to his huge patience and teaching.

Thirdly, I thank Prof. Jan Szymanski who gives me helpful assistance and advice in software debugging and hardware study. His unselfish assistance saves much time of my research. His self-made simulator solves debugging problems and helps me a lot.

Fourthly, I thank my dear colleagues Mitchell Yuwono, Ahmadreza Argha, Tao Zhang and Lin Ye for giving me useful information and timely assistance of my research. I am very appreciated and proud to work and study with all of you.

Finally, I would like to thank my family. Without their support, it is impossible for me to cross the ocean and reach this beautiful far away country to continue my study. Without their support, it is unlikely for me to finish my research on time. Without their support, it is unrealistic for me to meet my outstanding supervisor Dr. Steven Su and enthusiastic colleagues. I love my family more than words can describe. Many thanks to everyone help me and support me in my two years' research.

ABSTRACT

In recent years, patients' monitoring during their rehabilitation procedure has become an active research area in medical care as the anticipated research outcomes have great potential to save huge amount of funds and the time/efforts of medical professionals. In particular, portable motion sensors (e.g., micro Inertial Measurement Unit, μ-IMU) which can provide posture and acceleration information of patients during rehabilitation exercise, have already been applied in some health rehabilitation centres around the world to avoid falling injuries. Although there are diversity types of μ-IMU, two commonly encountered problems have not been solved completely. Firstly, similar with most portable sensors, the accuracy of the μ-IMUs is low due to their size and weight limitations. To improve their accuracy, it is urgently needed to develop efficient algorithms to implement the modelling and calibration of μ-IMUs in real time settings. Secondly, these algorithms need to be unified.

This thesis attempts to partially address the above two problems for μ-IMUs. An IMU often has several groups of sensors to collect enough information. These sensors are defined as micro-electro-mechanical system (MEMS) which consists of accelerometer, gyroscope and magnetometer. This thesis will focus on the calibration of tri-axial accelerometers of the μ-IMU. Some classical "complex/expensive" algorithms and calibration devices are already available. However, due to various limitations of portable devices, we are aiming for developing more "efficient" algorithms and devices to improve the accuracy of μ-IMUs.

The specific target of this thesis is completing a partly finished μ-IMU prototype for motion monitoring during exercise. Specifically, it mainly focuses on the on-site calibration of accelerometers as well as its associated firmware development.

This thesis firstly builds a general software structure for various functions embedded in the μ-IMU system. Secondly, a new calibration method has been proposed and implemented to improve calibration accuracy. Thirdly, a comparison between auto-calibration and classical calibration has been carried out in terms of accuracy. Finally, the best solution for the calibration of accelerometers of the μ-IMU has been adopted, implemented and tested experimentally. In addition, the calibration methods proposed in this thesis could be applied to other similar wearable products.

Key Words: Calibration, IMU, Auto-calibration

TABLE OF CONTENTS

I. INTRODUCTION

1.1 Problem Statement

An Australian national report about patients who suffered stroke was published in 2012. This report pointed out that there were 420,271 people who were living with stroke and over 65% patients also suffered disability and were unassisted [40]. In the United States, the problem is serious as well. Doshi [41] reports that approximately 795,000 people suffer stroke every year and this cost a large amount of funds for monitoring their rehabilitation [40]. He [41] also points out that an alternative monitoring method through a portable device has been found to replace human monitoring [40]. The motivation of this research comes from an uncompleted medical device, which would be used in the patient monitoring system. The system as an Auxiliary Medical Devices could replace nurses to monitor patients' movement and return relevant information to doctors. Particularly, the system should include a prediction functionality which can generate a warning massage before the patient falls down. In recent years, continuous monitoring of patients' motion has become one of the most active research areas in the field of Body Sensor Network (BSN), which includes human activity recognition by attached sensors and joint angle measuring through light-weight-wearable devices [32]. The research has a great potential to improve efficiency of medical institutions through continuous monitoring of patients' daily activities without the need to visit medical and rehabilitation centres. In order to track patients' motion, commercially available micro-inertial-measurement-units (IMUs) can achieve the required functionality of capturing full-body human motion; this could provide full 3-D posture and location information to physicians and therapists [31]. In the past, IMUs are mostly used in aircraft for posture adjustment because of their low cost, low power consumption and high accuracy. The use range varies from an inertial navigation system in an airplane to satellites and submarines [37]. An embedded accelerometer, gyroscope and magnetometer play the main function role in the system, which is defined as a micro-electro-mechanical system (MEMS). Currently, low cost IMUs are considered as an ideal viable solution for patients' motion monitoring [33]. Badwal notes that an IMU could be used in monitoring daily activities of younger patients who suffered hip osteoarthritis [42]. His

method consists of fixing two IMU on a patient's body; one is on the leg and the other one is on the waist [42]. An IMU could also be attached to a patient with Alzheimer's disease who could not maintain postural stability [14]. Although there are a variety of applications for an IMU, all these applications have three main problems. Firstly, Bluetooth is the main communication method in these applications. The distance of the communication is short. Secondly, the IMUs are high cost but low efficiency. Thirdly, the algorithms of the IMU are not unified. This thesis will mainly discuss the second problem.

In the second problem, complex algorithms and additional electronic assistance components cause low efficiency and high cost. In particular, the central factor responsible for low efficiency comes from calibration in the MEMS. In most cases, the output of IMU's MEMS sensors (Accelerometer, gyroscope and magnetometer) is unfortunately incorrect because of the tolerances which come from the manufacturing process and temperature [36]. Thus, an initial calibration has been performed after the manufacturing period because this is the only way to guarantee credibility of delivered information but the process is very costly and slow [35]. Nevertheless, the factory calibration results are changed slightly because of additional assembly error and thermal stress during the soldering process. Some other errors, such as environmental error, could not be rejected through initial calibration in the factory. This is the main reason that causes offset in the sensors. If the IMU is used in devices such as an airbag control system which could produce a high value of acceleration, this kind of error can be ignored. On the other hand, if the IMU is adopted in a human tracking system or airplane, the drifts should be calibrated to obtain acceptable accuracy [37]. The sensors, therefore, have to be calibrated again to obtain enough accuracy before using because technical data which are reported by the manufacturers are not accurate enough for applications [2]. In the MEMS, three sensors are integrated in the system. An accelerometer which provides accelerations in three axes could measure posture when patients are moving. A gyroscope, however, collects attitude angles to assist the accelerometer. Moving direction is measured by a magnetometer which can indicate moving orientation and location information. Generally, a real calibration is calibrating all three sensors in MEMS. This thesis, however, would only focus on discussing accelerometer calibration.

Accelerometer calibration is a complete experimental process. It is based on the fact that the quadratic sum of acceleration that is measured by the tri-axial accelerometer is equal to $g = 9.8m/s^2$ in static conditions [2]. The experiments are only performed through measuring different random orientations [2]. However, a different algorithm and experimental method could achieve different result. Frosio points out a linear model for output and Newton's iteration method for a solution [2]. Glueck [34] states a same linear model while using a nonlinear parameter estimator algorithm for solution. Krohn discusses two different models which are an orthogonal axis model and a non-orthogonal axis model [35]. Experimental methods are also different in different calibration methods. For example, use external actuators, a vibration generator [38], for calibration or align the sensor with each measurement axis parallel and antiparallel to a maximum of already known reference acceleration such as gravity acceleration. Another method for calibration is adopting robotic actuation or an optical tracking system [39]. A high performance position sensor in combination with nonlinear Kalman filter theory is also a recommended calibration technique [34]. All these calibration method could achieve a high accuracy. However, they require costly sophisticated laboratory equipment assistance. In the paper [35], Albert Krohn reports an inexpensive but high efficiency auto calibration method that is referenced by this thesis. He points out that hundreds of accelerometer sensor could be integrated in a box and calibrated simultaneously just through putting the box in different orientations [35]. In Iuri Frosio's paper, he states a different algorithm in auto calibration which uses random orientations to measure acceleration [4]. Manuel Glueck explains another auto calibration method which adopts a non-linear model for statistical linearization instead of analytical linearization [36]. Therefore, an experimental method and algorithm could affect the calibration result simultaneously.

The motivation of this thesis is completing an unfinished medical device (IMU) which could predict patient's falling. In the unfinished medical device, non-calibrated MEMS and uncompleted software should be finished in this thesis. In order to complete calibration work, the current situation should be considered carefully. That is calibration method selection. The selection should have three steps. The first step is algorithm selection. The second one is experimental method selection. The final step is achievement method selection.

In the algorithm selection, model and parameter estimation method are the core procedure. Frosio prefers Newton's iteration method and linear model in "Autocalibration of MEMS Accelerometer", which is simple to understand and achieve [2]. Glueck adopts a different linear model but a more complex parameter estimation method which is a nonlinear recursive parameter estimator method with unscented transformation [34]. Glueck's algorithm has higher accuracy but difficult calculation. Krohn states a more convenient algorithm through a rotation matrix, which is inexpensive to achieve but has a lower accuracy [35]. After referencing these algorithms, a linear model and least squares method are finally selected in this thesis.

In the experimental method selection, although there are many differences between those methods that were introduced previously, all of the methods could be classified into auto calibration method and classical calibration method. The auto calibration method generally collects acceleration through different random orientations while the classical calibration method often has known reference orientations. Because of the classification, another necessary procedure of the thesis is comparing auto calibration and classical calibration to select the most appropriate method for the IMU.

In the achievement method selection, current applications of patient's IMU supervising device algorithm is directly written into IMUs, which can be regarded as online calibration. Although this kind of method has higher adaptability which means frequent calibration is not necessary, this leads to a lower efficiency because the calculation procedure in the IMU is much more complex. Therefore, this thesis adopts offline calibration, in which calculation procedure is operated on a PC while using calculation results in IMUs. The only disadvantage of the offline calibration is it requires frequent calibration because of drifting. However, its advantages are considerable. The offline calibration is suitable for mass production and higher operation speed, that is to say, it has higher efficiency.

1.2 Aims

The aim of this thesis is to complete an unfinished medical device (IMU) that is intended to predict patient's falling. However, some problems need to be solved to reach the final goal. Firstly, the software of the IMU is not fully functional. The current software could only support operation of the IMU while the calibration function has not been achieved. Therefore, complete software in the experiment panel to achieve the real application. Secondly, it is important to select an appropriate calibration method. In this problem, algorithm selection, experimental method selection and achievement method selection are three necessary steps. In this circumstance, the IMU is similar to a No scale ruler in that the output cannot be adopted directly. Currently, the IMU output is several digital numbers without any value. Finally, to improve its accuracy is also a key step. It is credible that a fine calibration process could improve the accuracy. Nevertheless, if the factors or parameters which are associated with results could be found, it is possible to reach a higher accuracy.

This completed IMU system could provide accurate accelerometer data under units of "g". At the same time, this system can be adjusted at any time according to its drift. The resulting system will be utilized in current commercial products. The results could be referenced in future research.

The aims of the thesis are as follows:

- Design and install software of the IMU. Due to there being no calibration software structure in the IMU, any further calibration work could not be processed. Therefore, building an experiment panel to establish a calibration mode is needed.
- Design an experiment in order to make a rough calibration. This rough calibration can be the classical calibration method. These calibration results would be used to build calibration mode.
- Design a new experiment to achieve fine calibration. This procedure is used to improve accuracy. After that, a comparison between auto-calibration and classical calibration could be discussed.
- Discuss factors that affect calibration accuracy. This aim tries to explore a possible better solution for calibration.

- Use calibrated results to improve the IMU system.

1.3 Contribution

This thesis presents the hardware and software of the IMU and the calibration process that is adopted in the IMU. At the same time, it also compares different methods and discusses what factors influence accuracy. And finally, complete and install the IMU software system. The contributions are listed:

- Firstly, software structure is improved in the thesis. Calibration mode is an experiment panel which could support every offline calibration method. Therefore, the calibration mode would play a foundation role in the thesis.
- Secondly, the fine calibration method is finished in this thesis. The fine calibration method could improve calibration accuracy. These calibration results could be used directly in further research and as evidence for comparison.
- Thirdly, a low cost but high accuracy accelerometer calibration method is executed.
- Fourthly, a comparison between the auto calibration method and the classical calibration method is accomplished. The thesis uses experiment data and analyses experiment result to prove that auto calibration is better than the classical calibration method.
- Fifthly, the IMU system is more improved.
- Finally, relationship between initial value and root mean square value has been discussed in this thesis. This relationship would be helpful to build a calibration simulation model to improve accuracy in future research.

1.4 Structure of Thesis

The thesis describes the process of achieving the IMU calibration. In the former part of the thesis, basic knowledge, relevant background and necessary information are introduced. In the latter part of the thesis, the calibration procedure, method, experiments and algorithm are described in detail. The thesis has 5 chapters. The chapters are listed below:

- Chapter 1 Introduction

  This chapter presents a macro view of the whole thesis. It mainly introduces background of this research, motivation, current application, relevant research information, purpose of this research and problems will be faced.

- Chapter 2 Literature Review

  This chapter has two subparts. One describes the microcontroller and another describes principle of calibration. This chapter is a description of basic and necessary knowledge for accomplishing the IMU system. In the microcontroller part, the hardware structure and working principle of the microcontroller is explained. Its operation through a PC is demonstrated as well. The principle of calibration part focuses on calibration knowledge which includes the 6-parameter-calibration principle and linear least squares method that will be used in the research. The purpose of this chapter is to provide a comprehensive understanding of the IMU calibrating procedure.

- Chapter 3 Electronic Components of the IMU

  This chapter introduces the IMU that need to be calibrated. All components and their features are illustrated in this chapter. The hardware design could be helpful to recognise its functionality and operation principles. A hardware structure of the IMU would be demonstrated carefully.

- Chapter 4 IMU Core Components and Protocol

  This chapter illustrates the three core IMU components which provide the most significant functions to the device. Especially, their communication protocols are explained in detail. Another communication protocol FIFO and Packet is introduced at the same time. This chapter mainly focuses on software structure particularly communication protocol inside and outside the IMU.

- Chapter 5 Calibration Method for Accelerometer

  In this chapter, algorithm, experiments details, accuracy improvement procedure and calibration mode achievement has been presented. This chapter explains the process that achieves the final goal of the thesis. At the end of the chapter, a final conclusion is listed.

- Chapter 6 Conclusion and Future Research

  This chapter gives the conclusion of the thesis and states future research.

II. LITERATURE REVIEW

2.1 Basis of Microcontroller

2.1.1 Microcontroller Introduction

Microcontrollers can be discovered in every electronic product. Any products that are utilized to measurement, calculation, controlling or handling information require a microcontroller inside. The products from which has a small size like a calculator, a keyboard, a mouse, a MP3, to a big size such as an automobile, an industry control system and an airplane all need the microcontroller as a central control unit. Generally, a microcontroller has powerful functions while it has simple architectures. Every time when people talk about microcontrollers, it is recognised that the first microcontroller which was really widespread used was the Intel 8048. Its successors, Intel 8051 and the 68HCxx series of microcontrollers from Motorola, were also quite popular in the past days. Nowadays, there are thousands of microcontroller manufactures in the world but the number of the most authority companies in this area is much fewer; ARM, AVR, Freescale Semiconductor, Holtek, Intel, Microchip, National Semiconductor, NXP Semiconductors, Renesas Technology, STMicroelectronics, SyncMOS Technology, PADAUK and TI are the most famous companies. Although these companies' designs and products are quite different, the basic structure and the knowledge are totally similar. This chapter will be focused on the basic knowledge of microcontrollers. The architecture of microcontrollers will be presented in detail as well as their operation principle. In the final part of this chapter, some notes and experiments of the TI microcontroller will be illustrated in order to understand the microcontroller deeply.
Basically, a microcontroller should at least contain those components:

- Central processing unit(CPU)
- Random Access Memory(RAM)
- Read Only Memory(ROM)
- Timers and Counters
- Interrupt Controls
- Input/output ports
- Analog to digital converters

- Digital analog converters

This chapter will follow this structure to introduce the microcontroller. An example of a basic architecture and block diagram of a microcontroller is shown in the figure 2-1.

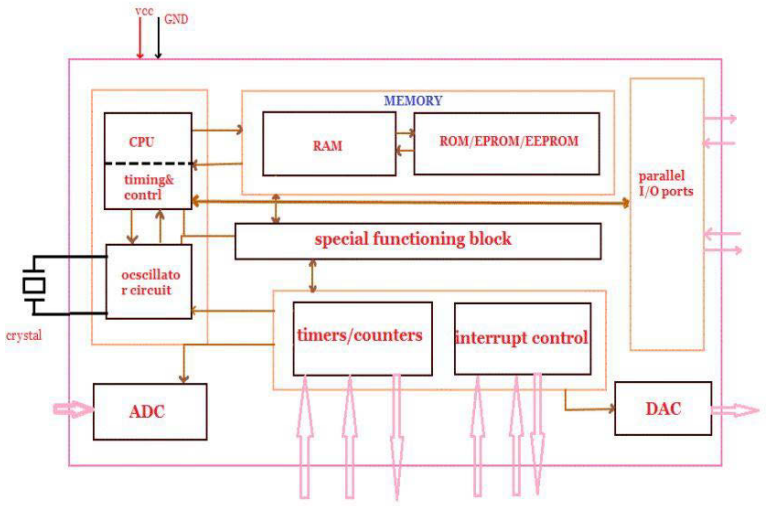


Figure 2-1 Basic Structure of Microcontroller

The structure of a microcontroller has been demonstrated clearly in the diagram. However, what is the cause of the microcontroller indispensable in our daily life? To better understand this, here provides an example of a water heater which is controlled by the microcontroller.

• Read the temperature periodically in every 5 seconds. Obtain the temperature which is an analog value that is digitized by the sensor; use a 4-bit data return to the microcontroller.

• Adjust heating period according to the temperature which is set up inside the microcontroller previously. If the water temperature is lower/higher, turn the heater on/off. Use a 1 bit bool value which is sent by the microcontroller to control its working.

• Display the current temperature value on a simple 3-digit numeric displayer (8+3 bits).

• Allow the user to adjust temperature thresholds (buttons; 4 bits)

• Be able to configure/upgrade the system over a serial interface.

This is a simple example of the microcontroller application in our life. There are thousands of more complex daily products consist of microcontrollers. Therefore, it is quite important to understand microcontrollers and their working principles.

2.1.2 CPU and its Architecture

When a machine obtains instructions from human, how does this machine understand this order? CPU is responsible for fetching the instruction and decoding it and finally executed. The CPU is the crucial part that connects every part of a microcontroller. A basic CPU architecture is illustrated in Figure 2-2. It consists of the data path which executes instructions and the control unit which basically tells the data path what to do. The data path consists of ALU (Arithmetic Logic Unit), Register File and SP (Stack Pointer).

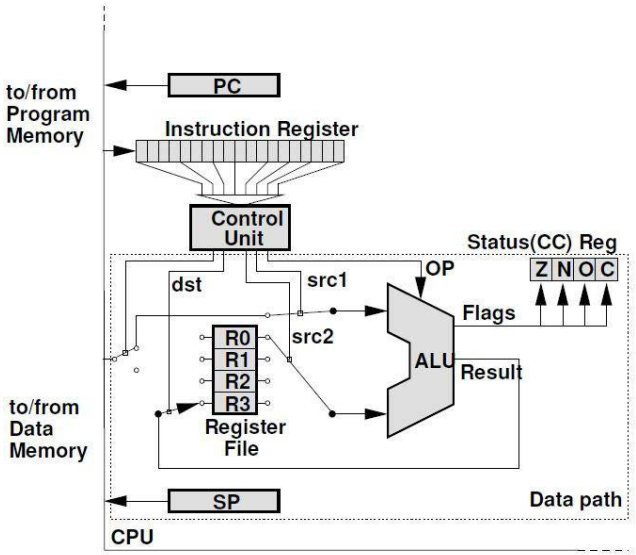


Figure 2-2 CPU Architecture

2.1.2.1 Control Unit

The Control Unit is utilized to decode instructions and configure the Data Path working. It performs fetching, decoding, managing execution and storing results. It is responsible for the Control Unit to determine which operation should be executed next and how to configure the data path accordingly. At this time, the Program Counter (PC) would store the address of the next instruction and the Instruction Register (IR) would load instructions into the Control Unit to decode. The Control Unit is a “brain in the brain” which carries out instructions in the software and directs the flow of data. It is also

responsible for configuring the Data Path working. The Data path configuration includes providing the appropriate inputs for the ALU, selecting the right ALU operation and writing the results to the correct register or memory.

The Control Unit is a hard-wired control unit which features a finite number of gates which can generate specific results based on the instructions. Although the Control Unit that uses this kind of approach can operate at high speed, the instructions have little flexibility and more complexity. That is to say, no matter how complex the instruction is, the Control Unit will execute in the same way. In order to improve its flexibility and decrease complexity, there are two additional architectures named Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) which are integrated inside the Control Unit. RISC operates a small and fixed code size with fewer instructions and fewer addressing modes. As a result, execution is very fast and the instruction set is rather simple. By contrast, the CISC often has a large and variable code size and offers many powerful instructions and addressing modes. Therefore, CISC has longer but more powerful executions.

2.1.2.2 Arithmetic Logic Unit

The core of the CPU is the arithmetic logic unit (ALU), which is used to perform all arithmetic calculation and logic operations. ALU is instructed by the Control Unit and performs all computations through different gates such as ADD, SUB, NOT, OR, AND, XOR. The ALU takes 2 inputs which are operand A and B coming from the register array and returns the result of the operation as its output which stores in register array. In addition, the ALU stores some information about the characteristics of the result in the status register:

Z (Zero): The result of the operation is zero.

N (Negative): The result of the operation is negative, that is, the most significant bit of the result is set 1.

O (Overflow): The operation produces an overflow, that is, there is a change of sign in a two complements operation.

C (Carry): The operation produces a carry.

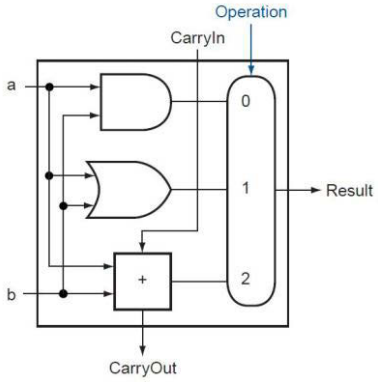


Figure 2-3 1-bit ALU

Here is a 1-bit ALU which could operate a 1-bit data per time. Some other more complex ALUs are all composed by the 1-bit ALU. Figure 2-4 shows a 32-bits ALU.

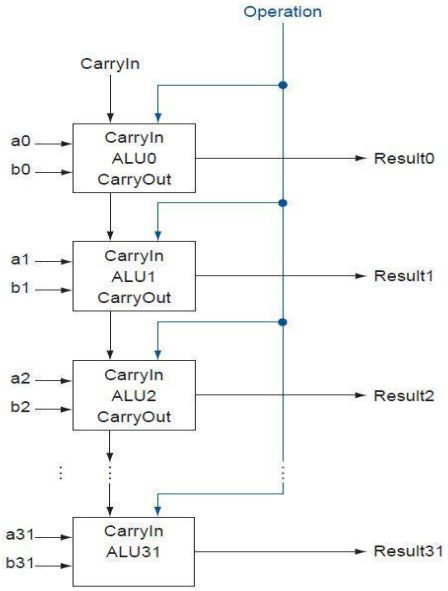


Figure 2-4 32-bits ALU

2.1.2.3 Register File

The register file contains the function registers of the CPU. It either consists of a set of general purpose registers, which are the source or destination of an operation, or some dedicated registers that has special functions. Dedicated registers are, for example, an accumulator, which is used for arithmetic or logic operations, an index register, which is used for addressing modes. Generally, the CPU could bring the operands to the ALU from the file, and it can store the operation's result back to the register file. Alternatively, operands and results can come from or be stored into the memory.

2.1.2.4 Stack Pointer

There is an area in memory unit called Stack. The Stack is used to store temporary storage of register values. It is usually maintained as a "last in first out" data structure. Microcontroller uses the stack during subroutine calls and interrupts. The Stack Pointer (SP) is a register of the micro-controller that is used to manage the Stack. It always points to the address of the last byte data saved on the Stack. It is operated with the commands PUSH that put something on the stack and POP that remove something from the stack. Figure 2-5 is an illustration of the Stack.

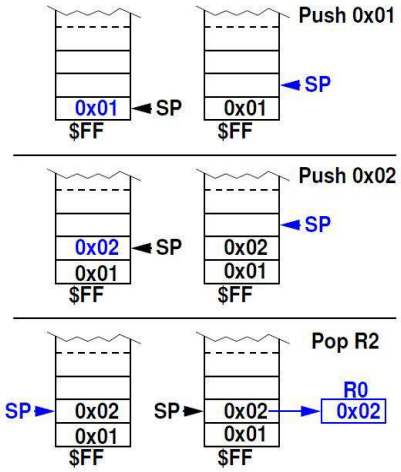


Figure 2-5 Diagram of Stack Pointer

2.1.3 Memory

Generally, a widely used memory unit is a Semiconductor Memory which includes volatile and non-volatile memory unit. Volatile memory unit includes SRAM and DRAM these are Static Random Access Memory and Dynamic Random Access Memory. The Non-volatile memory is ROM (Read Only Memory) and so on. Figure 2-6 is a diagram of memory in microcontroller.

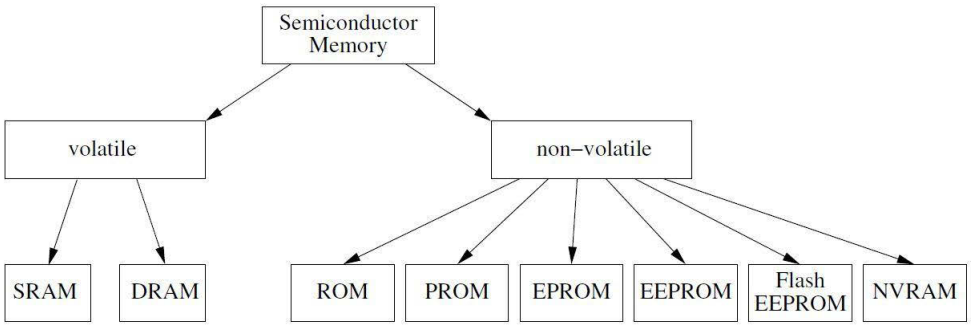


Figure 2-6 Types of Semiconductor Memory

2.1.3.1 SRAM

The first type of volatile memory to be widely used was Static Random Access Memory (SRAM). A SRAM chip is consisted by an array of cells and each cell could store one bit of information. A so-called flip-flop is used that consists of six transistors to store one bit of information. The internal structure of such a cell is illustrated in figure 2-7. To get a useful memory, many such cells are composed in a matrix as showed in Figure 4-8. All D_{out} lines are connected together.

A short between GND and VCC might occur if all cells would drive their outputs simultaneously, which would most likely destroy the chip. Therefore, the CS line is used to select one cell in the matrix and to set all other cells into their high resistance state. To drive one cell and hence access one particular bit, SRAMs need some extra logic to facilitate such addressing. A matrix of memory cells in an SRAM is illustrated in figure 8. As showed in Figure 8, a particular memory cell is addressed when both its CS signals associated row and column are pulled high. The purpose is to save address lines. If we were to drive each cell with an individual line, a 16Kx1 RAM (16 K bits), for example, would amazingly require 16384 lines. Thus, using the matrix layout with one and-gate per cell is efficient.

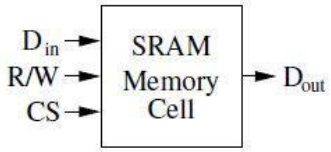


Figure 2-7 A cell of RAM

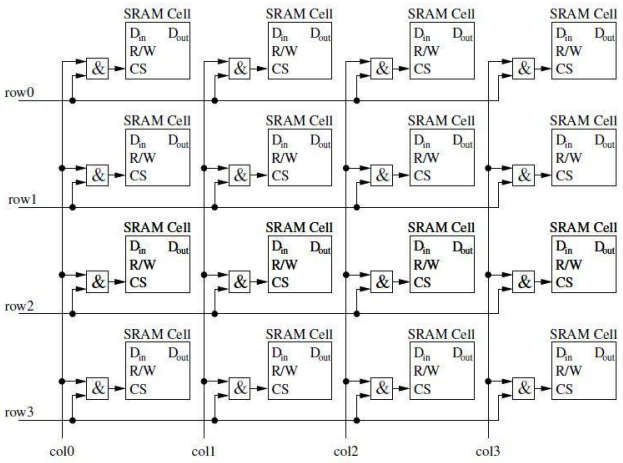


Figure 2-8 Matrix of Memory Cells in One SRAM

2.1.3.2 DRAM

SRAM usually requires six transistors to store one single bit of information. Therefore, the more transistors per cell are required, the larger silicon space will be. However, it is possible to reduce the number of components that are needed while obtains twice more the storage capacity. That is what was achieved by the Dynamic Random Access Memory – DRAM. The number of transistors that are needed per bit of information are just down to one. As a consequence, a DRAM has much larger storage capacity than an SRAM.

The secret of the DRAM is that it combines transistors and capacitor as its store unit. One bit of information is stored in a capacitor instead of using a lot of transistors to build flip-flops and the DRAM utilizes a transistor to select reading or writing operation. However, there is no perfect store unit in the world. Because of the flow of minimal currents through insulators on the chip, the capacitor would lose its charge though it is not accessed. The voltage will turn from high to low actually. Since these capacitors are rather small, the capacity is small as well. This leads to the fact that after loading the capacitor, the charge will unavoidably decrease. After some time, the information will lose with the charge.

2.1.3.3 ROM

ROM (Read Only Memory) is a typical non-volatile memory, which retains their content even when power is off. However, the problem is ROM is usually slow and complex to write. It always cost a long period and more money to write or re-write context. It is impossible to simply write data to a ROM directly. If you want to write something to ROMs, you have to give the data to the chip manufacturer who could use special methods to write the data in the ROM.

A common type of ROM is the so-called Mask-ROM (MROM) which has a similar structure with the RAM that contains a matrix of memory cells. The information to be stored in the MROM by fixed connections between rows and columns.

2.1.3.4 PROM

PROM is another kind of non-volatile memory. This memory is a so-called one time programmable memory. It contains matrices of memory cells like the RAM and each containing a silicon fuse. Initially, every fuse is intact and each cell stays at logical 1.

By selecting a cell and applying a short but high current pulse, the cell's fuse would be destroyed and therefore programming a logical 0 into the selected cell. After this programming, the logic inside PROM is fixed and never be changed. It is of course not suitable for development while it is suitable for middle range mass production.

2.1.3.5 EPROM

EPROM is a first Erasable Programmable Read Only Memory. It solves the problem that a ROM could only be read but be written. EPROM could allow a user to re-write the data into a memory unit. The memory is stored in field effect transistors (FETs) or in one of their pins named floating gate because of completely insulated from the rest of the circuit. However, by applying an appropriately high voltage, it is possible to charge the floating gate by a physical process called avalanche injection. Therefore, instead of burning fuses, electrons could be injected into the floating gate. Although there is minimal leakage current flow through the insulators, it is a long period that more than ten years for these floating gate to become discharged. However, there is a method could speed up this process by exposing the silicon chip to ultraviolet light. After about 30 minutes, an ultraviolet light will have discharged the floating gates and the EPROM is erased.

2.1.3.6 EEPROM

There is another Erasable Programmable Read Only Memory called EEPROM. That is Electrically Erasable and Programmable ROM. It does not use any special voltage to program and no ultraviolet light requirement for erasing. An elevated voltage is still required to write but it is provided on-chip instead of external voltage. And also, the electrons can be removed from the floating gate by an elevated voltage. The limitation of EEPROM is that, there is an ensured cycles of write/erase. If the operation number surpasses that cycle number, the information will not be retained.

2.1.3.7 Flash

The most popular way to store data now is flash. Flash is an almost perfect memory but too expensive. Flash is a variant of EEPROM where erasing is not possible for each address but for larger blocks or even the entire memory. Therefore, Flash EEPROM often has lower ensured write/erase cycle endurance than EEPROMs – about 1.000 to 10.000 cycles. If you do not need to erase quite often, the cycle number is enough.

2.1.3.8 Memory Access

Generally, the program memory will adopt the Flash-EEPROM type and the data memory will consist of some SRAMs and some EEPROMs. There are two kinds of methods to compose these three memory style working. The first one is to give these three memory types different address. A programmer could specify which memory would be accessed by using different access methods such as using a specific register or an index. Figure 2-9 shows the details of the first method. There is another method to compose these memory styles. That is to compose them by adopting the same address but giving them a different address range. The microcontroller uses the address to decide which memory to access. Figure 2-10 shows the details of this method.

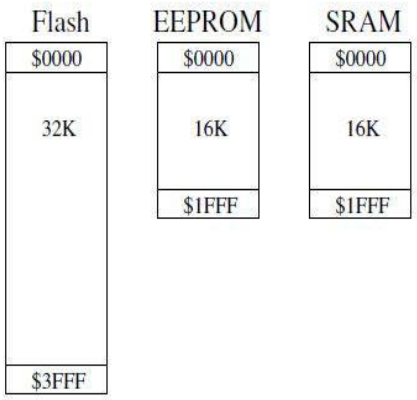


Figure 2-9 Access Method 1

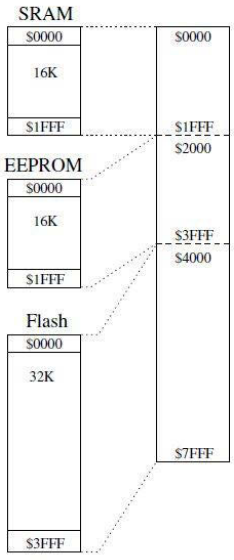


Figure 2-10 Access Method 2

2.1.4 Timers and Counters

A timer, which is also called a counter module that provides 8 and/or 16 bit resolutions, is an important part of every microcontroller. Timers are commonly used in measurement of periods or waveform generation. The most straightforward use of the timer is as a counter. However, the timers are also allowed a user to timestamp external events or to trigger an interrupt or to generate a pulse-width modulated signal for controlling purposes. As a result, most of the controlling orders in the microcontroller must be supported by the timers.

The timer is a natural counter which is either incremental or decremental upon every clock tick. That is, when a clock tick is passed, the counter will increase 1 or decrease 1. This current count value can be read or operated through a count register to read or through a specific register to set a value by the user. If a timer resolution is n, the count value is within [0, 2n − 1]. That is, if the resolution is 8, the count value is 255. The counter will increase, for example, from 0 to 255. When it counts to 256, the value will return to 0 depending on the configuration of the users. There is a problem that need be considered carefully. If a user applies a 16-bits timer on an 8-bits microcontroller, he must consider about carry bits. An 8-bits microcontroller could only operate the values up to 0X00FF while a 16-bits timer could operate the value up to 0XFFFF. At this time, if the user does nothing about this, he cannot obtain a constant after 0X00FF. In order to solve this problem, the user must adopt a buffer register to restore the carry bit.

There is another important application of the timer. The timer usually produces interrupt signals after an overflowing. At the same time, it also can be configured to generate interrupt signals in a specific count value. The interrupt is that, for example, a sensor integrated inside a microcontroller will sample 1 bit data after receiving an interrupt signal. Therefore, an interrupt is usually used as a controlling signal. When a module receives interrupt signals, this module will execute some specific action.

In the application of the timer and the counter, prescaler is a necessary part. Prescaler is used to divide the frequency of the timer to an extend frequency range. For example, an 8-bits timer provides 1MHz but the user only wants 0.5MHz for application. At this time, a prescaler (also called divider) would achieve this purpose. The prescaler is also a

counter that increases with every rising edge of the system clock. However, its least significant bit will rise 1 after twice the period of the system clock. Hence, the timer will operate with half the frequency of the system clock. Because of this, a prescaler could successfully divide 1MHz to 0.5MHz. Similar with this process, if a user set up a divider's value to 4, 1MHz will become 0.25MHz. The prescaler would widely increase the range of timer frequency and very convenient to use.

2.1.5 Interrupts of Microcontroller

Interrupt is widely used in microcontrollers. It can be regarded as an event that it happens when another event is in action. At this time, if this event has a higher priority, the acting event would be interrupted to execute this event. An event is always defined as changes in microcontroller's system. That is, when an event happens, it will change the state of the input line. The microcontroller would observe this change and respond a corresponding reaction. Generally, the microcontroller offers a convenient way to form interrupts. That is, the microcontroller polls the signal and interrupts the main program if a state change is detected. If there are no state changes, the main program will simply be executed without any interrupts. As long as the interrupt occurs, the microcontroller calls an interrupt service routine (ISR). The ISR must be provided by the programmer in a specific place in a code.

There are two main registers can be used as interrupt controlling. The first one is the Interrupt Enable (IE) register. This register is a switch of interrupt event. It indicates that the microcontroller can call an ISR in reaction to the event. Another register used for interrupt controlling is the Interrupt Flag (IF) register which is used to indicate an interrupt occurs. In consequence, IE is a key that allows interrupt event can be executed. If IE register is not set, system will ignore ISR and its interrupt events. The IF is a mark which indicates whether the interrupt event occurs. Microcontrollers periodically check the IF that if the IF register is set, the ISR will be executed. The IF register cannot be reset automatically.

Generally, IF and IE registers could solve most problems of interrupt controls. However, there are two registers that could be used in some special places. The first one is Global Interrupt Enable register which could enable or disable all interrupt operations. It is considerable to control all interrupt events at the same time. The other one is called

Non-Maskable Interrupt that only adopted in Texas Instrument MSP430 family. This kind of register cannot be disabled even by Global Interrupt Enable register. Such interrupts are useful for particularly important events.

After an interrupt is called, the programmer must tell the microcontroller that which ISR should be chosen. That is to choose an interrupt vector. An interrupt vector is a value that indicates a certain interrupt event (figure 2-11); each vector has fixed address and has a fixed base address in program memory. When an interrupt condition is satisfied and then a corresponding ISR would be called. At this time, the controller will either jump to the location given in the table or jump to the appropriate vector. Figure 2-11 is example of vector table.

Vector No.	Prg. Addr.	Source	Interrupt Definition
1	$000	RESET	External Pin, Power-on Reset, . . .
2	$004	INT0	External Interrupt Request 0
3	$008	INT1	External Interrupt Request 1
4	$00C	TIMER2 COMP	Timer/Counter 2 Compare Match
. . .	. . .	. . .	. . .

Figure 2-11 An Example of Interrupt Vector Table

There is another important concept that called priority. Priority is the rank of such interrupts in microcontroller. Most controllers with many interrupts use the interrupt vector as an indication to the priority. If an interrupt has higher priority, it is not only should execute first but also can interrupt other interrupts.

2.1.6 Digital I/O

Digital I/O is the main characteristic of microcontrollers, which directly monitor and control hardware. Digital I/O is mainly used in communication between internal modules. All microcontrollers have at least 1 or 2 digital I/O pins that can be directly connected to hardware. I/O pins are generally grouped into ports of 8 pins and bidirectional, that is, capable of both input and output. Because of the limitation of the chip size, most pins have one more alternative function which is adopted by other modules like timer or communication module. There are three registers for controlling pins. The first one is Data Direction Register (DDR) which used to select direction of pins. That is input or output direction. The second one is Port Register (PORT) which used to control the voltage level of output pins. The third one is Port Input Register (PIN) which only used to read the current state of all pins.

The digital input functionality is used when the signal need to be interpreted digitally. That is, communication between internal modules or peripheral digital devices. The signal will be transferred to 1 (high) and 0 (low). If the voltage is in the range of $V_{cc} - 0.25$ to V_{cc}, for example in MSP430F5, it is low level (digital 0). If the voltage stays in V_{ss} to $V_{ss} + 0.25$, it is high level (digital 1).

The digital output functionality is used to output given voltage levels. The levels correspond to high and low that are specified by the controller and depends on the controller's operating voltage. The registers that introduced before called DDR and PORT are used commonly in controlling pin output. Thus, if you want to output 1, first set PORT and then DDR.

2.1.7 Analog I/O

2.1.7.1 Digital to Analog Convertor (DAC)

DAC is always used as an output interface because microcontroller could only adopt digital signals internally. However, if a microcontroller requires outputing analog signals such as voltages, sounds or pictures, it should convert internal digital signals to analog signals. At this time, a Digital to Analog Convertor will be applied in this place. Here is a diagram that illustrates the relationship between microcontroller and DAC.

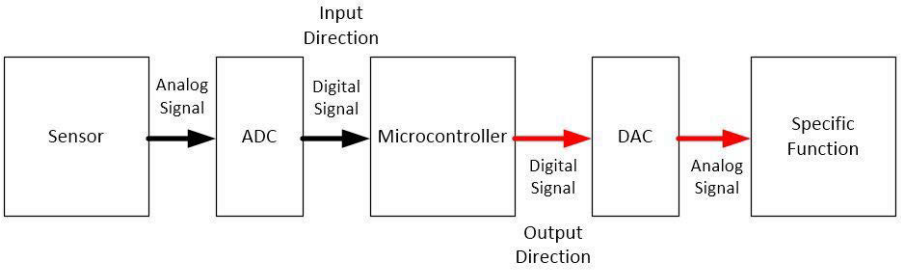


Figure 2-12 ADC and DAC

There are thousands of DACs in the market while their working principle is quite similar. Here is an example of 4 bits DAC for explaining how does DAC work. In this example, the I_{OUT1}satisfies that

$$I_{OUT1} = I_0 + I_1 + I_2 + I_3 \tag{2-1}$$

$$I_3 = \frac{V_{REF}}{2R} = 2^3 \times \frac{V_{REF}}{2^4 R} \tag{2-2}$$

$$I_2 = \frac{I_3}{2} = 2^2 \times \frac{V_{REF}}{2^4 R} \quad (2\text{-}3)$$

$$I_1 = \frac{I_2}{2} = 2^1 \times \frac{V_{REF}}{2^4 R} \quad (2\text{-}4)$$

$$I_0 = \frac{I_1}{2} = 2^0 \times \frac{V_{REF}}{2^4 R} \quad (2\text{-}5)$$

Because of the fact that the switches $S_3S_2S_1S_0$ are controlled by DAC register bits $b_3b_2b_1b_0$. The $b_3b_2b_1b_0$ maybe not equal to 1 simultaneously. Thus, the equation (1) is

$$I_{OUT1} = b_0I_0 + b_1I_1 + b_2I_2 + b_3I_3 = (2^0b_0 + 2^1b_1 + 2^2b_2 + 2^3b_3) \times \frac{V_{REF}}{2^4 R} \quad (2\text{-}6)$$

Since $I_{Rf} = -I_{OUT1}$, the output voltage is

$$V_{OUT} = I_{Rf}R_f = -(2^0b_0 + 2^1b_1 + 2^2b_2 + 2^3b_3) \times \frac{V_{REF}}{2^4 R} \times R_f = -B\frac{V_{REF}}{2^4} \quad (2\text{-}7)$$

Finally, if the DAC register is n bits, the output voltage is

$$V_{OUT} = -B\frac{V_{REF}}{2^n} \quad (2\text{-}8)$$

At this time, the digital signal B has already been converted to the analog signal voltage. Usually, the digital signal is a parallel input to the ADC convertor at one time. Therefore, the digital signal is consecutive parallel input and analog signal is consecutive output.

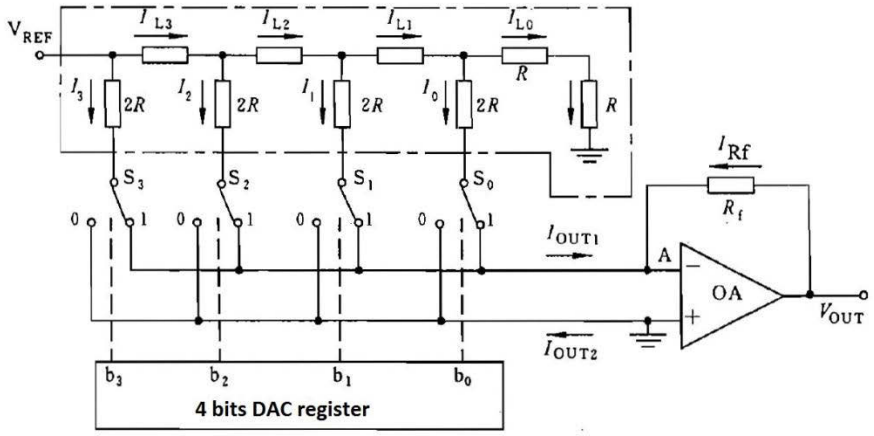


Figure 2-13 4 Bits DAC Convertor

There are some important characteristics of the DACs:

- Resolution

  This number describes the ability of a DAC that distinguishes the minimal increment of analog output. Resolution can be described as:

$$Resolution = S \times 2^{-n} \quad (2\text{-}9)$$

where S is maximum range of the DAC and n is the bit of the DAC. For example, if a DAC's range is 10V and its bit is 8, the resolution is $10V \times 2^{-8} = 39mV$.

- Conversion Accuracy

  This describes how close the actual output and the theoretical output. Generally, it equals to half of the resolution.

- Offset Error

  This describes the output number when the input is zero.

- Linearity

  This describes the maximum deviation between the actual characteristic conversion curve and theoretical characteristic conversion curve.

2.1.7.2 Analog to Digital Convertor (ADC)

The analog to digital convertor is used to convert an analog signal (temperature, voltage, current) to a digital signal that can be operated inside a microcontroller. It is commonly applied in every kind of sensors. ADCs work principle is similar to a curve integral in math. It divides a continuous analog signal into small pieces and converts that to a digital signal. Here is an example to demonstrate the ADC work principle. This is a 3-bits ADC. Therefore, its analog voltage will be converted to a digital signal in the range from 0 to2^3 (000 to 111). In other words, if an ADC has n-bits, its range is from 0 to2^n. At the same time, there is also a reference voltage V_{REF} used as comparing input. This V_{REF} is divided into 7 values by different resistances shown in figure 2-14. The V_x is the analog input voltage. This V_x will compare to those 7 values by 7 different comparators (C_1 to C_7). The result will be coded to a 3-bits binary code by a coder circuit. This binary code is digital signal that can be operated inside the microcontroller.

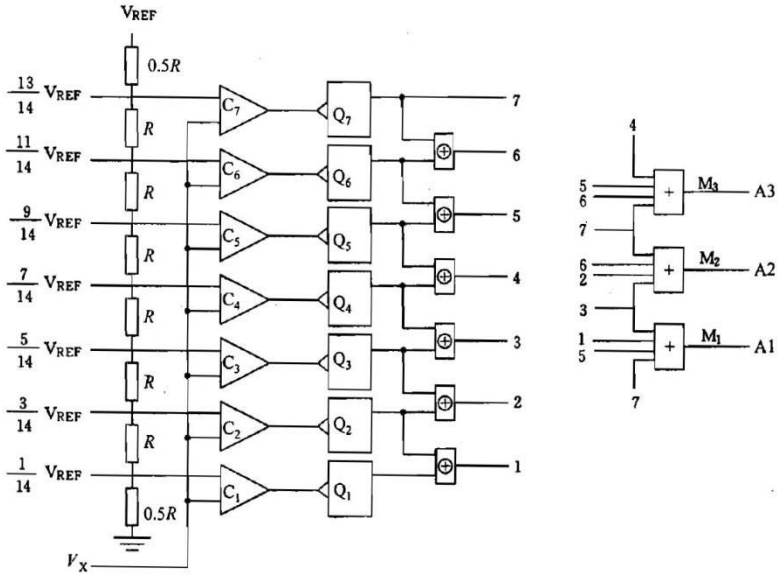


Figure 2-14 ADC Work Principle

In this example, there are some important characteristics:

- Resolution

  The number of n is the resolution of ADC. Usually, the higher the resolution is, the more expensive the ADC is.

- Granularity

  Those 7 values that used for comparing is called granularity.

- Conversion Rate

  This characteristic is the reciprocal of period when completes one time conversion.

- Conversion Accuracy

  This accuracy is consisted by the digital accuracy and the analog accuracy. The analog accuracy is affected by the resistance inside comparator and the coder or the reference voltage fluctuations. The digital accuracy is affected by the number of bit of the output. If the number of bit is increasing, there is a higher error.

2.1.8 Microcontroller Experiments

The experiments could help to understand the basic knowledge of the microcontrollers deeply. In my experiments, I used a Launchpad which is produced by TI as an experiment board. The microcontroller of this Launchpad is the MSP430G2231 which is very close to the microcontroller MSP430F5528 which is applied in the IMU. In this part, I will firstly list its characteristics and secondly introduce these experiments.

2.1.8.1 MSP430G2231 Launchpad

The MSP430G2231 is an ultra-low-power mixed signal microcontroller which integrates a 16-bits timer and a ten I/O pins and a 10-bit A/D converter and built-in a communication capability by using synchronous protocols (SPI or I2C). Typical applications include that low-cost sensor systems capturing analog signals and converting them to digital values or processing the data for display or for transmission to a host system. The package diagram and function diagram are shown in figure 2-15. The functionality of these pins is displayed in table 2-1.

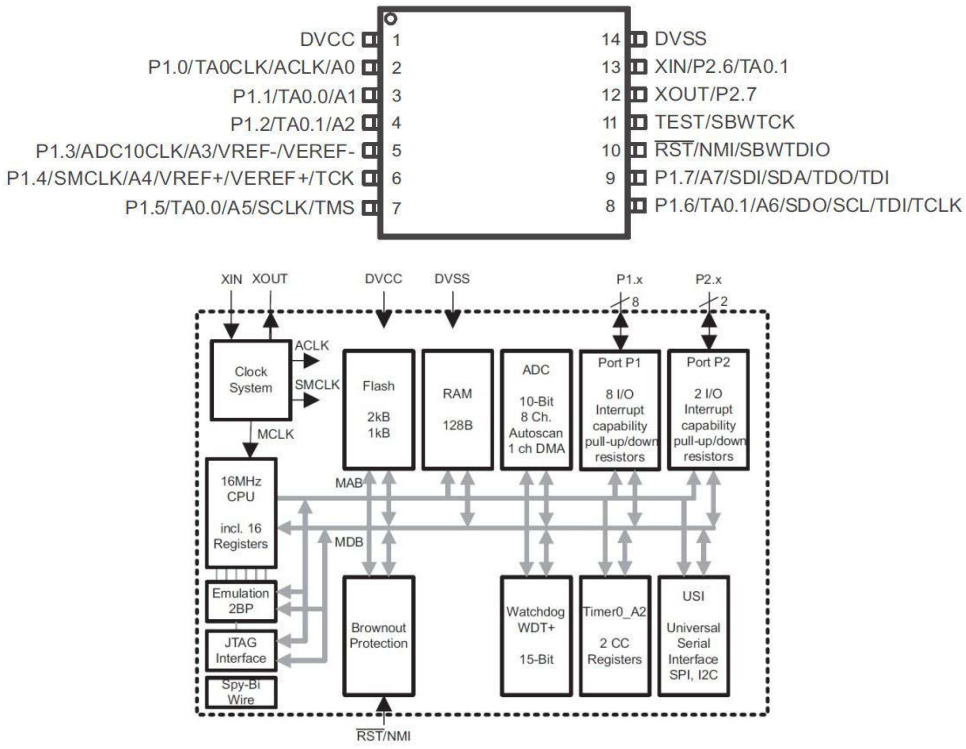


Figure 2-15 Package Diagram and Function Diagram

Table 2-1 Functions of Pins

Name	Pin NO.	Description
DVCC	1	Supply voltage
P1.0/ TA0CLK/ ACLK/ A0	2	General-purpose digital I/O pin Timer0_A, clock signal TACLK input ACLK signal output ADC10 analog input A0
P1.1/	3	General-purpose digital I/O pin

TA0.0/ A1/		Timer0_A,capture:CCI0Ainput,compare: Out0 output ADC10 analog input A1
P1.2/ TA0.1/ A2/	4	General-purpose digital I/O pin Timer0_A,capture:CCI1Ainput,compare: Out1 output ADC10 analog input A2
P1.3/ ADC 10CLK/ A3/ VREF-/VEREF	5	General-purpose digital I/O pin ADC10, conversion clock output ADC10 analog input A3 ADC10 negative reference voltage
P1.4/ SMCLK/ A4/ VREF+/VEREF+/ TCK	6	General-purpose digital I/O pin SMCLK signal output ADC10 analog input A4 ADC10 positive reference voltage JTAG test clock, input terminal for device programming and test
P1.5/ TA0.0/ A5/ SCLK/ TMS	7	General-purpose digital I/O pin Timer0_A, compare: Out0 output ADC10 analog input A5 USI:clock input in I2C mode; clock input/output in SPI mode JTAG test mode select, input terminal for device programming and test
P1.6/ TA0.1/ A6/ SDO/ SCL/ TDI/TCLK	8	General-purpose digital I/O pin Timer0_A,capture:CCI1Ainput,compare: Out1 output ADC10 analog input A6 USI: Data output in SPI mode USI: I2C clock in I2C mode JTAG test data input or test clock input during programming and test
P1.7/	9	General-purpose digital I/O pin

A7/ SDI/ SDA/ TDO/TDI		ADC10 analog input A7 USI: Data input in SPI mode USI: I2C data in I2C mode JTAG test data output terminal or test data input during programming and test
$\overline{RST}$/ NMI/ SBWTDIO	10	Input terminal of crystal oscillator General-purpose digital I/O pin Timer0_A, compare: Out1 output
TEST/ SBWTCK	11	Output terminal of crystal oscillator General-purpose digital I/O pin
XOUT/ P2.7	12	Reset Nonmaskable interrupt input Spy-Bi-Wire test data input/output during programming and test
XIN/ P2.6/ TA0.1	13	Selects test mode for JTAG pins on Port 1.The device protection fuse is connected to TEST. Spy-Bi-Wire test clock input during programming and test
DVSS	14	Ground reference

Based on the powerful functionalities of the MSP430G2231, the Launchpad is a convenient and widely used experiment board. The most outstanding design on this pad is the on-board emulation that allows users to program and debug without any additional tools. It also extends the number of pins that could output or input more resources. A general purpose switch and a reset switch and two LEDs are provided on the board for quick and simple development. The MSP430G2231 has 14 pins in general. However, there are 20 pins in Launchpad. The additional 6 pins are general purpose I/O pins which could provide users to connect more peripherals. The Launchpad has most functions of MSP430 series. Therefore, a user could operate all kinds of MSP430 series microcontrollers as long as he can understand the Launchpad well. The details of the Launchpad is shown in figure 2-16.

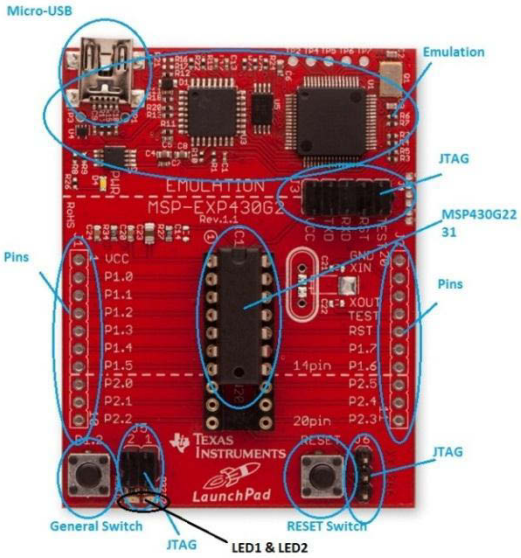


Figure 2-16 Launchpad

2.1.8.2 Experiments

Two professional compilers for the MSP430 are available. They are the IAR Systems and the Code Composer Studio system. The free versions have limitations that the code size you can create should less than 4 kb for IAR and 16 kb for CCS. For larger programs, you can either purchase the full compilers or use the MSP open source compiler on their website. But for the purposes of the experiment and IMU project I will use the TI's Code Composer Studio.

The MSP430 will run in any case between 1.8 V and 3.6 V, although at least 2.2 V is required to do any programming to the chip. Another important thing that needs to be noticed carefully is the speed at which the chip is able to run depends on the voltage. Though it's capable of running up to 16 MHz, the 1.8 V Launchpad cannot run any faster than 6 MHz.

a. Experiment 1: Blink LED

The purpose of this experiment is to understand the CCS panel and its operation. In this experiment, I will introduce two points. The first one is the 3 useful registers and how to use registers under this panel. The second one is the operators inside CCS.

The registers of the MSP430 are special sections of its Memory that configure and control the device. The registers are grouped into three sections: the special function registers (SFR), the 8-bit peripherals (need only 8-bits to read instructions), and the 16-

bit peripherals (obviously those need a full 16-bits to read instructions). When you require to using the registers, you just need to visit correct address. In assembly language, a user needs to type correct instruction code and address. However, thanks to CCS, it renames these addresses to a readable text in its library. The memory map of the MSP430G2231 which is shown in figure 2-17 can be helpful to understand the location of these registers. And a useful code library in this experiment is shown in figure 2-18 to understand new name of these addresses.

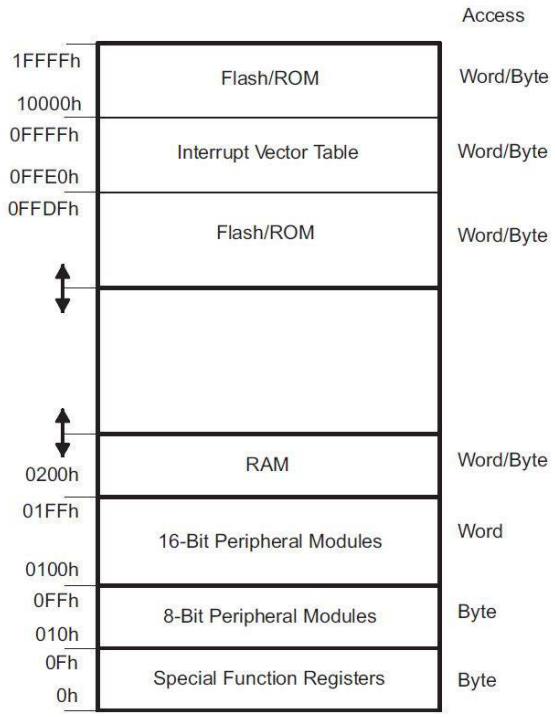


Figure 2-17 The Memory Map of MSP430G2231

```
/************************************************************
* DIGITAL I/O Port1/2 Pull up / Pull down Resistors
************************************************************/
P1IN               = 0x0020;
P1OUT              = 0x0021;
P1DIR              = 0x0022;
P1IFG              = 0x0023;
P1IES              = 0x0024;
P1IE               = 0x0025;
P1SEL              = 0x0026;
P1REN              = 0x0027;
P2IN               = 0x0028;
P2OUT              = 0x0029;
P2DIR              = 0x002A;
P2IFG              = 0x002B;
P2IES              = 0x002C;
P2IE               = 0x002D;
P2SEL              = 0x002E;
P2REN              = 0x002F;
/************************************************************
* Timer A2
************************************************************/
TAIV               = 0x012E;
TACTL              = 0x0160;
TACCTL0            = 0x0162;
TACCTL1            = 0x0164;
TAR                = 0x0170;
TACCR0             = 0x0172;
TACCR1             = 0x0174;
/************************************************************
* USI
************************************************************/
USICTL0            = 0x0078;
USICTL1            = 0x0079;
USICKCTL           = 0x007A;
USICNT             = 0x007B;
USISRL             = 0x007C;
USISRH             = 0x007D;
USICTL             = 0x0078;
USICCTL            = 0x007A;
USISR              = 0x007C;
/************************************************************
* WATCHDOG TIMER
************************************************************/
WDTCTL             = 0x0120;
```

Figure 2-18 Library of MSP430g2231

In this experiment, we need to visit address 0x0120 (WDTCTL, watchdog timer control register), 0x0021 (P1OUT) and 0x0022 (P1DIR) to blink LED. How do these registers help to achieve this function? This code will be displayed in figure 2-19.

```
#include <msp430.h>

/*
 * main.c
 */
unsigned int i=0;

int main(void)
{
    WDTCTL = WDTPW + WDTHOLD;   // Stop watchdog timer


P1DIR = 0x01;

for (;;)


{P1OUT ^= 0x01;
    for (i=0; i< 65535; i++);
    }



    }

```

Figure 2-19 Blink LED

The CCS is a panel that under C Programming. Therefore, the project needs a header file *"#include <msp430.h>"*. This header file tells CCS which library needs to be used in this project. For example, in this experiment, the header file is *"msp430.h"* which means every address will be renamed like figure 2-18. Libraries in the CCS are actually a set of renamed addresses.

Similar with C Programming, the CCS requires a *"main"* function. The main function is the start line of the whole project. Thus, the first sentence of this project is *"WDTCTL = WDTPW + WDTHOLD"*. In order to understand this sentence, we need an assistance of the User Guide of MSP430. In the User Guide of the MSP430, *"WDTCTL"* is a register of Watchdog Timer. The Watchdog Timer is a module that could reset system after a selected period. If the Watchdog Timer is not applied, it needs to be halted or configured as a normal internal timer. In this experiment, the Watchdog Timer is not used. Thus, the order *"WDTCTL = WDTPW + WDTHOLD"* is halt the Watchdog Timer. *"WDTCTL"* is a 16 bits register that used to control the Watchdog Timer. *"WDTPW"* is the Watchdog Timer Password that used to tell the CCS that "I want to configure Watchdog Timer". *"WDTHOLD"* is the instruction that halt the Watchdog Timer. The details of these three are illustrated in figure 2-20.

15	14	13	12	11	10	9	8
WDTPW, Read as 069h Must be written as 05Ah							
7	6	5	4	3	2	1	0
WDTHOLD	**WDTNMIES**	**WDTNMI**	**WDTTMSEL**	**WDTCNTCL**	**WDTSSEL**	**WDTISx**	
rw-0	rw-0	rw-0	rw-0	r0(w)	rw-0	rw-0	rw-0

WDTPW	Bits 15-8	Watchdog timer+ password. Always read as 069h. Must be written as 05Ah, or a PUC is generated.
WDTHOLD	Bit 7	Watchdog timer+ hold. This bit stops the watchdog timer+. Setting WDTHOLD = 1 when the WDT+ is not in use conserves power.

Figure 2-20 Details of WDT Register

The second sentence is *"P1DIR = 0x01"*. We also need an assistance of the User Guide. The User Guide tells that register *"PxDIR"* is used to select the direction of the corresponding I/O pin. Therefore, we should set *P1DIR* to 1 in order to configure its output direction before using. It is noteworthy that this sentence here will set P1.0 pins as output. Similar to this, the register *"PxOUT"* is used to control its output. If we set *"P1OUT"* to 0x01, P1.0 pin will output high level.

The third sentence is a *"for cycle"*. The content which is inside this cycle block will be executed forever as long as there is no order to halt it. In this experiment, this sentence means that the LEDs will blink until power off.

Notice that there is a special mark *"^"* in the cycle block. This is an "exclusive or operator". In this sentence, the *"P1OUT ^= 0x01"* means toggling. As previous mentioned, set the *"P1OUT"* to 0x01 is to set the P1.0 output high level. Therefore, adding a *"^"* will make the P1.0 output toggle between 0 and 1.

The last sentence is also a *"for cycle"*. This cycle is used to set the toggling time. Because the default frequency of the Launchpad is 32768Hz (32kHz), that is to say, after 32768 periods, the real time is 1 second. And also, the parameter *"i"* increase 1 after each period. Therefore, the number 65535 is actually 2 seconds.

This experiment is the getting start of using the TI microcontrollers. It introduces three basic points. The first one is every MSP430 series microcontroller should be configured its Watchdog Timer first. The second one is that a user should configure the pin direction before using. The last one is to learn exclusive or operator.

b. Experiment 2: Using Timer to Blink LEDs

This experiment focuses on how to use its timer to control LEDs. The timer is a more powerful 16 bits counter in the MSP430 series microcontrollers. Most importantly, it has higher accuracy than the system clock and can be used as an interrupt resource. This experiment uses a TimerA interrupt function to control LEDs. The code is shown in figure 2-21. In this experiment, there are 3 main points to be introduced. The first one is the keyword *"#define"*. The second one is another three operators. The last one is the TimerA operation.

```
3 #include <msp430g2231.h>
4 #define LED_Red BIT0
5 #define LED_Green BIT6
6
7
8 void main()
9 {
10     WDTCTL=WDTPW+WDTHOLD;
11     P1DIR|=(LED_Red+LED_Green);
12     P1OUT&= ~(LED_Red+LED_Green);
13 TACCTL0=CCIE;
14             TACTL=TASSEL_2+MC_2;
15
16 __enable_interrupt();
17
18 }
19 #pragma vector=TIMERA0_VECTOR
20 __interrupt void TimerA0 ( void )
21 {
22     P1OUT=0x00;
23  __delay_cycles(512384);
24  P1OUT=0x01;
25  __delay_cycles(1024768);
26  P1OUT ^= 0x40;
27  __delay_cycles(512384);
28
29  }
```

Figure 2-21 Using TimerA to Control LED

```
* STANDARD BITS
*****************************************

#define BIT0                (0x0001)
#define BIT1                (0x0002)
#define BIT2                (0x0004)
#define BIT3                (0x0008)
#define BIT4                (0x0010)
#define BIT5                (0x0020)
#define BIT6                (0x0040)
#define BIT7                (0x0080)
#define BIT8                (0x0100)
#define BIT9                (0x0200)
#define BITA                (0x0400)
#define BITB                (0x0800)
#define BITC                (0x1000)
#define BITD                (0x2000)
#define BITE                (0x4000)
#define BITF                (0x8000)
```

Figure 2-22 MSP430G2231 Library for General Purpose Pins

The keyword *"#define"* is widely used in the library. The functionality of this keyword is to rename. As mentioned previously, the microcontroller register is coded by different addresses. These addresses are numbers and hard to remember. At this time, the keyword *"#define"* can give these addresses an easy remember name. Figure 2-22 shows the first rename in the library and figure 2-21 is the second rename. In the experiment, I rename *"BIT0"* as *"LED_Red"* and *"BIT6"* as *"LED_Green"*.

In the last experiment, I have introduced the operator *"^"*. There are another three operators that I could introduce in this experiment. In the line 11 and 12, there are two

sentence *"P1DIR |= (LED_Red + LED_Green)"* and *"P1OUT &= ~(LED_Red + LED_Green)"*. The first one set *"LED_Red"* and "*LED_Green"* to 1 simultaneously. The operator *"|"* is "*or"*. The second one set *"LED_Red"* and "*LED_Green"* to 0 simultaneously. The operator *"&"* is *"and"* and *"~"* is *"Not"*. Actually, these two sentences have an opposite function.

The central part of this experiment is to use the TimerA. The TimerA here is an interrupt resource. When the system reaches to *"_enable_interrupt()"*, the system recognises that this is an interrupt operation. Then it will find the interrupt vector to execute the interrupt block. However, similar with configure pins, the TimerA also need to be configured before its using. In the TimerA, there is a register called *"TACCR0"* which is a capture and compare register. This register can be given a specific number that is lower than 0FFFFh. And then, the register will capture the counter number inside the TimerA. If the captured number equals to this given number, the counter will return to 0 or stop increasing to decreasing which depends on working mode. At the same time, the TimerA will produce an interrupt signal. Therefore, the two sentences *"TACCTL0 = CCIE"* and *"TACTL = TASSEL_2 + MC_2"* are used to configure the TimerA's working mode. The first sentence enables the register *"TACCR0"* to be interrupted. The second sentence selects the clock resource and the working mode of the TimerA. After this, the TimerA will working under the frequency of 1MHz and continue mode.

The working mode of TimerA is illustrated as follows:

MCx	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TACCR0.
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/down	The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.

Figure 2-23 TimerA Working Modes

The register *"TACTL"* can be used to select the TimerA's working mode. In this experiment, I select the continuous mode. Therefore, the counter will count from 0 to 0FFFFh and then return to 0 immediately and count again. When the counter reaches to 0FFFFh, the system will execute the interrupt vector once. Thus, the interrupt vector can be executed once in every period.

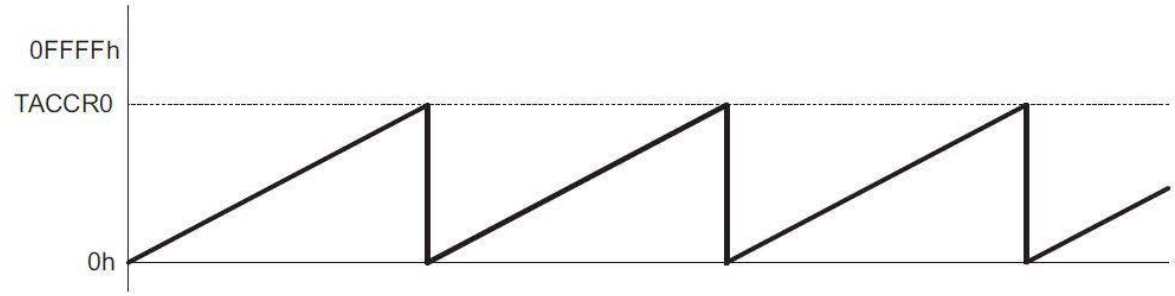


Figure 2-24 Up Mode

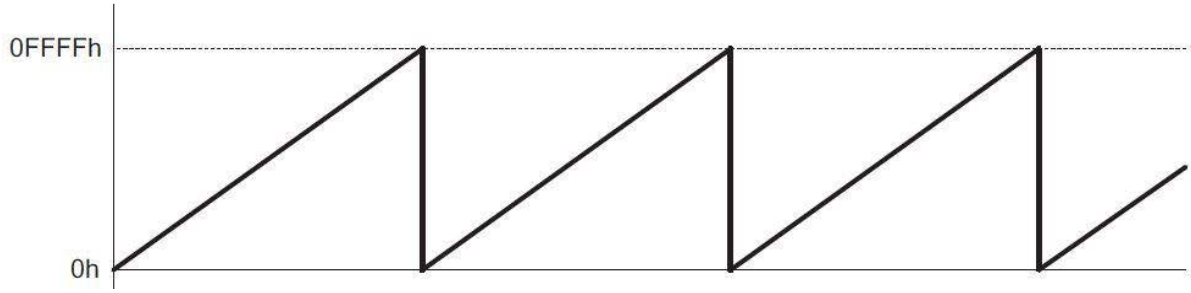


Figure 2-25 Continuous Mode

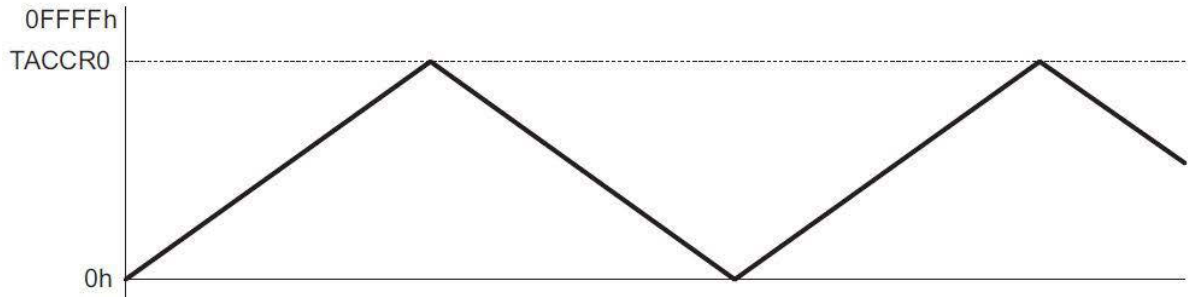


Figure 2-26 Up and Down Mode

Inside the interrupt vector, there is a sentence called *"__delay_cycles()"*. There are 2 different numbers inside the bracket. This sentence makes a delay after the LED blinking. That is to say, the period of turn on and turn off. The number is the period. Because of the TimerA is working in 1MHz. Therefore, the number 1024768 equals to 1 second and 512384 equals to 0.5 seconds. By changing this number, it is simple to control its blinking time.

c. Experiment 3: Using Button to Control LEDs

The Launchpad has two buttons that located on the bottom of the board. According to the User Guide of the Launchpad, the button S1 connects to the RST pin and button S2 connects to the pin p1.3. When the button is up, the resistor ties the pins to a Vcc that pulling up the voltage to whatever level the board is. When press the button, the short to ground is occurred. Thus non-pressed is logic 1, and pressed is logic 0. At the same time, it is viable to use the opposite convention as well. The new thing that is needed in this experiment is to tell the Launchpad to check the state of the button periodically. This is called polling. When the program reaches the point where it is waiting for the user to push the button, it checks and rechecks the button until it has been pressed. Since the button is attached to the P1.3, the bit 3 of the P1IN tells us whether the button is pressed (logic 0) or non-pressed (logic 1).

Like the TimerA in the last experiment, buttons can also be used as an interrupt resource. That is, when the button is pressed, the connected pin, such as the p1.3, will generate an interrupt signal. At this time, the system will find the interrupt vector to execute the interrupt code block.

```
1 #include <msp430g2231.h>
2
3 #define LED_0 BIT0
4 #define LED_1 BIT6
5 #define LED_OUT P1OUT
6 #define LED_DIR P1DIR
7 #define BUTTON BIT3
8
9 unsigned int blink = 0;
10
11 void main(void)
12 {
13 WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
14 LED_DIR |= (LED_0 + LED_1); // Set P1.0 and P1.6 to output direction
15 LED_OUT &= ~(LED_0 + LED_1); // Set the LEDs off
16 P1IE |= BUTTON;
17
18 __enable_interrupt();
19
20 for (;;)
21 {
22
23 if(blink > 0)
24 {
25 P1OUT ^= (LED_0 + LED_1); // Toggle P1.0 and P1.6 using exclusive-OR
26
27 __delay_cycles(100000); // SW Delay of 10000 cycles at 1Mhz
28
29 }
30 }
31
32 }
33
34 // Port 1 interrupt service routine
35 #pragma vector=PORT1_VECTOR
36 __interrupt void Port_1(void)
37 {
38 blink ^= 0x01;
39
40
41 P1IFG &= ~BUTTON; // P1.3 IFG cleared
42 }
```

Figure 2-27 Using Button to Control LEDs

The button interrupt should assist by 2 important registers. The first one is the line 16 *"P1IE |= BUTTON"*. The second one is the line 41 *"P1IFG &= ~BUTTON"*. The *"P1IE"* register is used to enable port interrupt and the *"P1IFG"* is an interrupt flag that is used to indicate the interrupt situation. When the interrupt signal from the button is detected, the interrupt flag will be set. However, the interrupt flag can be set automatically while it cannot be reset automatically. That is to say, when the flag is set, it must be reset manually. If the interrupt flag will not be reset, the interrupt vector will not respond next time.

Therefore, when a user presses the button, the system reaches line 38 directly. The parameter *blink* is set to 1 first time and reset its interrupt flag. After that, the system will return to the line 23. The LEDs will blink every 1 second. If user presses the button again, the parameter *blink* in line 38 will be reset to 0. The line 23 will not be satisfied and executed. Therefore, LEDs will stop blinking.

In this experiment, I use the button interrupt to control LEDs' blinking.

d. Experiment 4: Using ADC

The former part has already introduced the ADC principle. The ADC is used to convert an analog to a digital signal. This experiment will illustrate how to use the ADC. In this experiment, there are 3 points that need to be noticed. The first one is the Clock System Configuration. The second one is the ADC configuration. The third one is the Low Power Mode. Figure 2-28 is the details of this experiment.

```
#include  "msp430g2231.h"

#define     LED0                  BIT0
#define     LED1                  BIT6

unsigned int value=0;
void ConfigureAdc(void)
{
  /* Configure ADC  Channel */
  ADC10CTL1 = INCH_5 + ADC10DIV_3 ;         // Channel 5, ADC10CLK/4
  ADC10CTL0 = SREF_0 + ADC10SHT_3 + ADC10ON + ADC10IE;  //Vcc & Vss as reference
  ADC10AE0 |= BIT5;                         //P1.5 ADC option
}
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  BCSCTL1 = CALBC1_1MHZ;                    // Set range
  DCOCTL = CALDCO_1MHZ;
  BCSCTL2 &= ~(DIVS_3);                  // SMCLK = DCO = 1MHz
  P1DIR |= LED0 + LED1;
  P1SEL |= BIT5;                             //ADC Input pin P1.5
  P1OUT &= ~(LED0 + LED1);

  ConfigureAdc();
  __enable_interrupt();                     // Enable interrupts.

  while(1)
    {
      __delay_cycles(1000);                   // Wait for ADC Ref to settle
      ADC10CTL0 |= ENC + ADC10SC;             // Sampling and conversion start
      __bis_SR_register(CPUOFF + GIE);        // LPM0 with interrupts enabled
      value = ADC10MEM;
      if (value>511)
      {
         P1OUT &= ~(LED0 + LED1);
         P1OUT |= LED0;
      }
      else
      {
         P1OUT &= ~(LED0 + LED1);
         P1OUT |= LED1;}
      }
  }
// ADC10 interrupt service routine
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR (void)
{
  __bic_SR_register_on_exit(CPUOFF);        // Return to active mode
}
```

Figure 2-28 ADC Experiment

This experiment applies the ADC to compare input voltage. If the voltage is higher than a specific value, the LED0 (red) will be lighted. If the voltage is lower than the specific value, the LED1 (green) will light. The code starts at main function. It first of all configures Clock System.

The clock system includes four clock sources:

• LFXT1CLK: This is a Low-frequency and high-frequency oscillator that can be used with low-frequency watch crystals, external clock sources of 32768 Hz, standard crystals, resonators, external clock sources in the 400-kHz to 16-MHz range.

• XT2CLK: This is an optional high-frequency oscillator that can be used with standard crystals, resonators, external clock sources in the 400-kHz to 16-MHz range.

• DCOCLK: This is an internal digitally controlled oscillator (DCO).

• VLOCLK: This is an internal very low power, low frequency oscillator with 12-kHz typical frequency.

The main clock signals originate from those 4 resources. Three clock signals are available in the clock system:

• ACLK: This is Auxiliary clock which can select LFXT1CLK or VLOCLK as clock resource by software. It also can be divided by 1, 2, 4, or 8 and provides clock signals for individual peripheral modules.

• MCLK: This is Master clock that can select LFXT1CLK, VLOCLK, XT2CLK (if available), or DCOCLK as resources. MCLK is divided by 1, 2, 4, or 8 and mainly used by the CPU and system.

• SMCLK: Sub-main clock can originate from LFXT1CLK, VLOCLK, XT2CLK (if available on-chip), or DCOCLK. It can be divided by 1, 2, 4, or 8 and provides clock signals for individual peripheral modules.

These 4 clock resources and 3 clocks compose the Clock System. All modules inside or connected outside can use this Clock System as their clock signals. In this experiment, the MSP430 system can use register BCSCTL1, BCSCTL2 and DCOCTL to configure Clock System in line 17, 18, 19. The line 17 *"BCSCTL1 = CALBC1_1MHZ"* configures ACLK working under 1MHz. the line 18 *"DCOCTL = CALDCO_1MHZ"* configures DCO to 1MHz. The line 19 *"BCSCTL2 &= ~(DIVS_3)"* configures register BCSCTL2 to 0. The register BCSCTL2 is shown in figure 2-29. This operation set all contents to 0.

7	6	5	4	3	2	1	0
SELMx		DIVMx		SELS	DIVSx		DCOR(1)(2)
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

SELMx	Bits 7-6	Select MCLK. These bits select the MCLK source.
	00	DCOCLK
	01	DCOCLK
	10	XT2CLK when XT2 oscillator present on-chip. LFXT1CLK or VLOCLK when XT2 oscillator not present on-chip.
	11	LFXT1CLK or VLOCLK
DIVMx	Bits 5-4	Divider for MCLK
	00	/1
	01	/2
	10	/4
	11	/8
SELS	Bit 3	Select SMCLK. This bit selects the SMCLK source.
	0	DCOCLK
	1	XT2CLK when XT2 oscillator present. LFXT1CLK or VLOCLK when XT2 oscillator not present
DIVSx	Bits 2-1	Divider for SMCLK
	00	/1
	01	/2
	10	/4
	11	/8
DCOR	Bit 0	DCO resistor select. Not available in all devices. See the device-specific data sheet.
	0	Internal resistor
	1	External resistor

Figure 2-29 BCSCTL2 Register

The second thing is configure ADC. The ADC core converts an analog input to its 10-bit digital representation and stores the result in the ADC10MEM register. It uses two programmable voltage levels (VR+ and VR-) to define the range limits of the conversion. The digital output (NADC) is full scale (03FFh) when the input signal is equal to or higher than VR+ while zero when the input signal is equal to or lower than VR-. Conversion results may be in straight binary format or 2s-complement format. The conversion formula for the ADC result is:

$$N_{ADC} = 1023 \times \frac{V_{IN} - V_{R-}}{V_{R+} - V_{R-}} \tag{2-10}$$

Thus, we can also obtain voltage input V_{IN} from equation (2-10):

$$V_{IN} = \frac{N_{ADC} \times (V_{R+} - V_{R-})}{1023} + V_{R-} \tag{2-11}$$

The ADC10 core is configured by two registers (ADC10CTL0 and ADC10CTL1). The core is on with the ADC10ON bit and ENC must be set to 1 before any conversion started.

The ADC10CLK is used both as the conversion clock and to generate the sampling period. The ADC10 resource clock is selected by ADC10SSELx bits and can be divided from 1, 2, 4, 8 through ADC10DIVx bits. Its sources are SMCLK, MCLK, ACLK, and internal oscillator ADC10OSC. The clock which is chosen for ADC10CLK must remain active until the end of a conversion. If the clock is closed during a conversion, the operation does not complete and any result is invalid. The ADC clock configuration details are shown in figure 2-30.

over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted)

PARAMETER		TEST CONDITIONS		V_{CC}	MIN	TYP	MAX	UNIT
$f_{ADC10CLK}$	ADC10 input clock frequency	For specified performance of ADC10 linearity parameters	ADC10SR = 0	3 V	0.45		6.3	MHz
			ADC10SR = 1		0.45		1.5	
$f_{ADC10OSC}$	ADC10 built-in oscillator frequency	ADC10DIVx = 0, ADC10SSELx = 0, $f_{ADC10CLK} = f_{ADC10OSC}$		3 V	3.7		6.3	MHz
$t_{CONVERT}$	Conversion time	ADC10 built-in oscillator, ADC10SSELx = 0, $f_{ADC10CLK} = f_{ADC10OSC}$		3 V	2.06		3.51	µs
		$f_{ADC10CLK}$ from ACLK, MCLK, or SMCLK, ADC10SSELx ≠ 0			13 × ADC10DIV × $1/f_{ADC10CLK}$			
$t_{ADC10ON}$	Turn-on settling time of the ADC	(1)					100	ns

(1) The condition is that the error in a conversion started after $t_{ADC10ON}$ is less than ±0.5 LSB. The reference and input signal are already settled.

Figure 2-30 ADC Clock Details

An ADC interrupt should be introduced simply in this experiment. Similar with TimerA interrupt, if both the ADC10IE and the GIE bits are set, the ADC10IFG flag generates an interrupt request and the ADC10IFG flag is automatically reset when the interrupt request is finished, or it reset by software.

In this experiment, the line 10, 11, 12 are used to configure ADC. In the line 10, *"ADC10CTL1 = INCH_5 + ADC10DIV_3"* set pin p1.5 as analog input port and clock divider as 4. The line 11 *"ADC10CTL0 = SREF_0 + ADC10SHT_3 + ADC10ON + ADC10IE"* selects Vcc and Vss as reference voltages and set sampling and hold time time to 64 ADC10CLKs. The line 12 *"ADC10AE0 |= BIT5"* is used to enable analog input from pin p1.5.

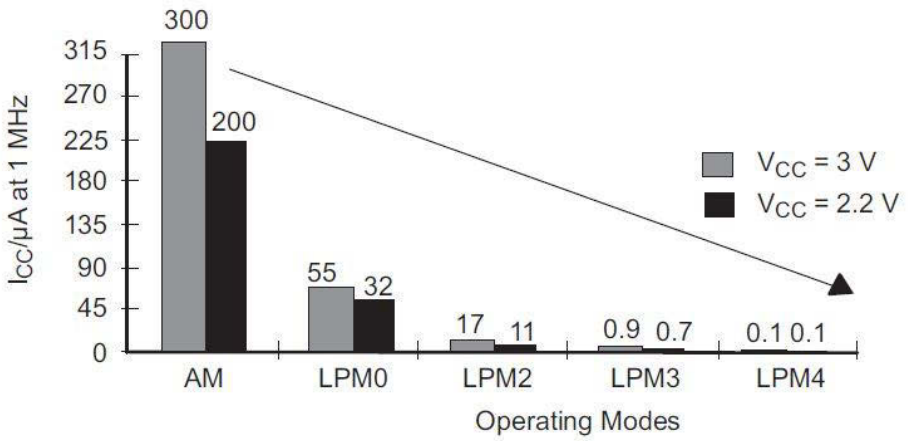
The last one should be introduced is Low Power Mode. The MSP430 family is designed for ultralow-power applications and different modes. The operating modes could reach 3 different purposes:

• Ultralow-power

• Speed and data throughput

• Minimization of individual peripheral current consumption

The MSP430 typical current consumption is shown in figure 2-31.



SCG1	SCG0	OSCOFF	CPUOFF	Mode	CPU and Clocks Status
0	0	0	0	Active	CPU is active, all enabled clocks are active
0	0	0	1	LPM0	CPU, MCLK are disabled, SMCLK, ACLK are active
0	1	0	1	LPM1	CPU, MCLK are disabled. DCO and DC generator are disabled if the DCO is not used for SMCLK. ACLK is active.
1	0	0	1	LPM2	CPU, MCLK, SMCLK, DCO are disabled. DC generator remains enabled. ACLK is active.
1	1	0	1	LPM3	CPU, MCLK, SMCLK, DCO are disabled. DC generator disabled. ACLK is active.
1	1	1	1	LPM4	CPU and all clocks disabled

Figure 2-31 Current Consumption in Different Operating Modes and Details of Modes

The low-power modes 0 to 4 are configured with the CPUOFF, OSCOFF, SCG0, and SCG1 bits in the status register respectively.

After setting any of the mode-control bits, the selected operating mode takes effect immediately. Peripherals and their individual control registers operating with any disabled clock are disabled until the clock becomes active. However, all I/O port pins and RAM/registers are unchanged. Wake up can be executed through all enabled interrupts.

In this experiment, the line 31 *"__bis_SR_register(CPUOFF + GIE)"* make system enter Low Power Mode 0. In this situation, the only active module in this system is ADC. Therefore, the low power mode can stop other function except ADC to save power consumption. The line 48 *"__bic_SR_register_on_exit(CPUOFF)"* can exit the low power mode. This mode would be widely used in portable devices.

e. Experiment 5: How to Use Thermometer

In the last experiment, I introduced ADC10 and used it to compare voltage. In this experiment, I will display how to use another ADC10, the thermometer. Here is the code.

```
1 #include<msp430g2231.h>
2 void tempInit()
3 {
4  ADC10CTL0=SREF_1 + REFON + ADC10ON + ADC10SHT_3 ; //1.5V ref,Ref on,64 clocks for sample
5  ADC10CTL1=INCH_10+ ADC10DIV_3; //temp sensor is at 10 and clock/4
6 }
7 int tempOut()
8 {    int t=0;
9  __delay_cycles(1000);              //wait 4 ref to settle
10 ADC10CTL0 |= ENC + ADC10SC;      //enable conversion and start conversion
11  while(ADC10CTL1 & BUSY);         //converting
12  t=ADC10MEM;                       //store val in t
13  ADC10CTL0&=~ENC;                    //disable adc conv
14  return(int) ((t * 27069L - 18169625L) >> 16); //convert and pass
15 }
16 void main(void)
17 {   volatile int temp;    //initialise
18  WDTCTL = WDTPW + WDTHOLD;
19   temp=0;
20  tempInit();//initialise adc
21   while(1)
22  {
23  __delay_cycles(500); //wait and set break point
24    temp=tempOut();           //read temp
25  __delay_cycles(500);  //wait and set breakpoint
26    }
27 }
```

Figure 2-32 Using Thermometer

In this code, there are three code blocks. The first one is the *tempInit()* which is used to configure the ADC10. The second one is the *tempOut()* which is adopted to control the thermometer. The last one is the *main()*.

The *tempInit()* function configures the ADC10 that it sets a reference voltage to 1.5V and a sampling holding period to 64 clocks. The line 5 is used to select working mode as temperature sensor and divider in one fourth.

The function *tempOut()* is used to obtain current temperature value. The line 12 indicates that the current temperature value is stored in the register ADC10MEN and is transfered to a variable *t.* Therefore, the variable *t* is the current temperature value. However, as it is previously presented, all values inside the microcontrollers are under a binary form. At the same time, any sensors have the errors. In this situation, variable *t* is

a binary and an un-calibrated value. The purpose of line 14 is to transfer this binary and un-calibrated value to a readable and an accurate temperature value.

These two function blocks are preparations of using the temperature sensor. After that, the *main()* function is applied to complete its using. A local variable *temp* obtains the result from the function *tempOut()*. Thus, we can simply visit this variable to get the current temperature value.

This experiment is the simplest in these experiments. It is an example of how to use the ADC10.

f. Experiment 6: UART Communication

In the previous experiments, we introduced the microcontroller input and output controlling, using a timer and the interrupt controlling, the ADC application and the system clock configuration. However, all of these are operated inside the microcontroller. The problem is, for example in the experiment 5, how to transfer the ADC values to your PC. That is how to display temperature value automatically on the PC screen. This is a quite important issue in application because it not only transfers temperature value but also transfers other information. It can display details to the user. Therefore, it is the meaning of the communication.

In this experiment, an important communication UART (Universal Asynchronous Receiver-Transmitter) would be demonstrated. The definition of the *"Universal"* originates from first telegraph systems. In today's communication, most protocol reserves specifics of the first telegraph system. In the telegraph system, when the telegraph key was pressed down, a current would flow in the receiver by pushing a stylus into a strip of paper and leaving a "mark" on it. In today's communication systems, the voltage level signal replaces telegraph key, all kinds of data are encoded by ASCII to a digital signal. Therefore, the data would become a serial digital signal.

The concept of the *"Asynchronous"* indicates *UART's* working characteristic. The *"Asynchronous"* means devices in communication will use their own clock signals. The transmitter will give a start and end signal to the receiver. The data is located between the start signal and end signal. The time interval between every communication is arbitrary. For example, the most common device of the asynchronous communication is

keyboard. The user could type letters any time without any notices. Here is an example of sending a letter “D”. In ASCII, the letter “D” is 0x44. The binary format is 1000100. Therefore, the UART transmission of letter “D” is illustrated in figure 2-33.

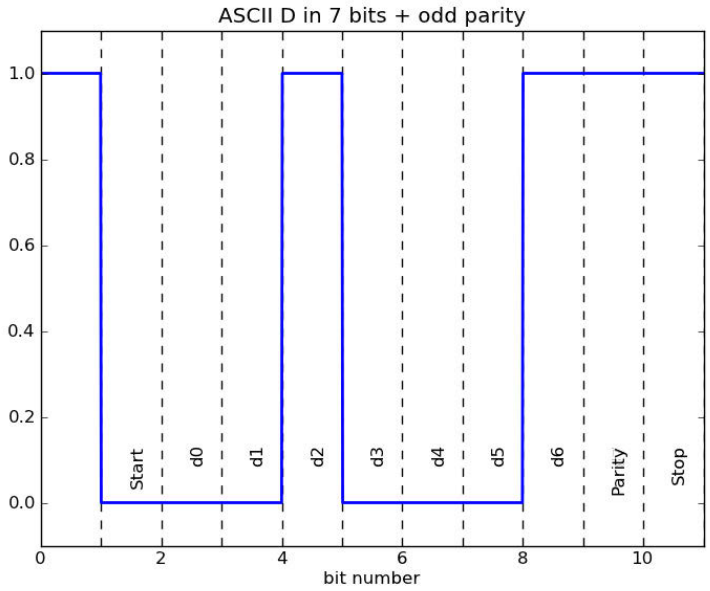


Figure 2-33 Format of “D”

It is obviously found that the transmission starts from the least significant bit to the most significant bit. And also, the start bit is “High to Low” while the parity and stop bits are “High”. All transmission form must follow this in the UART communication.

This experiment will combine all knowledge that introduced in the former experiments. At the same time, this experiment will also introduce a new concept that is the UART simulation. The Launchpad has no UART communication module. Therefore, if we want to use UART communication, we should use its pins and timer as transmitter/receiver and clock. That is the UART simulation. The code of this experiment is demonstrated in figure 2-34.

```c
//*************************************************************
// Lab7.c  Software UART
//
//
//*************************************************************

#include <msp430g2231.h>

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR    TIMERA1_VECTOR
#define TIMER0_A0_VECTOR    TIMERA0_VECTOR
#endif

#define TXD BIT1                            // TXD on P1.1
#define RXD BIT2                            // RXD on P1.2
#define Bitime 13*4                         // 0x0D

unsigned int TXByte;
unsigned char BitCnt;
volatile long tempRaw;
volatile long tempSet = 0;
volatile int i;

unsigned int TxHI[]={0x48,0x49,0x0A,0x08,0x08};
unsigned int TxLO[]={0x4C,0x4F,0x0A,0x08,0x08};
unsigned int TxIN[]={0x49,0x4E,0x0A,0x08,0x08};

void FaultRoutine(void);
void ConfigWDT(void);
void ConfigClocks(void);
void ConfigPins(void);
void ConfigADC10(void);
void ConfigTimerA2(void);
void Transmit(void);

void main(void)
{
  ConfigWDT();
  ConfigClocks();
  ConfigPins();
  ConfigADC10();
  ConfigTimerA2();

  while(1)
  {
   _bis_SR_register(LPM3_bits + GIE);       // turn on interrupts and LPM3
   if (tempSet == 0)
   {
    tempSet = tempRaw;                      // Set reference temp
   }
   if (tempSet > tempRaw + 5)               // test for lo
   {
    P1OUT = BIT6;                           // green LED on
    P1OUT &= ~BIT0;                         // red LED off
    for (i=0;i<5;i++)
     {
      TXByte = TxLO[i];
      Transmit();
     }
   }
```

Figure 2-34 UART Simulation 1

```
61    if (tempSet < tempRaw - 5)                // test for hi
62    {
63     P1OUT = BIT0;                            // red LED on
64     P1OUT &= ~BIT6;                          // green LED off
65     for (i=0;i<5;i++)
66      {
67       TXByte = TxHI[i];
68       Transmit();
69      }
70    }
71    if (tempSet <= tempRaw + 2 & tempSet >= tempRaw - 2)
72    {                                         // test for in range
73     P1OUT &= ~(BIT0 + BIT6);                 // both LEDs off
74     for (i=0;i<5;i++)
75      {
76       TXByte = TxIN[i];
77       Transmit();
78      }
79     }
80   }
81 }
82
83 void ConfigWDT(void)
84 {
85  WDTCTL = WDT_ADLY_250;                      // <1 sec WDT interval
86  IE1 |= WDTIE;                               // Enable WDT interrupt
87 }
88
89 void ConfigClocks(void)
90 {
91 if (CALBC1_1MHZ ==0xFF || CALDCO_1MHZ == 0xFF)
92   FaultRoutine();                           // If calibration data is erased
93                                             // run FaultRoutine()
94  BCSCTL1 = CALBC1_1MHZ;                     // Set range
95  DCOCTL = CALDCO_1MHZ;                      // Set DCO step + modulation
96  BCSCTL3 |= LFXT1S_2;                       // LFXT1 = VLO
97  IFG1 &= ~OFIFG;                            // Clear OSCFault flag
98  BCSCTL2 = 0;                               // MCLK = DCO = SMCLK
99 }
100
101 void FaultRoutine(void)
102 {
103   P1OUT = BIT0;                            // P1.0 on (red LED)
104   while(1);                                // TRAP
105 }
106
107 void ConfigPins(void)
108 {
109  P1SEL |= TXD + RXD;                       // P1.1 & 2 TA0, rest GPIO
110  P1DIR = ~(BIT3 + RXD);                    // P1.3 input, other outputs
111  P1OUT = 0;                                // clear output pins
112  P2SEL = ~(BIT6 + BIT7);                   // P2.6 and 7 GPIO
113  P2DIR |= BIT6 + BIT7;                     // P1.6 and 7 outputs
114  P2OUT = 0;                                // clear output pins
115
116 }
```

Figure 2-35 UART Simulation 2

```
118 void ConfigADC10(void)
119 {
120   ADC10CTL1 = INCH_10 + ADC10DIV_0;         // Temp Sensor ADC10CLK
121 }
122
123 void ConfigTimerA2(void)
124  {
125   CCTL0 = OUT;                              // TXD Idle as Mark
126   TACTL = TASSEL_2 + MC_2 + ID_3;           // SMCLK/8, continuous mode
127
128  }
129
130 // WDT interrupt service routine
131 #pragma vector=WDT_VECTOR
132 __interrupt void WDT(void)
133 {
134  ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON;
135  _delay_cycles(500);                        // Wait for ADC Ref to settle
136  ADC10CTL0 |= ENC + ADC10SC;                // Sampling and conversion start
137  _delay_cycles(100);
138  ADC10CTL0 &= ~ENC;                         // Disable ADC conversion
139  ADC10CTL0 &= ~(REFON + ADC10ON);           // Ref and ADC10 off
140  tempRaw = ADC10MEM;                        // Read conversion value
141  _bic_SR_register_on_exit(LPM3_bits);       // Clr LPM3 bits from SR on exit
142 }
143
144 // Function Transmits Character from TXByte
145 void Transmit()
146 {
147  BitCnt = 0xA;                              // Load Bit counter, 8data + ST/SP
148  while (CCR0 != TAR)                        // Prevent async capture
149    CCR0 = TAR;                              // Current state of TA counter
150  CCR0 += Bitime;                            // Some time till first bit
151  TXByte |= 0x100;                           // Add mark stop bit to TXByte
152  TXByte = TXByte << 1;                      // Add space start bit
153  CCTL0 =  CCIS0 + OUTMOD0 + CCIE;           // TXD = mark = idle
154  while ( CCTL0 & CCIE );                    // Wait for TX completion
155 }
156
157 // Timer A0 interrupt service routine
158 #pragma vector=TIMER0_A0_VECTOR
159 __interrupt void Timer_A (void)
160 {
161  CCR0 += Bitime;                            // Add Offset to CCR0
162  if (CCTL0 & CCIS0)                         // TX on CCI0B?
163  {
164    if ( BitCnt == 0)
165    {
166      CCTL0 &= ~ CCIE ;                      // All bits TXed, disable interrupt
167    }
168
169    else
170    {
171      CCTL0 |=  OUTMOD2;            // TX Space
172      if (TXByte & 0x01)
173      CCTL0 &= ~ OUTMOD2;           // TX Mark
174      TXByte = TXByte >> 1;
175      BitCnt --;
176    }
177  }
```

Figure 2-36 UART Simulation 3

In this code, this project is consisted of 8 function blocks. The first function block is the *FaultRoutine()*. This function block is located from line 101 to line 105. It is used as a protection function that when an error is occurred, the system will call this function in order to prevent possible damages. In this function, there are only 2 codes inside. Line 103 is used to light the red LED and line 104 is used to enter a countless cycle. That is,

when this function is called, the red LED is lighted and system will enter a loop and stop any work.

The second function block is the *ConfigWDT()*. It is located from the line 83 to line 87. In the experiment 1, it briefly introduces the functionality of *Watchdog Timer,* while in this experiment, another function would be described. The Watchdog Timer can be configured as an interval timer and it can generate interrupts at selected time intervals like normal timer. The line 85 is an abbreviation of *"WDTCTL = WDTPW + WDTTMSEL + WDTCNTCL + WDTSSEL + WDTIS0"*. This code indicates that the Watchdog Timer produces interrupt signal every 250ms. That is, every 250ms, system will reach to the line 131 to find interrupt vector. In this experiment, the Watchdog Timer is working like the TimerA in the former experiment. It is used to control the ADC's work.

The third function block is the *ConfigClocks().* This block is located from the line 89 to the line 99. This block configures the clocks system. All clocks should work under 1MHz and XT1 will work under 12KHz. This function unifies the ADC's clock and the Timer's clock. All of them are working under 1MHz.

The fourth one is the *ConfigPins().* This function is located from the line 107 to the line 116. This block configures the pin1.1 and the pin1.2 as the TimerA's output. The pin1.1 is redefined to a new name of TXD which is used to output data and the pin1.2 is redefined to RXD which is used to receive data.

The fifth one is the *ConfigADC10*. This block is located from the line 118 to the line 121. It is used to select a temperature sensor and set the working clock to 1MHz. The *Watchdog Timer* interrupt is adopted in the ADC sampling (line 131 to line 142).

The sixth block is the *ConfigTimerA2().* It is located from the line 123 to the line 128. In the pin configuration, the pin TXD and RXD is configured as the TimerA's port. And the line 125 is used to set these two ports to output 1 as default. The line 126 configures the ADC to work under continuous mode and 1/8 * 1MHz working frequency.

The last two function blocks are the *Transmit()* and the *main()*. Because of their close relationship, they should be introduced together. 3 global integer arrays play vital role in the *main()* function. Each array has 5 elements inside. These numbers are ASCII code. These arrays are used to transfer data to PC and display on the screen. If the temperature value which is obtained by the sensor is higher than "5", the screen will display *"HI"* and microcontroller lights the red LED. If the temperature value is lower than "-5", the screen will display *"LO"* and the microcontroller lights the green LED. If the temperature value stays between "-2" and "2", the screen will display *"IN"* and both LEDs off.

However, how can the *"HI" "LO" "IN"* transfer from the microcontroller to the PC screen. The function *Transmit()* can achieve this. This function is located from the line 145 to the line 155. The line 147 defines that there are 10 bits in each transfer mission. And then, the system enters a *while loop*. The *CCR0* and the *TAR* are registers of the *TimerA*. *CCR0* is a counter as mentioned previously and the *TAR* is a storage register of number. If the value in the *CCR0* reaches to the value in the *TAR*, the *TimerA* will produce an interrupt signal and the system will jump into the interrupt vector. Therefore, the line 149 is an initialization of these two register. The line 150 is used to set the period of each transfer mission. The time is 52 clocks. The line 151 and the line 152 are used to code the data. The line 151 adds a stop bit while the line 152 adds a start bit. The principle of this code work is "or" operation that compares to 000100000000b and a left shift operation. Here is an example of coding a letter "D". The "D" in ASCII is 44 in hex and 1000100 in binary. After the line 151 operation, this one becomes 0101000100. And after the line 152 operation, this one becomes 1010001000. Therefore, this format satisfied the UART protocol. The transmission will start from the last 0 to the first 1 which is shown in figure 33. After that, the system reaches to the line 153. The line 153 is used to call the *TimerA* interrupt vector. This vector is located from the line 158 to the line 177. This interrupt is mainly used to transfer the data bit by bit. The method is using the register *CCTL0* output mode as a transmitter. The *CCTL0* will compare to the coded value bit by bit and then send it. This loop will last for 10 times (the bits of each character is 10).

This experiment composes all knowledge of these experiments. The UART simulation is actually achieved through the *TimerA* output mode. As long as the output serial data

is consistent with UART protocol, the communication is viable. According to this, it is possible to use normal pin output to achieve the UART communication as well. This should be quite simpler than using the *TimerA*.

2.1.9 Summary of Microcontroller

This part mainly introduces microcontroller structure. The basic structure includes the Central processing unit (CPU), the Random Access Memory (RAM), the Read Only Memory (ROM), the Timers and the Counters, the Interrupt Controls, the Input/output ports, the Analog to digital converters, the Digital analog converters. In the last part of this part, some related experiments are demonstrated and analysed in detail. These experiments are the technical guides of our IMU project in later chapters.

2.2 Principles of Accelerometer Calibration

2.2.1 Introduction

The accelerometer calibration can be divided into an absolute calibration and a relative calibration. The absolute calibration is to use a gravitational accelerometer as a reference while a relative calibration uses "1g" as its unit. As it is well known, the earth's gravitational field directs to ground is in the unit of ms^{-2}. Although the earth gravitational field is usually regarded as $9.81ms^{-2}$, actually there is 0.7% difference between the minimum to the maximum over the earth's surface. The gravitational field near North Pole is 9.832 ms^{-2} while it is only 9.763 ms^{-2} near equator. Therefore, it is not convenient to use gravitational acceleration as the calibration rule. Most calibration adopts the relative calibration.

2.2.2 6-parameter Calibration Principle

A standard calibration model in factory is through a six-parameter calibration. This calibration relates the outputs of the ADC value to an accelerometer output with unit "g" by a linear model and three channel gains and offsets:

$$NG_f = N \cdot \begin{pmatrix} G_{fx} \\ G_{fy} \\ G_{fz} \end{pmatrix} = \begin{pmatrix} P_{xx} & 0 & 0 \\ 0 & P_{yy} & 0 \\ 0 & 0 & P_{zz} \end{pmatrix} \begin{pmatrix} ADC_x \\ ADC_y \\ ADC_z \end{pmatrix} + \begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix} \quad (2\text{-}12)$$

The N is a dividing number that depends on the features of the sensor. This thesis uses the max value of the ADC output as N. G_f is an output with unit "g". And therefore, the NG_f could be written as equations:

$$NG_{fx} = p_{xx}ADC_x + q_x \tag{2-13}$$

$$NG_{fy} = p_{yy}ADC_y + q_y \tag{2-14}$$

$$NG_{fz} = p_{zz}ADC_z + q_z \tag{2-15}$$

A given origination could be used for every data point. Each point could obtain 3 ADC values in three channels (ADC_x, ADC_y, ADC_z). A given origination could ensure $G_f$in three channels. Generally, the two measurement originations should equal to $g/\sqrt{3}$ and $-g/\sqrt{3}$ in each axis. Thus, for example, define ADC_{x0} as the first measurement origination output and define ADC_{x1} as the second measurement origination output. The equations in three axes are:

$$p_{xx}ADC_{x0} + q_x = \frac{g}{\sqrt{3}} \tag{2-16}$$

$$p_{xx}ADC_{x1} + q_x = \frac{-g}{\sqrt{3}} \tag{2-17}$$

$$p_{yy}ADC_{y0} + q_y = \frac{g}{\sqrt{3}} \tag{2-18}$$

$$p_{yy}ADC_{y1} + q_y = \frac{-g}{\sqrt{3}} \tag{2-19}$$

$$p_{zz}ADC_{z0} + q_z = \frac{g}{\sqrt{3}} \tag{2-20}$$

$$p_{zz}ADC_{z1} + q_z = \frac{-g}{\sqrt{3}} \tag{2-21}$$

At the same time, the gain and offset are:

$$p_{xx} = \frac{2N}{\sqrt{3}(ADC_{x0} - ADC_{x1})} \tag{2-22}$$

$$p_{yy} = \frac{2N}{\sqrt{3}(ADC_{y0} - ADC_{y1})} \tag{2-23}$$

$$p_{zz} = \frac{2N}{\sqrt{3}(ADC_{z0} - ADC_{z1})} \tag{2-24}$$

$$q_x = \frac{-N(ADC_{x0}+ADC_{x1})}{\sqrt{3}(ADC_{x0}-ADC_{x1})} \quad (2\text{-}25)$$

$$q_y = \frac{-N(ADC_{y0}+ADC_{y1})}{\sqrt{3}(ADC_{y0}-ADC_{y1})} \quad (2\text{-}26)$$

$$q_z = \frac{-N(ADC_{z0}+ADC_{z1})}{\sqrt{3}(ADC_{z0}-ADC_{z1})} \quad (2\text{-}27)$$

The advantage of this method is only 2 measurement originations while the disadvantage is that it requires other mechanical device assistance. Similarly, there is another calibration method may be simpler. This method requires the measurement origination to equal to 1g and -1g instead of $g/\sqrt{3}$ and $-g/\sqrt{3}$. Therefore the equations are:

$$p_{xx}ADC_{x0} + q_x = g \quad (2\text{-}28)$$

$$p_{xx}ADC_{x1} + q_x = -g \quad (2\text{-}29)$$

$$p_{yy}ADC_{y0} + q_y = g \quad (2\text{-}30)$$

$$p_{yy}ADC_{y1} + q_y = -g \quad (2\text{-}31)$$

$$p_{zz}ADC_{z0} + q_z = g \quad (2\text{-}32)$$

$$p_{zz}ADC_{z1} + q_z = -g \quad (2\text{-}33)$$

Then the gain and offset become:

$$p_{xx} = \frac{N}{(ADC_{x0}-ADC_{x1})} \quad (2\text{-}34)$$

$$p_{yy} = \frac{N}{(ADC_{y0}-ADC_{y1})} \quad (2\text{-}35)$$

$$p_{zz} = \frac{N}{(ADC_{z0}-ADC_{z1})} \quad (2\text{-}36)$$

$$q_x = \frac{-N(ADC_{x0}+ADC_{x1})}{(ADC_{x0}-ADC_{x1})} \quad (2\text{-}37)$$

$$q_y = \frac{-N(ADC_{y0}+ADC_{y1})}{(ADC_{y0}-ADC_{y1})} \quad (2\text{-}38)$$

$$q_z = \frac{-N(ADC_{z0}+ADC_{z1})}{(ADC_{z0}-ADC_{z1})} \quad (2\text{-}39)$$

2.2.3 Optimal Measurement Originations

In the previous discussion, there are two originations for measuring. One is making $\frac{g}{\sqrt{3}}$ in all three x, y, z channels and another one is $\frac{-g}{\sqrt{3}}$. At the same time, another 6 origination measurement is also processed. This gives each channel the $-g$ and the g respectively. However, which origination is the best? The measurement origination should satisfy those two principles:

- Each gravity field vector should separate to each other maximally.
- Any pairs of measurement originations should separate to each other maximally.

In order to mathematically analyse those originations, use a smart phone as an example to explain origination angles.

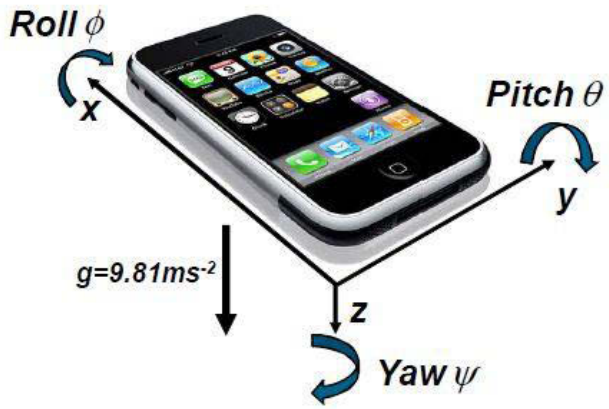


Figure 2-37 Rotation Angles

Those angles could be adopted to make a rotation matrix. These matrixes can represent gravity accelerometer as well.

$$R_x(\psi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\phi & sin\phi \\ 0 & -sin\phi & cos\phi \end{pmatrix} \quad (2\text{-}40)$$

$$R_y(\theta) = \begin{pmatrix} cos\theta & 0 & -sin\theta \\ 0 & 1 & 0 \\ sin\theta & 0 & cos\theta \end{pmatrix} \quad (2\text{-}41)$$

$$R_z(\psi) = \begin{pmatrix} cos\psi & sin\psi & 0 \\ -sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2-42}$$

$$G = \begin{pmatrix} G_x \\ G_y \\ G_z \end{pmatrix} = R_x(\psi) \cdot R_y(\theta) \cdot R_z(\psi) \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -sin\theta \\ cos\theta sin\phi \\ cos\theta cos\phi \end{pmatrix} \tag{2-43}$$

Therefore, here several Optimum orientations that could be referenced in calibration.

Table 2-2 Optimum Orientations for Two Measurements

Origination 0	Origination 1	
$\theta[0] = -35°$ $\phi[0] = 45°$ $G[0]$ $= \begin{pmatrix} -sin(-35°) \\ cos(-35°)sin45° \\ cos(-35°)cos45° \end{pmatrix}$	$\theta[1] = 35°$ $\phi[1] = -135°$ $G[1]$ $= \begin{pmatrix} -sin(35°) \\ cos(35°)\sin(-135°) \\ cos(35°)\cos(-135°) \end{pmatrix}$	
$Y_x = \begin{pmatrix} -sin(-35°) \\ -sin(35°) \end{pmatrix}$	Y_y $= \begin{pmatrix} \cos(-35°)sin(45°) \\ \cos(35°)sin(-135°) \end{pmatrix}$	Y_y $= \begin{pmatrix} \cos(-35°)cos(45°) \\ \cos(35°)sin(-135°) \end{pmatrix}$

Table 2-3 Optimum Orientations for Three Measurements

Origination 0	Origination 1	Origination 2
$\theta[0] = 0°$ $\phi[0] = 45°$ $G[0]$ $= \begin{pmatrix} -sin(0°) \\ cos(0°)sin45° \\ cos(0°)cos45° \end{pmatrix}$	$\theta[1] = -45°$ $\phi[1] = 180°$ $G[1]$ $= \begin{pmatrix} -sin(-45°) \\ cos(-45°)\sin(180°) \\ cos(-45°)\cos(180°) \end{pmatrix}$	$\theta[0] = 45°$ $\phi[0] = -90°$ $G[2]$ $= \begin{pmatrix} -sin(45°) \\ cos(45°)\sin(-90°) \\ cos(45°)\cos(-90°) \end{pmatrix}$
$Y_x = \begin{pmatrix} -sin(0°) \\ -sin(-45°) \\ -sin(45°) \end{pmatrix}$	Y_y $= \begin{pmatrix} cos(0°)sin45° \\ cos(-45°)\sin(180°) \\ cos(45°)\sin(-90°) \end{pmatrix}$	Y_z $= \begin{pmatrix} cos(0°)cos45° \\ cos(-45°)\cos(180°) \\ cos(45°)\cos(-90°) \end{pmatrix}$

Table 2-4 Optimum Orientations for Four Measurements

Origination 0	Origination 1	Origination 2
$\theta[0] = 39°$ $\phi[0] = -158°$ $G[0]$ $= \begin{pmatrix} -sin(39°) \\ cos(39°)\sin(-158°) \\ cos(39°)\cos(-158°) \end{pmatrix}$	$\theta[1] = -66°$ $\phi[1] = 164°$ $G[1]$ $= \begin{pmatrix} -sin(-66°) \\ cos(-66°)\sin(164°) \\ cos(-66°)\cos(164°) \end{pmatrix}$	$\theta[2] = 18°$ $\phi[2] = 66°$ $G[2]$ $= \begin{pmatrix} -sin(18°) \\ cos(18°)\sin(66°) \\ cos(18°)\cos(66°) \end{pmatrix}$
Origination 3		
$\theta[3] = -1°$ $\phi[3] = -44°$ $G[3]$ $= \begin{pmatrix} -sin(-1°) \\ cos(-1°)\sin(-44°) \\ cos(-1°)\cos(-44°) \end{pmatrix}$		
$Y_x = \begin{pmatrix} -sin(39°) \\ -sin(-66°) \\ -sin(18°) \\ -sin(-1°) \end{pmatrix}$	Y_y $= \begin{pmatrix} cos(39°)\sin(-158°) \\ cos(-66°)\sin(164°) \\ cos(18°)\sin(66°) \\ cos(-1°)\sin(-44°) \end{pmatrix}$	Y_z $= \begin{pmatrix} cos(39°)\cos(-158°) \\ cos(-66°)\cos(164°) \\ cos(18°)\cos(66°) \\ cos(-1°)\cos(-44°) \end{pmatrix}$

Table 2-5 Optimum Orientations for Six Measurements

Origination 0	Origination 1	Origination 2
$\theta[0] = 6°$ $\phi[0] = -55°$ $G[0]$ $= \begin{pmatrix} -sin(6°) \\ cos(6°)\sin(-55°) \\ cos(6°)\cos(-55°) \end{pmatrix}$	$\theta[1] = -6°$ $\phi[1] = 125°$ $G[1]$ $= \begin{pmatrix} -sin(-6°) \\ cos(-6°)\sin(125°) \\ cos(-6°)\cos(125°) \end{pmatrix}$	$\theta[2] = 20°$ $\phi[2] = -147°$ $G[2]$ $= \begin{pmatrix} -sin(20°) \\ cos(20°)\sin(-147°) \\ cos(20°)\cos(-147°) \end{pmatrix}$
Origination 3	Origination 4	Origination 5
$\theta[3] = -20°$ $\phi[3] = 33°$	$\theta[4] = -69°$ $\phi[4] = -128°$	$\theta[5] = 69°$ $\phi[5] = 52°$

$G[3]$ $= \begin{pmatrix} -sin(-20°) \\ cos(-20°)\sin(33°) \\ cos(-20°)\cos(33°) \end{pmatrix}$	$G[4]$ $= \begin{pmatrix} -sin(-69°) \\ cos(-69°)\sin(-128°) \\ cos(-69°)\cos(-128°) \end{pmatrix}$	$G[5]$ $= \begin{pmatrix} -sin(69°) \\ cos(69°)\sin(52°) \\ cos(69°)\cos(52°) \end{pmatrix}$
$Y_x = \begin{pmatrix} -sin(6°) \\ -sin(-6°) \\ -sin(20°) \\ -sin(-20°) \\ -sin(-69°) \\ -sin(69°) \end{pmatrix}$	Y_y $= \begin{pmatrix} cos(6°)\sin(-55°) \\ cos(-6°)\sin(125°) \\ cos(20°)\sin(-147°) \\ cos(-20°)\sin(33°) \\ cos(-69°)\sin(-128°) \\ cos(69°)\sin(52°) \end{pmatrix}$	Y_z $= \begin{pmatrix} cos(6°)\cos(-55°) \\ cos(-6°)\cos(125°) \\ cos(20°)\cos(-147°) \\ cos(-20°)\cos(33°) \\ cos(-69°)\cos(-128°) \\ cos(69°)\cos(52°) \end{pmatrix}$

Table 2-6 Optimum Orientations for Eight Measurements

Origination 0	Origination 1	Origination 2
$\theta[0] = -35°$ $\phi[0] = -45°$ $G[0]$ $= \begin{pmatrix} -sin(-35°) \\ cos(-35°)\sin(-45°) \\ cos(-35°)\cos(-45°) \end{pmatrix}$	$\theta[1] = -73°$ $\phi[1] = 161°$ $G[1]$ $= \begin{pmatrix} -sin(-73°) \\ cos(-73°)\sin(161°) \\ cos(-73°)\cos(161°) \end{pmatrix}$	$\theta[2] = 5°$ $\phi[2] = 17°$ $G[2]$ $= \begin{pmatrix} -sin(5°) \\ cos(5°)\sin(17°) \\ cos(5°)\cos(17°) \end{pmatrix}$
Origination 3	Origination 4	Origination 5
$\theta[3] = -16°$ $\phi[3] = 84°$ $G[3]$ $= \begin{pmatrix} -sin(-16°) \\ cos(-16°)\sin(84°) \\ cos(-16°)\cos(84°) \end{pmatrix}$	$\theta[4] = 16°$ $\phi[4] = -96°$ $G[4]$ $= \begin{pmatrix} -sin(16°) \\ cos(16°)\sin(-96°) \\ cos(16°)\cos(-96°) \end{pmatrix}$	$\theta[5] = -5°$ $\phi[5] = -163°$ $G[5]$ $= \begin{pmatrix} -sin(-5°) \\ cos(-5°)\sin(-163°) \\ cos(-5°)\cos(-163°) \end{pmatrix}$
Origination 6	Origination 7	
$\theta[6] = 73°$ $\phi[6] = -18°$ $G[6]$ $= \begin{pmatrix} -sin(73°) \\ cos(73°)\sin(-18°) \\ cos(73°)\cos(-18°) \end{pmatrix}$	$\theta[7] = 35°$ $\phi[7] = 135°$ $G[7]$ $= \begin{pmatrix} -sin(35°) \\ cos(35°)\sin(135°) \\ cos(35°)\cos(135°) \end{pmatrix}$	

$Y_x = \begin{pmatrix} -sin(-35°) \\ -sin(-73°) \\ -sin(5°) \\ -sin(-16°) \\ -sin(16°) \\ -sin(-5°) \\ -sin(73°) \\ -sin(35°) \end{pmatrix}$	$Y_y = \begin{pmatrix} cos(-35°)\sin(-45°) \\ cos(-73°)\sin(161°) \\ cos(5°)\sin(17°) \\ cos(-16°)\sin(84°) \\ cos(16°)\sin(-96°) \\ cos(-5°)\sin(-163°) \\ cos(73°)\sin(-18°) \\ cos(35°)\sin(135°) \end{pmatrix}$	$Y_z = \begin{pmatrix} cos(-35°)\cos(-45°) \\ cos(-73°)\cos(161°) \\ cos(5°)\cos(17°) \\ cos(-16°)\cos(84°) \\ cos(16°)\cos(-96°) \\ cos(-5°)\cos(-163°) \\ cos(73°)\cos(-18°) \\ cos(35°)\cos(135°) \end{pmatrix}$

2.2.4 Linear Least Square Method

Linear Least Squares Method is often used to find the linear relationship between two variables x and y. Assume data is in the form of (x_n, y_n) for $n\epsilon\{1,2,3,\dots,N\}$. A researcher often needs experiments to determine their relationship. For example, the relationship between x and y through experiments is:

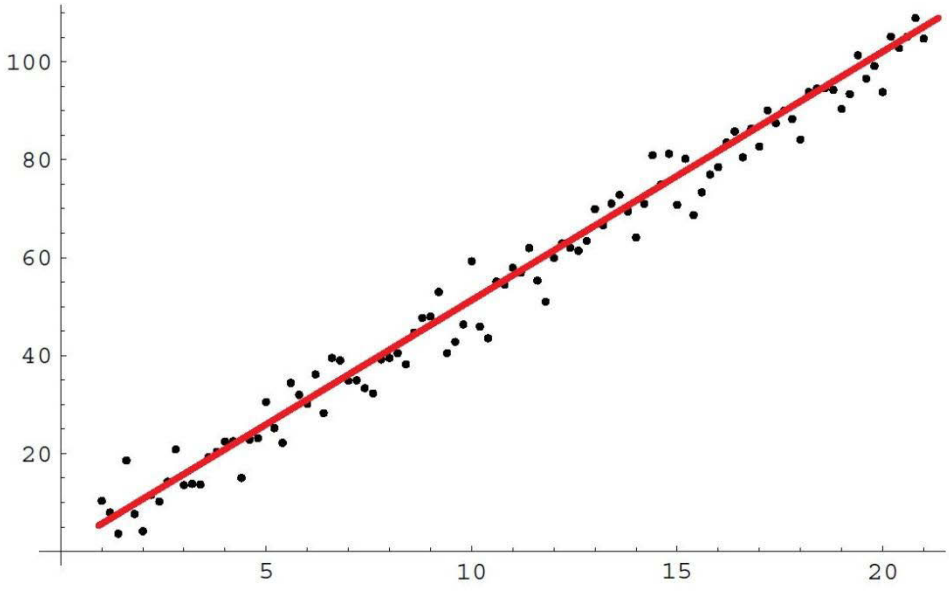


Figure 2-38 Line Fitting

It is impossible to obtain a perfect linear relationship because of the experiment error and the overestimating of their linearity. The Linear Least Squares Method is a procedure that to determine the best fit line to the data. Here is a simple introduction of the linear least squares method. Given a serial of data $x_1, x_2, x_3, \dots, x_n$. Define mean value as:

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{2-44}$$

However, the mean value could not be used directly. For example, serial data $\{1,2,3,4,5\}$ and $\{3,3,3,3,3\}$ has the same mean value but the first group has a higher variation. Therefore, the concept of variance is more useful in a quantify data. The variance of $x_1, x_2, x_3, \dots, x_n$ is:

$$\sigma_x^2 = \frac{1}{N}\sum_{n=i}^{N}(x_i - \bar{x})^2 \tag{2-45}$$

Thus, the square root of the variance is:

$$\sigma_x = \sqrt{\frac{1}{N}\sum_{n=i}^{N}(x_i - \bar{x})^2} \tag{2-46}$$

The square root of the variance has a good measurement of the deviation around x's mean value. Of course, there are some other alternate measurements. For example:

$$\frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x}) \tag{2-47}$$

$$\frac{1}{N}\sum_{n=1}^{N}|x_n - \bar{x}| \tag{2-48}$$

These two could also be considered as a good measurement. However, the equation 2-47 has a signed quantity. For example, a large number of negative values could cancel a small number of positive values. And also, it may be zero if the positive values cancel the negative values perfectly. The equation 2-48 also has problems. It is difficult to analyse. Therefore, the equation 45 is the best solution in mathematical analysis.

After the analysis of the equations 46,47, 48, it is clear that square root of the variance plays an important role in the linear least squares method. The linear least square method is finding the best fit line. Therefore, if there is a linear relationship$y = ax + b$, and $y - (ax + b)$ should be zero. However, as the thesis discussed previously, it is impossible to obtain a perfect linear relationship because of the experiment error and the overestimating of their linearity. Thus, in reality, the experiment data $y - (ax + b)$ may not be zero. This value is defined as the error e. Therefore, a group of experiment data $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_n, y_n)\}$ should have their error:

$$\{(y_1 - (ax_1 + b)), (y_2 - (ax_2 + b)), \dots, (y_n - (ax_n + b))\} \tag{2-49}$$

The variance is:

$$\sigma^2_{y-(ax+b)} = \frac{1}{N}\sum_{n=1}^{N}(y_n - (ax_n + b))^2 \tag{2-50}$$

It is obviously that this equation is a function $E(a, b)$ about $a\ and\ b$. That is x and y should be obtained by experiment while $a\ and\ b$ are unknown. Therefore, the most central thinking of the linear least squares method is finding $a\ and\ b$ that minimizes $\sigma_{y-(ax+b)}$. This $a\ and\ b$ could be the best fit line.

In order to minimise the error, define multivariable calculus as:

$$\frac{\partial E}{\partial a} = 0, \frac{\partial E}{\partial b} = 0 \tag{2-51}$$

$$\frac{\partial E}{\partial a} = \sum_{n=1}^{N} 2 \cdot (y_n - (ax_n + b)) \cdot (-x_n) = 0 \tag{2-52}$$

$$\frac{\partial E}{\partial b} = \sum_{n=1}^{N} 2 \cdot (y_n - (ax_n + b)) \cdot 1 = 0 \tag{2-53}$$

Re-write the equations 2-52 and 2-53:

$$(\sum_{n=1}^{N} x_n^2)a + (\sum_{n=1}^{N} x_n)b = \sum_{n=1}^{N} x_n y_n \tag{2-54}$$

$$(\sum_{n=1}^{N} x_n)a + (\sum_{n=1}^{N} 1)b = \sum_{n=1}^{N} y_n \tag{2-55}$$

In the matrix form:

$$\begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{n=1}^{N} x_n y_n \\ \sum_{n=1}^{N} y_n \end{pmatrix} \tag{2-56}$$

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix}^{-1} \begin{pmatrix} \sum_{n=1}^{N} x_n y_n \\ \sum_{n=1}^{N} y_n \end{pmatrix} \tag{2-57}$$

Here, it is necessary to discuss about whether matrix $\begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix}$ is invertible.

Denote $\begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix}$ by M. The determinant of *M* is:

$$\det(M) = \sum_{n=1}^{N} x_n^2 \cdot \sum_{n=1}^{N} 1 - \sum_{n=1}^{N} x_n \cdot \sum_{n=1}^{N} x_n \tag{2-58}$$

Because

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n,$$

Therefore

$$\det(M) = N \cdot \sum_{n=1}^{N} x_n^2 - (N\bar{x})^2$$

$$= N^2 \cdot \frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})^2 \quad (2\text{-}59)$$

As long as x_n are not equal to each other, $\det(M)$ is non-zero and M is invertible. Therefore, equation 2-59 would be useful.

2.3 Chapter Conclusion

In this chapter, the basic micro-controller and the calibration knowledge are introduced. A typical micro-controller has the Central processing unit (CPU), the Random Access Memory (RAM), the Read Only Memory (ROM), Timers and Counters, Interrupt Controls, Input/output ports, Analog to digital converters and Digital analog converters. In these components, CPU's responsibility is to handle the data which is the core part of a micro-controller. The Memory Unit is used to store data that comes from different ports. Timers and Counters are used to generate interrupt signals. Ports are the communication components. The ADC and DAC are mostly dealing with sensors operation data.

Calibration part mainly introduces the principle of the accelerometer calibrating work. The classical calibration requires some known orientations as the reference while auto calibration requires a more complex algorithm other than a reference direction.

III. ELECTRONIC COMPONENTS OF IMU

3.1 Introduction

At the beginning of this chapter, there are 4 pictures that show the physical map of our IMU (Inertial Measurement Unit) in order to provide a general understanding of the electronic components. When you open any electronic product, there must be a green board (some may be brown or golden) with a variety of electronic components similar to the IMU inside a protective shell. This board is a PCB (Printed Circuit Board) which is a basic structure of any electronic product. The functionality of this green board is not only to support or fix electronic components but also to connect them to achieve required functions. This "green board" is a final product after several designing and manufacturing processes. Therefore, the best method to understand this IMU is to follow and understand the relevant knowledge of PCBs. This chapter will briefly introduce each electronic component that is integrated on the board.

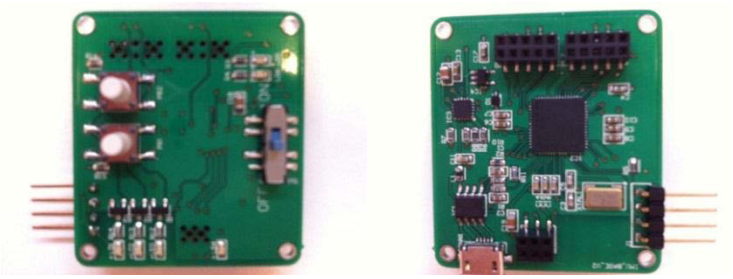


Figure 3-1 Top Board of IMU

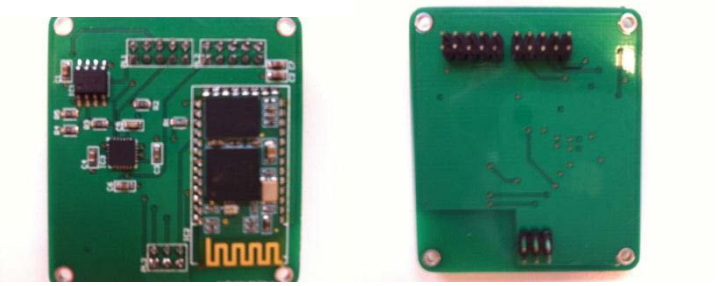


Figure 3-2 Bottom Board of IMU

The first step to make a real PCB is the schematic design. A circuit schematic is developed and analysed with a basic knowledge of the PCB design to ensure the desired

functionality and performance. At the same time, a designer needs to select proper electronic components according to the schematic. It is possible to use the schematic capture software and other related software to simulate the designed circuit without building it on a prototype board. This kind of software allows the PCB designer to create an electronic schematic which contains the information; the symbol of every part in the schematic could contain what the footprint symbol is associated with. Although the symbol for the components in the electronic schematic does not show what the actual physical component looks like, it allows the designer to connect all the components in a circuit and to test the performance of the circuit. Figure 2-3 shows the schematic of the IMU. This schematic illustrates the selection and connection of every component. The software (Altium designer 09) could use the schematic to simulate the functionality of the circuit. In order to understand the functionality of this schematic, every functional module needs to be explained first.

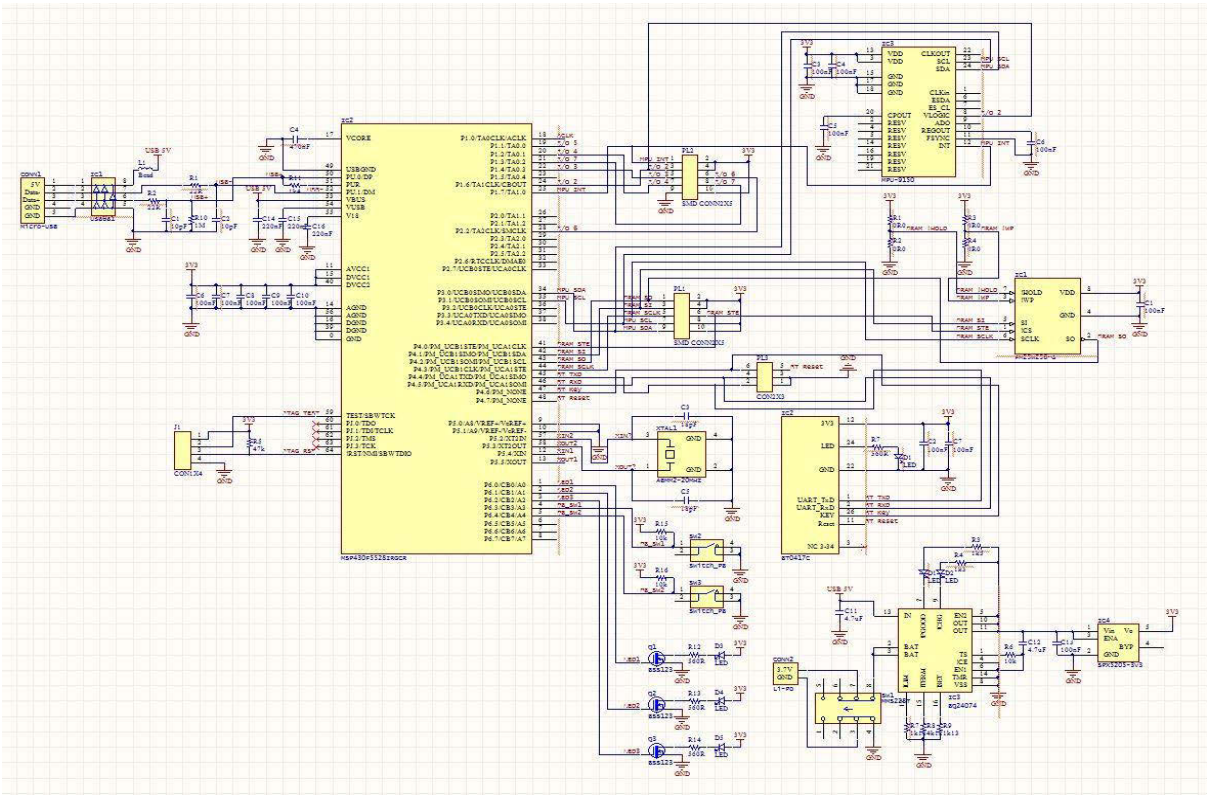


Figure 3-3 The schematic of the IMU

3.2 Power and Battery Management Unit

The most important functional module is the power and battery management unit. The functionality of this unit is to provide stable power which comes from a USB or a

battery to the whole IMU and to manage the charging or discharging cycles of the battery. It can also protect the system from overvoltage damage. In summary, this unit could supply safe power and improve the efficiency of a battery. Figure 3-4 is the schematic of the Power and Battery Management Unit.

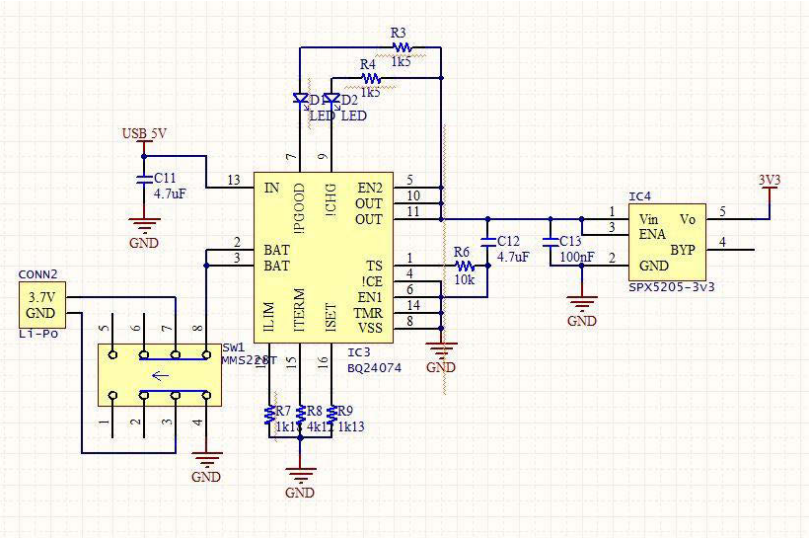


Figure 3-4 Power and Battery Management Unit

This unit consists of 3 parts: a switch *MMS228T*; a 1.5A USB-Friendly Li-Ion battery charger and power–path management *BQ24074*; a 150mA, Low-Noise LDO Voltage Regulator *SPX5205*. The picture and features of the *MMS228T* switch are shown in figure 3-5 and table 3-1 [20]. The functionality will be illustrated in table 3-2.

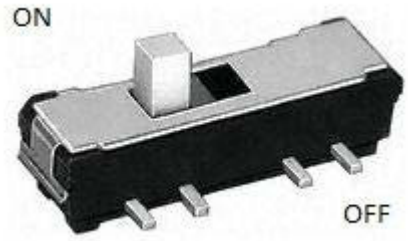


Figure 3-5 MMS228T

Table 3-1 FEATURES OF MMS228T

Contact rating	12 V DC, 200 mA / 6 V DC, 300 mA
Contact resistance	< 70 mΩ
Insulation resistance	> 100 MΩ at 500 V DC
Dielectric strength	500 V, 50 Hz for the duration of 1 minute
Operating temperature	– 10° to + 60° C
Mechanical life	minimum 10 000 operations
Contacts/Terminals	copper alloy, silver-plated
Contact timing	non-shorting

Slide colour	white

Table 3-2 FUNCTIONS OF SWITCH

USB Status	Switch Status	Description	LED5 Status	LED6 Status
Unconnected	OFF	IMU off	OFF	OFF
Unconnected	ON	IMU powered on by battery	OFF	OFF
Connected	OFF	IMU powered on by USB	ON	OFF
Connected	ON	IMU powered on and battery charging (not full)	ON	ON
Connected	ON	IMU powered on and battery charging (but full)	ON	OFF

The switch is only used for controlling the battery work. If the switch is turned on, the battery will work and charge while if the switch is turned off, the battery will stop work.

The second electronic component is the 1.5A USB-Friendly Li-Ion battery charger and power–path management *BQ24074* [17]. This electronic component has a recommended application circuit in its datasheet. Our IMU system modifies this recommended circuit to fit our application (Figure 3-4). Figure 3-7 is the recommended circuit. The purposes of this circuit are: firstly, protecting the charger from an incorrectly configured crashing; the second one is to manage the power supply [17]. That is, this circuit can charge the battery and supply power to the system simultaneously; it also improves the efficiency of the battery by reducing the charging and discharging cycles. The component could extend the battery life.

Figure 3-6 BQ24074

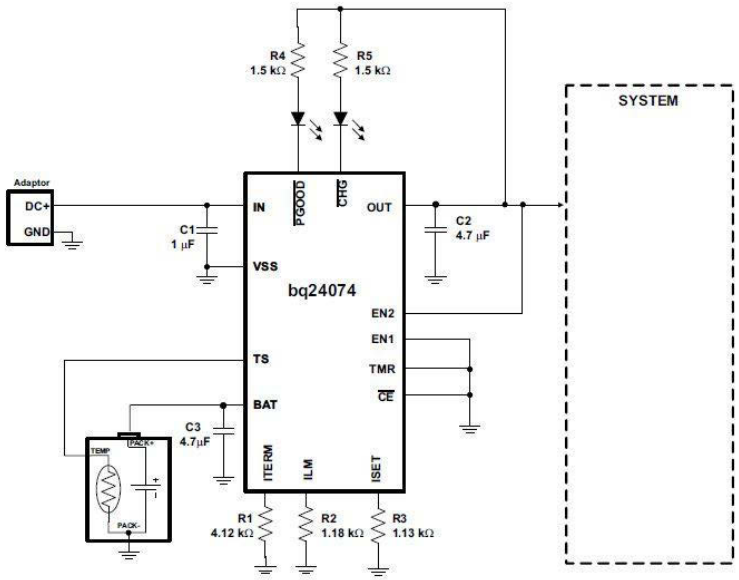


Figure 3-7 Recommended BQ24074 Charger Application

The "system" position above the figure is replaced by another electronic component which is a 150mA, Low-Noise LDO Voltage Regulator *SPX5205* [23]. This device is a positive voltage regulator with a very low dropout voltage and low output noise; it uses a very little quiescent current of 750 μA at 100 mA output load [23]. It has a very high accuracy while its tolerance is less than 1% with a compensated temperature coefficient. Similar to the *BQ24074*, there is also a recommended circuit shown in figure 8 for the *SPX5205* application which is adopted in our IMU. The only purpose of this circuit is to obtain a regulated output voltage. All 3V3 regulated voltage power which is supplied to every electronic component in the IMU comes from the circuit pin V_{out}.

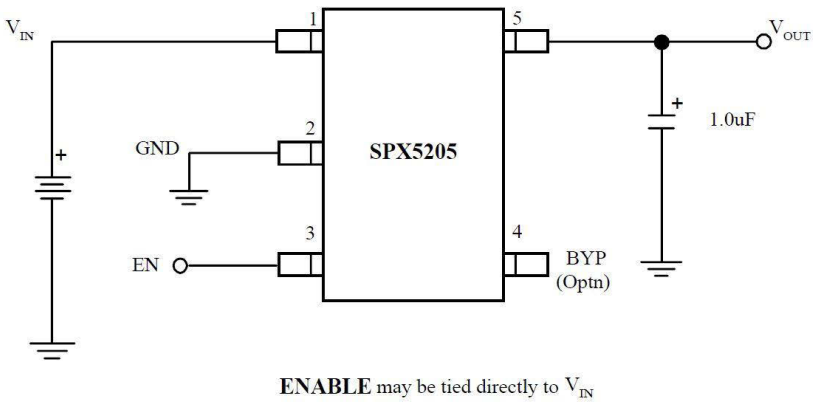


Figure 3-8 Recommended SPX5205 Application

Figure 3-9 SPX5205

In summary, the MMS228T, the BQ24074 and the SPX5205 consist of the Power and Battery Management Unit. The power comes from the USB 5V or the battery; it will be converted to a stable 3V3 power output to the system.

3.3 SMD (Surface Mounted Devices) Connectors and JTAG

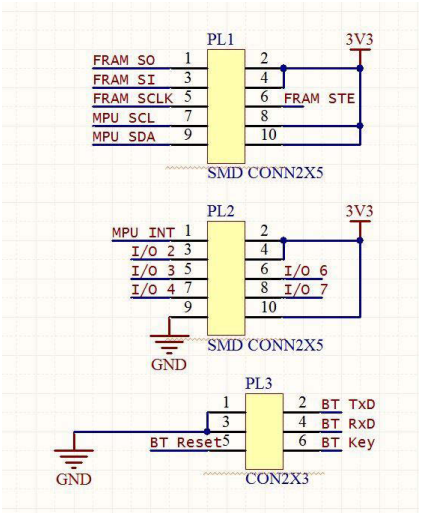


Figure 3-10 SMD connectors

A SMD connector is an electro-mechanical device which plays a role of an interface in the circuit. Connectors often consist of plugs (male-ended) and sockets (female-ended). In our IMU, there are 3 female headers (sockets) on the surface of the top board and 3 pin headers (plugs) on the surface of the bottom board. These 3 pairs of connectors build the connection and communication path between the two boards. The female header and the pin header are illustrated in figure 3-11 and figure 3-12.

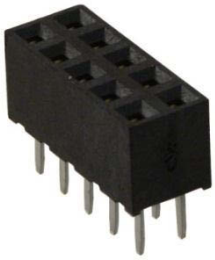


Figure 3-11 2X5 Female Header

Figure 3-12 2X5 Pin Header

The features of the pin header can be found in table 3 and the female header features are shown in table 3-4 [21].

Table 3-3 FEATURES OF PIN HEADER

Part number	MTMM-103-04-GD-B7
Insulator Material	Black Liquid Crystal Polymer
Terminal Material:	Phosphor Bronze
Plating	Sn or Au over 50µ" (1,27 µm) Ni
Operating Temp Range	-55°C to +105°C with Tin; -55°C to +125°C with Gold
RoHS Compliant	Yes
Lead–Free Solderable	Yes

Table 3-4 FEATURES OF FEMALE HEADER

Current Rating	1.5AMP
Operation Temperature	-40℃ to +105℃
Contact Resistance	20mΩ MAX
Insulation Resistance	1000mΩ MIN
Withstanding Voltage	AC 500V
Insulator Material	Polyester(UL94V-O)
Standard	Nylon-6T
Contact Material	Phosphor Bronze
Contact Plating	Au over 50µ" (1,27 µm) Ni

There is another important connector that appears in the left corner of the schematic. This connector is the JTAG (Joint Test Action Group) which is used for an online program debugging or testing in the IMU. Figure 3-13 illustrates the schematic of the JTAG part.

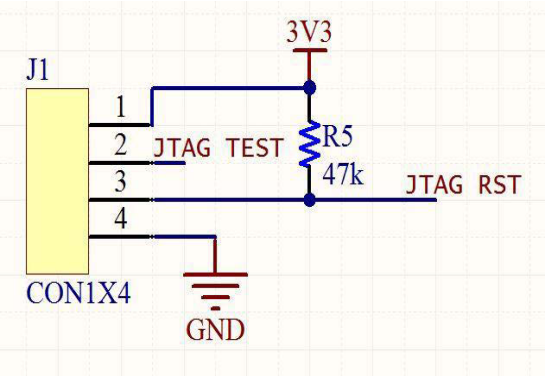


Figure 3-13 JTAG Interface

JTAG, as defined by the IEEE Std.-1149.1 standard, is an integrated component for testing or debugging on the PCBs; it is implemented at the IC (integrated circuit) level due to the inability of the testing and the debugging of highly complex and dense PCBs under the traditional method [19]. The standard JTAG is a 4-wire JTAG interface which requires 4 signals for communication [22]. The features of the 4-wire JTAG are demonstrated in table 3-5.

Table 3-5 FEATURES OF 4-WIRE JTAG

Pin	Direction	Description
TMS	IN	Signal to control the JTAG state machine
TCK	IN	JTAG clock input
TDI	IN	JTAG data input and TCLK input
TDO	OUT	JTAG data output
TEST	IN	Enable JTAG pins (shared JTAG devices only)

However, there is an optimized 2-wire JTAG interface adopted in the IMU because of the physical space constraints [22]. The advantages of a 2-wire JTAG interface are less pins being used and it is compatible with general I/O pins. The disadvantage is that it is slower than the 4-wire JTAG interface. The core JTAG logic which is integrated into the devices supports the 2-wire mode; it is similar to the 4-wire JTAG devices. The fundamental difference between them is that the 2-wire JTAG devices integrate an additional logic circuit which is used to convert the 2-wire communication protocol into the standard 4-wire communication protocol internally. Figure 3-14 demonstrates the details of 2-wire JTAG protocol [22]. The physical map of the JTAG is shown in figure 3-15.

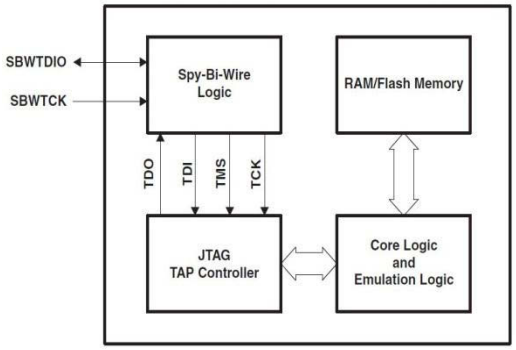


Figure 3-14 2-wire JTAG Protocol

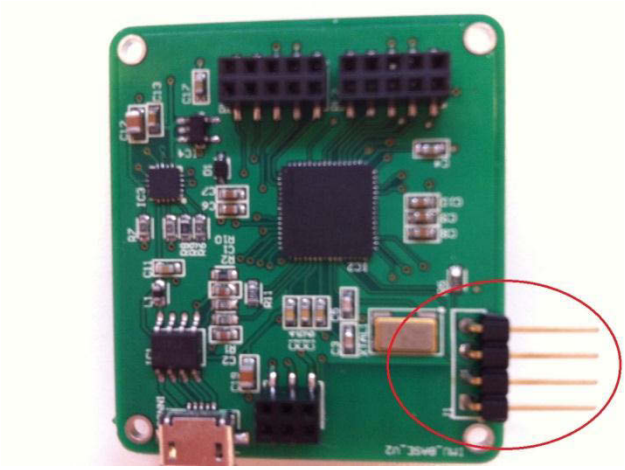


Figure 3-15 2-wire JTAG on the IMU

In summary, this part mainly demonstrates 3 kinds of connectors which are pin headers, female headers and JTAG. The pin headers and the female header are used to build the connection between the top board and the bottom board while the JTAG is used to debug and test.

3.4 XT2 Oscillator

In the MSP430F5 family, a MCU (Micro Control Unit) often has its own oscillator which is named XT1 to supply the system clock signal. The highest frequency of the XT1 oscillator, however, is only 32 kHz which is not useful in the ADC sampling [14]. Therefore, a higher frequency oscillator needs to be integrated on the PCB to provide a reliable sampling frequency. The ABMM2 is used as an external clock signal which adopts 20 MHz frequency on the XT2IN pin [16]. Figure 3-16 is the schematic of this oscillator that is integrated in the IMU. Other features would be illustrated in table 3-6 [16].

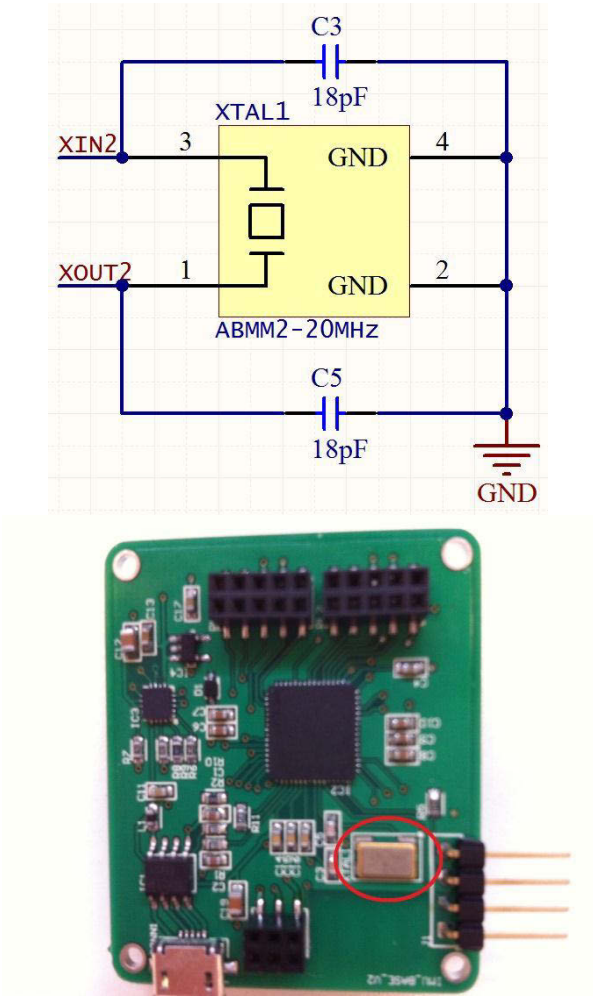


Figure 3-16 ABMM2 Circuit

Table 3-6 FEATURES OF ABMM2 OSCILLATOR

ABRACON P/N	ABMM2 Series
Frequency Range	8.00 MHz - 110.00 MHz
Operation Mode	8.00 MHz ~ 50.00 MHz (Fundamental) 30.01 MHz ~ 100.00 MHz (3rd Overtone) 90.00 MHz ~ 110.00 MHz (5thOvertone)
Operating Temperature	-10°C to + 60°C
Storage Temperature	- 40°C to + 85°C
Frequency Tolerance @ 25°C	± 20 ppm max.
Frequency Stability over Operating Temp. (Ref to +25°C)	± 20 ppm max.
Equivalent Series Resistance	8.0 - 9.9 (Fund) 72 10.0 - 50.0 (Fund) 50 30 - 100 (3rd OT) 60

	90 - 110 (5th O/T) 80
Shunt Capacitance C0	7pF max.
Load Capacitance CL	18pF
Drive Level	500µW max., 100µW correlation
Aging at 25°C First Year	± 2ppm max.
Insulation Resistance	500MΩ min. @ 100Vdc ± 15V

3.5 USB Communication Unit

There is an important module that is used for the communication in the IMU. The USB (Universal Serial Bus) Communication Unit consists of 2 electronic components which are the Micro USB interface and the data line protection unit. Figure 3-17 and figure 3-18 illustrate these two components.

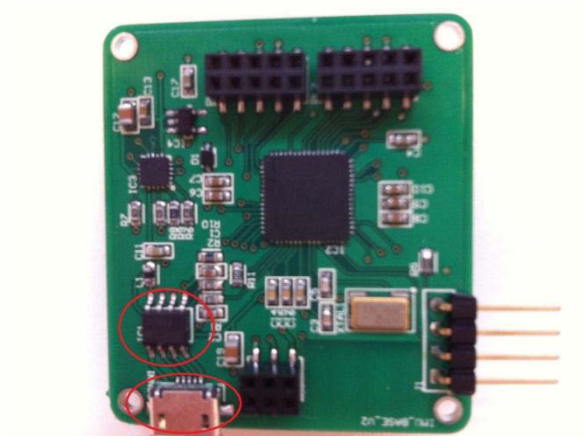


Figure 3-17 Physical map of Micro USB Interface and Data Line Protection Unit

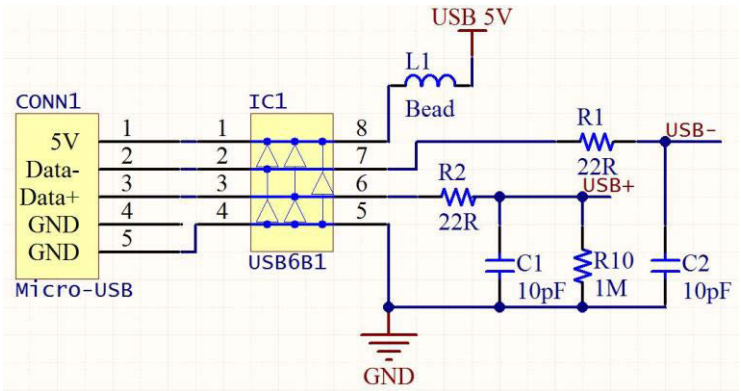


Figure 3-18 Schematic of USB Communication Unit

The USB Communication Unit is powered by the USB5V pin and connected to the MSP430 MCU USB pin 50, 51, 52 for communication. The USB6B1 is the Data Line Protection Unit which is integrated in the middle of the Micro-USB and the MCU in order to prevent the fast transients; this leads to severe damage in a high speed data system [24]. It provides protection for each line between the supply voltage and GND:

25 A, 8/20 µs and separates inputs and outputs to improve ESD (electrostatic-sensitive device) susceptibility [24]. The features are shown in table 3-8. The Micro USB interface is the port that communicates with other devices. Table 3-7 illustrates some Micro USB characters.

Table 3-7 CHARACTERS OF MICRO-USB INTERFACE

Insulator Material	Black LCP
Contact Material	Copper Alloy
Plating	Au over 50µ" (1,27 µm) Ni
Shield Plating	Sn over 50µ" (1,27 µm) Ni
Operating Temp Range	-30°C to +80°C
Voltage Rating	100 VAC
Contact Resistance	50 mΩ
Cycles	10,000

Table 3-8 FEATURES OF USB6B1

V_{pp}	Peak pulse voltage	IEC61000-4-2 contact discharge	8	KV
		IEC61000-4-2 air discharge	15	
		MIL STD883C-Method 30 15-6	4	
P_{pp}	Peak pulse power	8/20 µs	500	W
I_{pp}	Peak pulse current	8/20 µs	25	A
		2/10 µs	40	
T_{stg}	Storage temperature range		- 55 to + 150	℃
T_{op}	Operation Temperature Range		- 40 to + 85	℃
T_L	Lead solder temperature		260	℃
V_{br}	Break down voltage	IR = 1 mA	6	V
I_{RM}	Leakage current	VRM = 5.25 V	10	µA
C	Capacitance between pins D+ and D- V_{OSC}= 30 mV, F = 1 MHz, VR = 0 V	VCC not connected	15	pF
	Capacitance			

	between pins D+(or D-) and GND V_{OSC}= 30 mV, F = 1 MHz, V_R= 5 V	VCC = 5 V	25	pF

3.6 Other Electronic Components

The most functional components have already been introduced in the former parts. This part will briefly introduce the four remaining components: the switch, the Bluetooth, the FRAM storage and the sensor. The Bluetooth, the FRAM and the sensor will be focused on in a later chapter in detail. This part is only a general description.

The switch that used in our IMU is illustrated in figure 3-19 and 3-20.

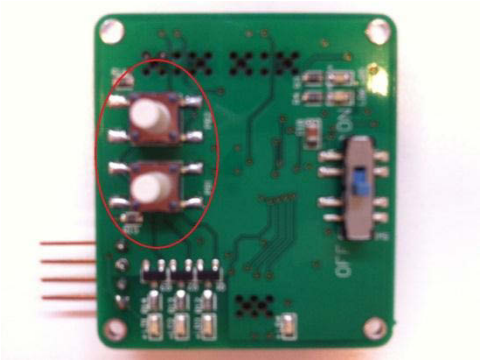


Figure 3-19 2 Switches on IMU

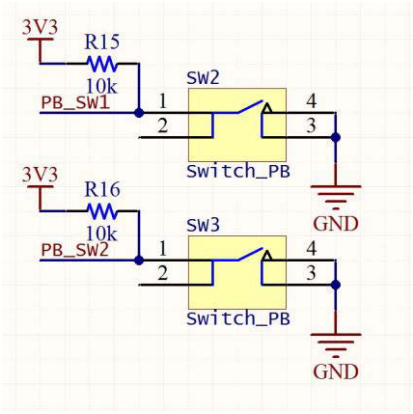


Figure 3-20 Schematic of switches

The functionality of the switch that is used in the IMU is to select an operation mode. The modes are changed by using the buttons; PB1 is the “Calibration Mode Button” and PB2 is the “Alignment Mode Button”. When PB1 is pressed, the IMU will:

- Enter the Calibration Mode if in another mode

- Leave the Calibration Mode back to the default mode if already in the Calibration Mode

This is the same for PB2. Here is a diagram of the state machine, where the PB1 or the PB2 indicates that a particular button has just been pressed:

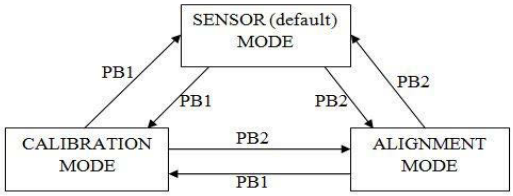


Figure 3-21 Mode Controlling

The Bluetooth that is used in our IMU is the RF-BT0417C. This is a wireless TTL (Transistor-Transistor Logic) Transceiver Module. Figure 3-22 is the physical map and the schematic of the RF-BT0417C. Its characteristics will be listed in table 3-9 [26].

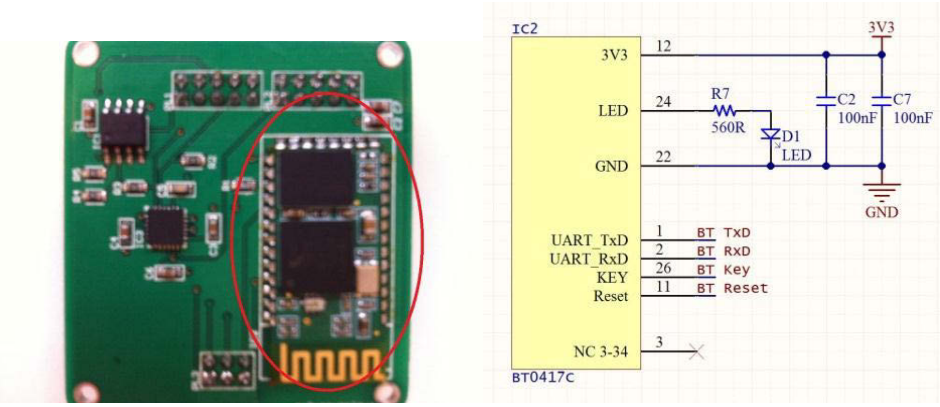


Figure 3-22 RF-BT0417C

Table 3-9 CHARACTERS OF RF-BT0417C

Operating Frequency Band	2.4GHZ-2.48GHZ unlicensed ISM band
Bluetooth Specification	V2.0+EDR
Output Power Class	Class 2
Operating Voltage	3.3V
Host Interface	USB1.1/2.0 or UART
Audio Interface	PCM and Analog interface
Flash Memory Size	8Mbit
Dimension	26.9mm(L)×13mm(W)×2.2mm(H)

FRAM is a 256-kilobit nonvolatile memory which employs an advanced ferroelectric process and provides reliable data retention for 10 years. It eliminates the complexity and reliability problems which are caused by EEPROM and other nonvolatile factors

[27]. The FRAM replaces the RAM in the MCU to play the role of storage due to its bigger space and faster speed. Here the figure 3-22 demonstrates the FRAM (*FM25W256*) used in our IMU.

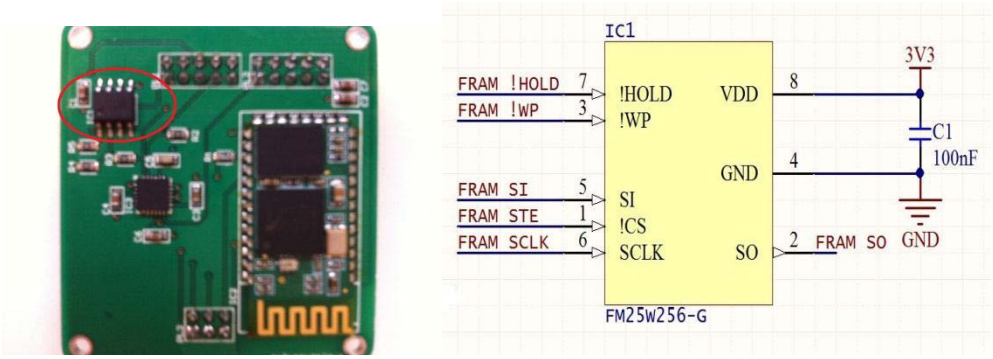


Figure 3-23 FM25W256

The last one is a sensor of the IMU, which is the central part of whole device. The sensor that is integrated on the IMU is the MPU9150. It is the world's first integrated 9-axis Motion Tracking device that includes a 3-axis MEMS gyroscope, a 3-axis MEMS accelerometer, a 3-axis MEMS magnetometer and a Digital Motion Processor hardware accelerator engine [28]. It has three 16-bit analog-to-digital converters (ADCs) for digitizing the gyroscope outputs; three 16-bit ADCs for digitizing the accelerometer outputs; three 13-bit ADCs for digitizing the magnetometer outputs. More details of MPU9150 will be introduced in the next chapter. Figure 3-24 is the physical map and schematic of MPU9150.

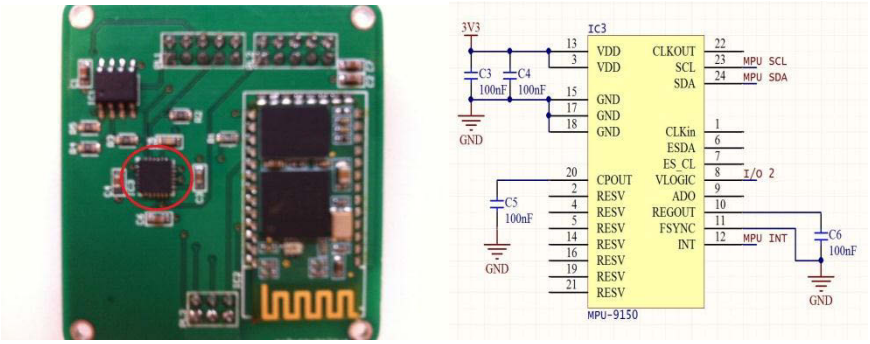


Figure 3-24 MPU9150

In summary, all major electronic components that are integrated on the IMU have already been introduced briefly. The next chapter will introduce some important components in order to understand this device more deeply. More information will be given in the next chapter about the operation in the software.

3.7 Physical layout design and some relevant knowledge about PCB

All the electronic components have been demonstrated in former parts. Their characteristics and physical maps are shown in figures and the tables. After designing the schematic, the second step is the physical layout design. The physical layout design is similar to a house design. A blueprint of a house tells the size of lumber to use, as well as the dimensions of the living room wall and the dimensions of the window. The physical layout plays the same role in the PCB design. Therefore, knowledge about the physical layout design is necessary before the manufacturing procedure. A physical layout of the IMU has been illustrated in figure 3-25.

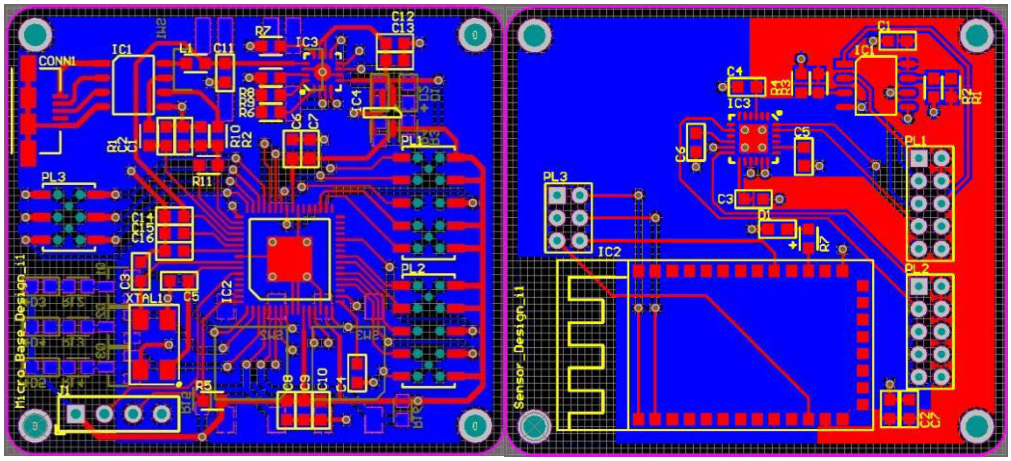


Figure 3-25 Physical Layout of IMU

The PCB materials are available in various grades which are defined by the National Electrical Manufacturers Association (NEMA). Our IMU adopts FR-4 (flame resistant) material which is commonly fabricated. The number of the layouts is also a factor to be considered. Doubled-sided boards which are used in the IMU are simpler to route because there are two layers of foil and it is possible to route by crossing traces on the different layers. There are some points that need to be noted about a double sided PCB design.

The Most Important Rule: Keep Grounds Separate

- Separate the ground plane for the analog and the digital portions of the circuitry is one of the simplest and the most effective methods of noise reduction. One or more layers on a multi-layer PCB usually have ground planes. Every ground connection in the analog circuitry must be lifted from the board and connected together. The ground and power planes are the potential noise sources. Therefore, it is important to isolate the power planes as well.

- Overlapping the digital and the analog planes is unacceptable. If any parts of the analog and the digital planes are overlapped, the overlapping portions will couple high-speed digital noise into the analog circuitry.

- It is unnecessary to make *grounds* electrically separate in the system. They have to be common at some point, in particular, a single, low-impedance point. Generally, there is only one electrical safety ground in an AC-powered system or a battery ground in a DC-powered system.

Trace Reflections

- When a PCB trace turns a corner at a 90°angle, a reflection can occur. This is primarily because of the change of width of the trace. At the apex of the turn, the trace width is increased 1.414 times. This upsets the transmission line characteristics especially the distributed capacitance and the self–inductance of the trace. It is not allowed that all PCB traces can be straight and therefore they have to turn corners.

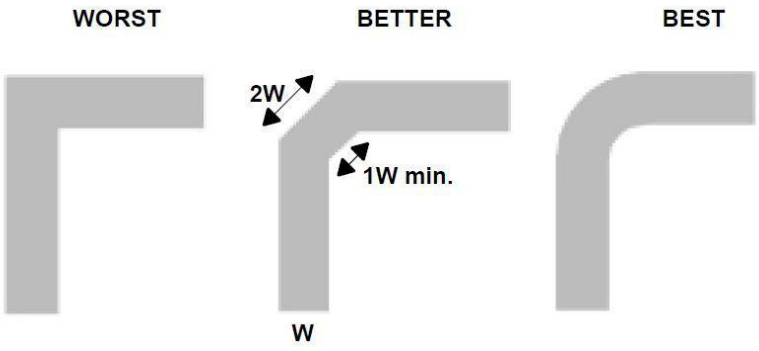


Figure 3-26 Trace Type

3.8 The procedure of PCB manufacturing (Example of Multi-layer board)

When you start manufacturing a multi-layer PCB (Printed Circuit Board), you should first ensure that all multi-layer PCBs originate from a double sided board. A double sided board is made from a rigid laminate which consists of a woven glass epoxy base material clad with copper on the both sides; it means *conductor pattern* can be located on the two sides of the board. Here is a figure of the double sided board.

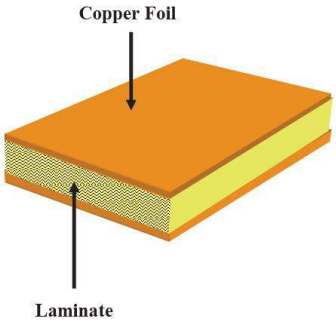


Figure 3-27 Double Sided Board

A light sensitive resist coat is applied, by using heat and pressure, to cover the copper surfaces of the board. The coat is sensitive to ultraviolet light. Figure 3-28 is detail of this procedure.

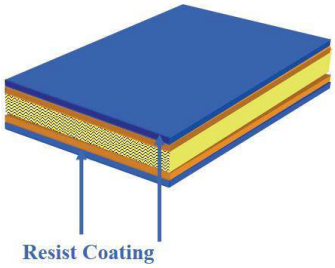


Figure 3-28 Covering a Resist Coating on the Copper Surfaces

Because the board will become an inner layer of the whole Multi-payer PCB, a special film needs to be applied to protect copper line exist. This kind of film is called the negative film which is shown in figure 3-29. The black part of this film can prevent the ultraviolet light passing through while the white part could allow the ultraviolet light to get through. The ultraviolet light can polymerize (harden) the coat that is covered on the surface of the copper. As a consequence, the film that is covered by the black part would be simple to remove by chemicals while the film covered by the white part cannot be removed by that chemical.

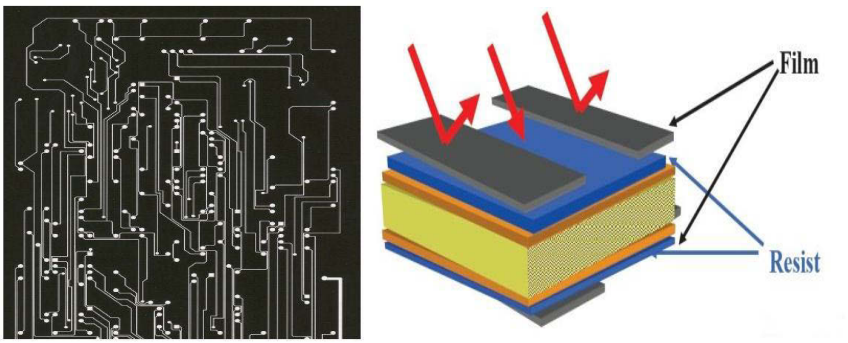


Figure 3-29 Negative Film

After the hardening process, the exposed board is processed through a chemical solution or developer that removes the resist from the areas that were not polymerized by the ultraviolet light. After that, the uncovered copper would be removed. The laminate surface is exposed in the areas where copper was etched away. Figure 3-30 shows this procedure.

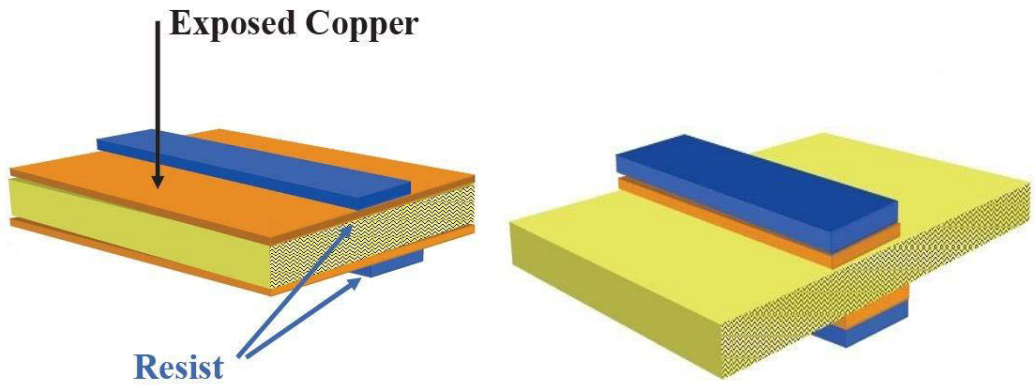


Figure 3-30 Remove Coat and Copper

Finally, the developed dry film resist is chemically removed from the panel, leaving the copper on the panel. Traces, pads, ground plane and other design features are now exposed. At this time, the inner layer is completed.

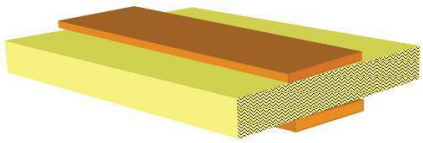


Figure 3-31 Remove Film Resist on The Copper

As mentioned in the former part, a multi-layer PCB is based on a double sided board. The double sided board is called an inner core board. After manufacturing the inner core board, it is time to manufacture the outer layer. Firstly, a PrePreg (Preimpregnated Bonding Sheet) should be added on the surface of the inner core board and a copper foil would be stuck on the surface of the PrePreg. It is the glue that holds the cores together. The material is called FR4 –a woven fiberglass cloth pre-impregnated with an epoxy resin. The resin is activated during the lamination process from the pressure and the heat. The inner layer core, the copper foil and the prepreg are bonded together under the heat and the pressure, which is achieved in a vacuum during the lamination process. The

result is a panel with several layers of copper inside as well as the copper foil on the outside.

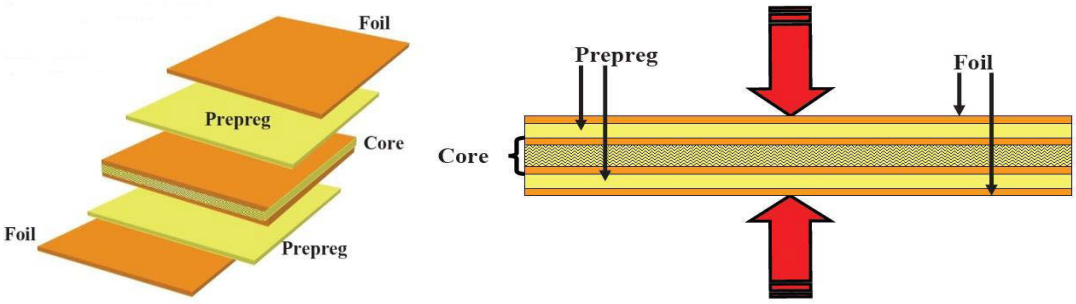


Figure 3-32 Combination of Multi Layers

After the board combination, there is an important procedure that is called the First Drill. The multi-layer board should have holes to connect every layer. Holes of various sizes are drilled through a stack of panels. The locations are determined by the board's designer to fit the specific components. The drilled hole sizes are usually 5 mm larger than the finished plated holes in order to allow the copper plating process. Then, a mechanical process removes the raised edges of the metal or burrs surrounding the holes.

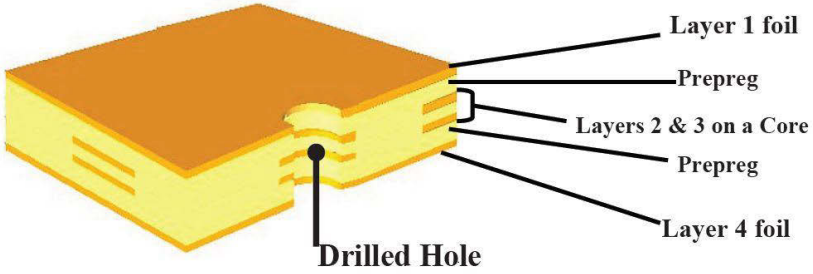


Figure 3-33 First Drill

Once the holes are ready, a thin coating of copper is chemically deposited on all of the exposed surfaces of the panel, including the holes walls by the electrolyses copper deposition. This procedure makes every layer connect each other through the holes.

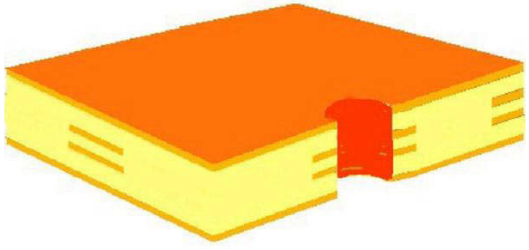


Figure 3-34 Electrolyses Copper on Holes Walls

The same resist or light sensitive film that is used on the inner layers is used for the outer layers. The film covers the entire surface including the drilled holes.

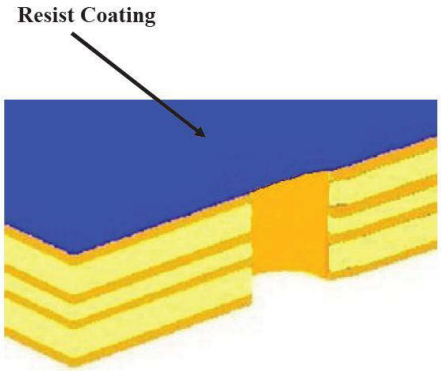


Figure 3-35 Cover Resist On Outer Layer Surface

Different from the inner layer film, the outer layer film is a positive film. The black part will prevent ultraviolet passing through while the white part could let ultraviolet pass through. The covered areas will become hard and the uncovered areas will be removed.

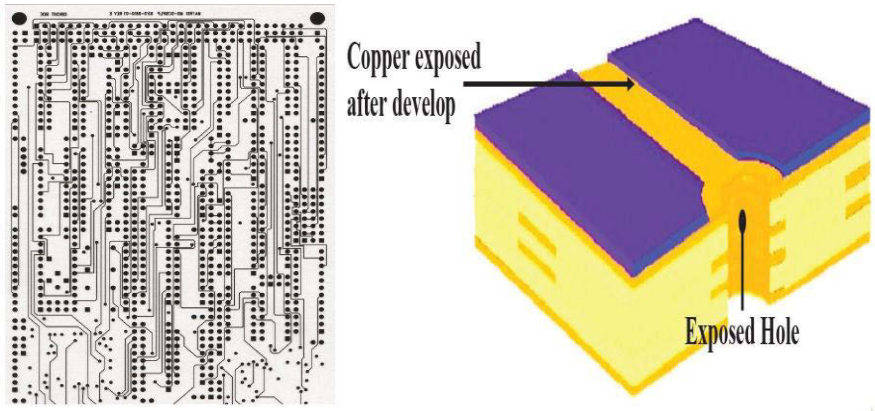


Figure 3-36. Image Develop

The next step is the electroplating processes that electrically plates copper onto the exposed metal surfaces. The copper will be plated up to a thickness of approximately 1 mm (0.001m). The copper plating step is followed by the plating of tin onto all exposed copper surfaces as an etch resist to maintain the copper traces, hole pads and walls during the outer layer etch process.

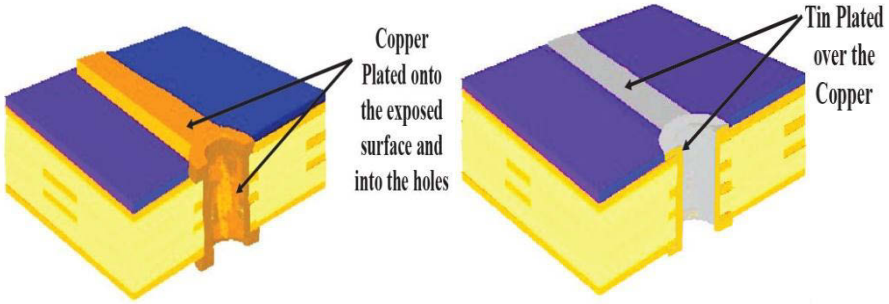


Figure 3-37 Copper Plating and Tin Plating

The developed dry film resist is now removed from the panel while the tin plating is not affected. After that, copper is now removed from all parts of the panel which are not covered by tin. Then the tin is chemically removed, leaving behind a bare copper and laminate panel. Only the pads and traces from the artwork are left behind on the panel surface.

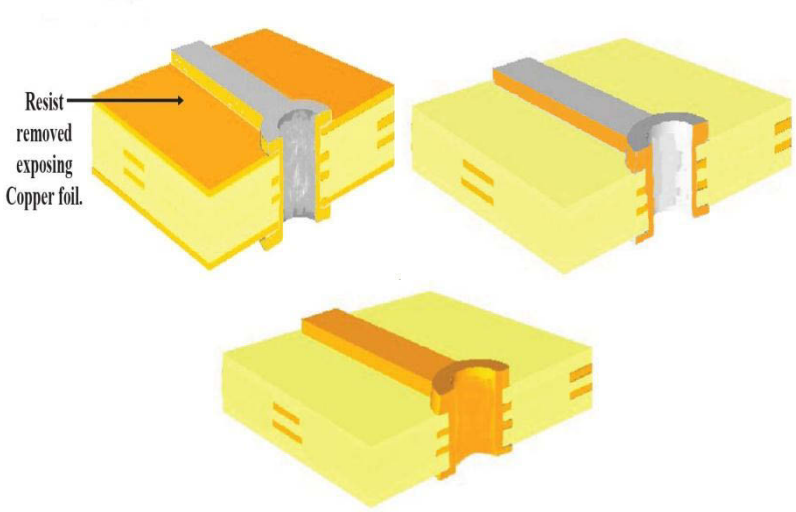


Figure 3-38 Etch Strip

After that, cleaning work is necessary before application of a solder mask. A photo-sensitive epoxy based ink can entirely cover the surface of the panel. The panels are exposed to an ultraviolet light source through a film tool and then the panel is developed by the artwork. High pressure hot air is applied on both sides of panel simultaneously to cure the solder mask. The primary purpose of the solder mask is to restrict the areas that are covered by solders. This means that only pin place could be soldered. It also protects panels from contamination, handling damage and possible electrical shorting during assembly and installation procedures.

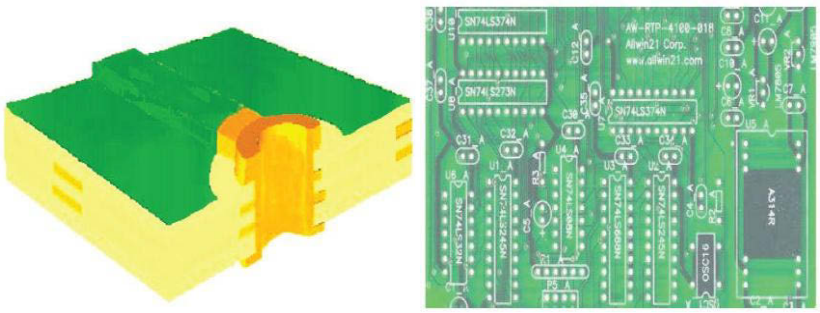


Figure 3-39 Solder Mask

3.9 Chapter Conclusion

This chapter mainly introduces the hardware structure of the IMU. The IMU consists of the power supply and management unit, the SMD connectors, the oscillator, the USB, the FRAM, the Bluetooth, the MPU9150, the switches and some resistors and capacitors. The functionality and the features are described carefully in the chapter. The power supply and management unit can supply safe power and improve the efficiency of the battery. The SMD connectors are used to connect the top board and the bottom board; the JTAG is used to debug the software. The oscillator provides reliable sampling frequency to the IMU. The USB can supply power to the IMU and build the wired communication with PCs. The FRAM, the Bluetooth and the MPU9150 are the core components which will be introduced in the next chapter.

IV. IMU CORE COMPONENTS AND PROTOCOL

4.1 Introduction

The previous chapter has briefly introduced all the major electronic components. In these electronic components, there are 4 components which play a vital role in the IMU. The MCU (Micro controlling Unit) MSP430F5528 provides controlling orders and clock signal to other parts. It is the role that is similar to a brain of a human. The sensor MPU9150 provides samples to analyse the posture of the IMU. This function corresponds to a human cerebellum. The FRAM FM25W256 is the storage unit of the IMU; all data and code are stored in this unit. The Bluetooth RF-BT0417C is a communication unit which makes the IMU possible to be a portable device. This unit has the same functionality with a mouth and ears of a human. These 4 ICs (Integrated Circuit) are the core components of the IMU. This chapter will be focused on their operational protocols. In the final part of this chapter, another communication path, a USB (Universal Serial Bus), will be introduced. Figure 4-1 shows the structure of the IMU.

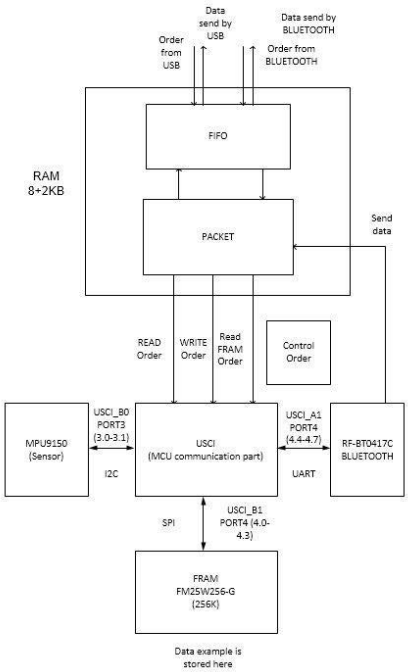


Figure 4-1 Structure of IMU

This structure can be divided into 3 parts. The first part is the USCI (Universal Serial Communication Interface) part and its connections. The second part is the FIFO and Packet part. The third part is the USB. The central part of this structure is the USCI which is the MCU communication module. It adopts three different protocols to communicate with the sensor, the Bluetooth and the FRAM. Orders that come from a PC or an MCU must go via the USCI module to reach the three parts. The FIFO and Packet part in the MCU plays the role of a buffer that prevents the transmission error due to a different transmission speed between the PC and the MCU. The USB is another communication path with outside the MCU. Therefore, this chapter will be started at the MCU and later the USCI communication protocols. After that, the buffer (FIFO and Packet) and the USB will be introduced in detail.

Generally, the most important factor to successfully understand a MCU is to understand its pin functionality. The characteristic of the MCU usually depends on these pins. Because of this, the MCU MSP430F5528 and its pins will be illustrated in figure 4-2. This section will focus on the functionality of these useful pins.

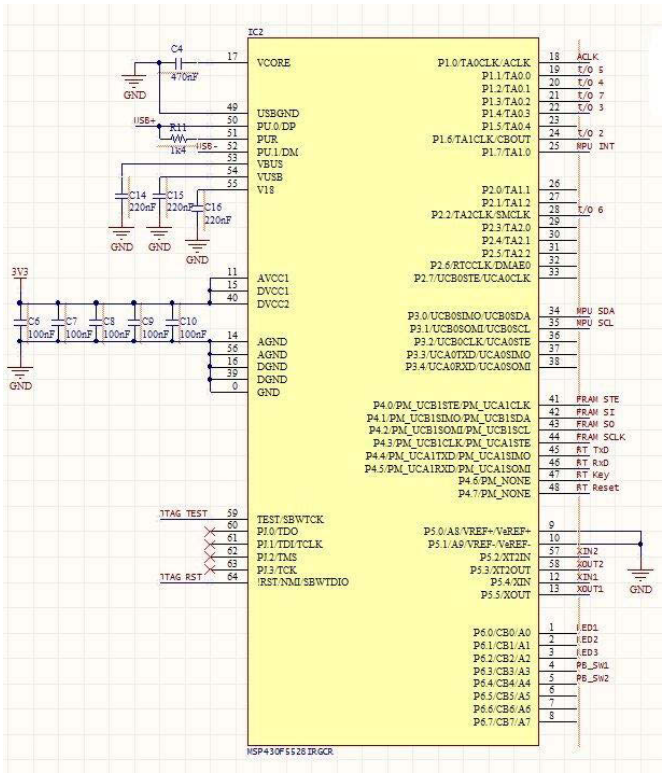


Figure 4-2 MSP430F5528

The pins that connect to the sensor MPU9150 are the pin 24, 25, 34, 35. Their functionalities are demonstrated in table 4-1 [29].

Table 4-1 PINS CONNECTION BETWEEN MPU9150 AND F5528

Pin Number of F5528	Pin Functionality	Pin Number of MPU9150	Pin Functionality
34	I2C DATA USCI_B0	24	I2C Serial Data
35	I2C CLOCK USCI_B0	23	I2C Serial Clock
24	GPIO	8	Digital I/O Supply Voltage
25	TA1 CCR0 capture	12	Interrupt Digital Output

4.2 MPU9150 and I2C

As it is shown in table 4-1, the communication protocol between the MCU and the MPU9150 is I2C. A USCI_B0 module supports I2C protocol, communicating with the MPU9150 [14]. In this communication, the USCI acts as a master while the MPU9150 is a slave. Hence the USCI controls write or read operations through a bus. The bus consists of two lines; one is a serial data (SDA) line and another is serial clock (SCL) line. At the beginning of communication, I2C line needs a “start” mark to tell the master or the slave to start the communication. The SDA line pulls voltage from high level to low level while the SCL line keeps high level. After that, the master sends a 7-bit address and a 1-bit write/read order on the SDA line. The slave which matches this address will send an acknowledge signal to the master; then the receiving of data or the transmitting of data begins. The master or the slave can operate an 8-bit data (1 byte) each time. After an 8-bit data has been transferred, a 1-bit acknowledge signal is sent. Finally, the communication also needs a stop signal to indicate the end of the transmission. The SDA line will pull up voltage from low level to high level and the SCL line keeps high level.

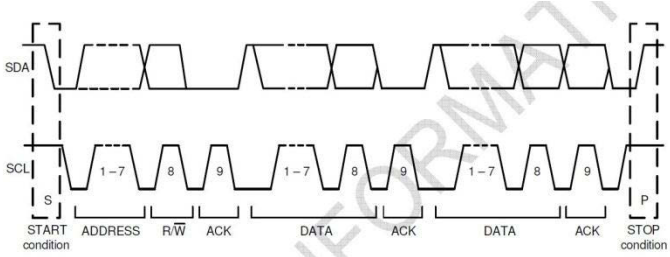


Figure 4-3 I2C Protocol

In operational codes inside the IMU, there are 6 functions in I2C.c file to achieve the I2C communication protocol. In order to explain their function, I separate them into 3 groups. The first one is used to define the acknowledge signal, named as the *ackReceived (void)*. This function block checks *ACK Interrupt Flag* to indicate whether it is time to send a stop bit. If there is no ACK flag, the system continues sending or receiving operations until an ACK flag has been detected. The second one is *I2C setup*. In this function block, some configurations have been made. It defines the slave address, the I2C work mode (Transmitting mode, master mode, synchronic working), the clock signal source and its frequency (SMCLK, 99.942KHZ), the pins for the SDA line and the SCL line which are used to connect to the MPU9150 (P3.0 and P3.1). The last group contains 4 function blocks which are used for sending and receiving data. These 4 functions define transmission mode by one byte mode or multi-byte mode which matches I2C protocol format. Figure 4-3 shows the details of the I2C protocol.

By using the I2C protocol, the MSP430F5528 builds a reliable communication with the MPU9150. The MPU9150 integrates a 3-axis accelerometer, a 3-axis gyroscope and a 3-axis magnetometer. It uses a 16-bit ADC for the accelerator output and the gyroscope output and a 13-bit ADC for the magnetometer output. A full-scale range of ±250, ±500, ±1000, ±2000°/sec (dps) programmable gyroscope, a full-scale range of ±2*g*, ±4*g*, ±8*g*, ±16*g* programmable accelerometer and a full-scale range of ±1200μT magnetometer can be selected in the MPU9150 [28]. The 400 kHz I2C protocol is used to communicate with the MCU. The accelerometer and the gyroscope are adopted in the MPU9150 for the IMU application. In order to achieve the function that we need, an accurate and proper configuration is required. Based on the Microcontroller operational system and our device, the MPU9150 is integrated on the bottom of the whole device. Thus it is vitally necessary to redefine the code for the MPU9150. In the *Sensor.h* file,

all the digital addresses have been redefined to a new but simple readable name. The address of MPU9150 is redefined; 0x69 is the first address in the whole device for the MPU9150. Hence a specific register function has been listed. After defining all the addresses and their functions, another process is that it needs to convert the analog data to ASCII for reading on the PC.

After all the preparations have been done, sensor configuration is required. In the *Sensor_Setup* function, four working mode is selected. It configures the full scale range of gyroscope and the accelerometer to 250 degree per second and 8g respectively. Then it disables the FIFO buffer inside the MPU9150 and the I2C Master Mode. However, since the accuracy of the sensor needs to be guaranteed, a self-test is necessary. The most basic principle of this test is comparing the Self-Test Response between the Sensor Output with Self-Test and the Sensor Output without Self-Test. If the Self-Test Response is lower than 14%, that means the configuration is available. In the last part of the code, it defines the read and write format and how to get samples from the sensors.

4.3 FRAM and SPI

The second IC that will be introduced is the FRAM FM25W256. The pins are illustrated in table 4-2 [27].

Table 4-2 PINS CONNECTION BETWEEN FRAM AND F5528

Pin Number of F5528	Pin Functionality	Pin Number of FM25W256	Pin Functionality
41	Slave Transmit Enable	1	Chip Select
42	Slave in Master out SPI mode	5	Serial Data Input
43	Slave out Master in SPI mode	2	Serial Data Output
44	Clock Signal Output SPI master mode	6	Serial Clock

The table 4-2 illustrate the fact that the FRAM communicates with the MCU under a SPI master mode. That is, the USCI_B1 module is a master and the FRAM is a slave. The master provides a clock signal for the slave [14]. Thus the SPI is a synchronous communication (figure 4-4). The data which is required to transmit from the master should firstly enter the Transmit Buffer then move to the Transmit Shift Register. After

that, data reaches the *UCxSIMO* register and starts transmitting from its most significant bit (MSB). When the character is received, it first enters the Data Shift Register before it moves to the SPI Receive Buffer. And also, the received data should pass through the SOMI register and reach the Receive Shift Register and finally move to the Receive Buffer. At this time, the receive interrupt flag UCRXIFG is set. The TX/RX operation is completed.

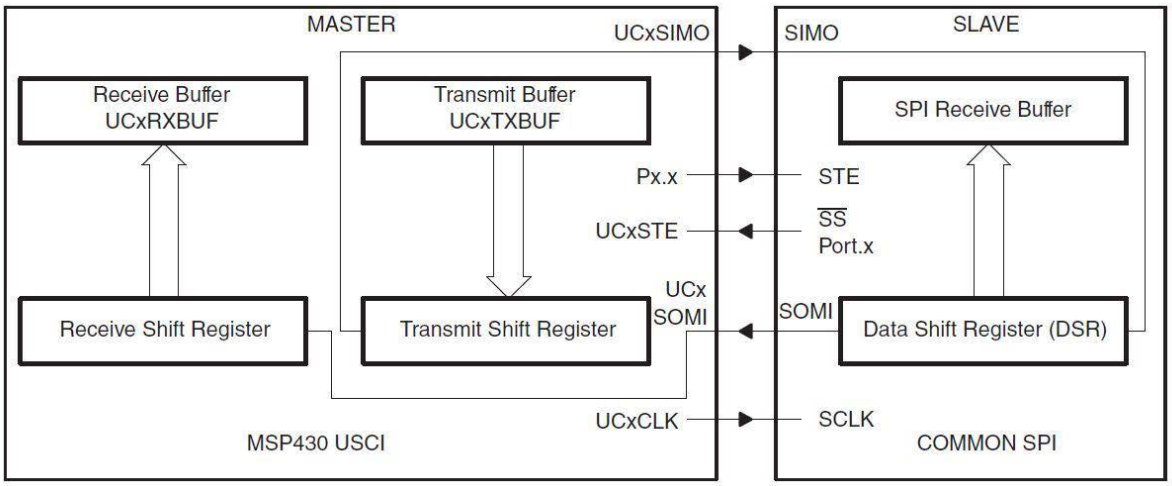


Figure 4-4 SPI master mode

In the code, the SPI part is quite simple. In the SPI.c file, there are only 2 functions. One is the *SPI_Setup* and another is the *SPI_ExchangeChar*. In the first one, it configures the SPI module working mode (MSB first, 8-bit data, Master mode, 3-pins SPI, synchronous), frequency (8MHz) and Pins (P4.1-P4.4 as input, output, state and clock). The second function is that it is used to follow the protocol, putting data in the Transmit Buffer and receiving data in the Receive Buffer. It uses the UCTXIFG and the UCRXIFG as supervisors of the whole transmission. Figure 4-4 is the diagram of SPI master mode.

The FRAM is a new powerful memory register that has faster speed and more space than normal RAM. The style of FRAM that is used in our program is FM25W256 which has 256KB space and 25MHz frequency [27]. However, unlike RAM, it also needs operational code before its address and data. The formats are as follows:

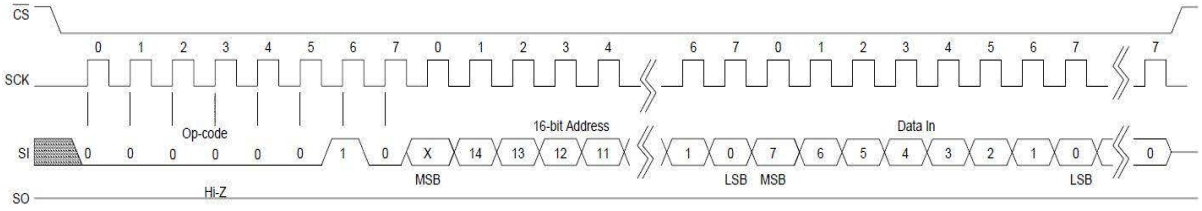


Figure 4-5 Memory Write

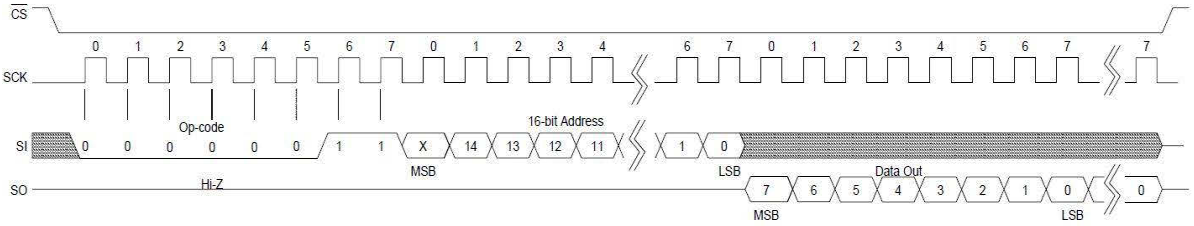


Figure 4-6 Memory Read

The code requires following the specific format of writing/reading in order to operate FRAM. At the beginning of a character, an op-code is necessary. At the same time, the PC needs to know the status of the FRAM to ensure whether it is at a status of write protection. If the FRAM is at the write protection status, the data cannot be written in the FRAM. Thus, in the code, a function block *FRAMRead_Status* notes the PC what the situation of the FRAM is. If the FRAM is not at write protection, the *FRAMWrite_Status* function can operate the FRAM to store data. A 16-bit address follows the op-code. The useful address is from 0 to 7FFFh [27]. If the address surpasses that, it rolls back to 0000h. After that, there is an 8-bit data which follows. Consequently, as long as you follow the data format, the operation of the FRAM is not difficult.

4.4 Bluetooth and UART

The last one is the Bluetooth communication with the MCU. The pins are shown in table 4-3.

Table 4-3 PINS CONNECTION BETWEEN BLUETOOTH AND F5528

Pin Number of F5528	Pin Functionality	Pin Number of FM25W256	Pin Functionality
45	Transmit Data USCI_UART mode	1	UART TX
46	Receive Data USCI_UART mode	2	UART RX

47	GPIO	26	KEY
48	GPIO	11	RESET

Bluetooth communicates with the MCU by a UART (*Universal Asynchronous Receiver/Transmitter) protocol. The* UART communication is the only asynchronous communication in these three and its data format is illustrated in figure 4-7 [14]. The default voltage is high level. When it falls down to the low level, this is the "start" bit which is followed by an 8-bit data. After that, the address bit, the parity and stop bit are required. Totally, 14-bits are used to transmit one character.

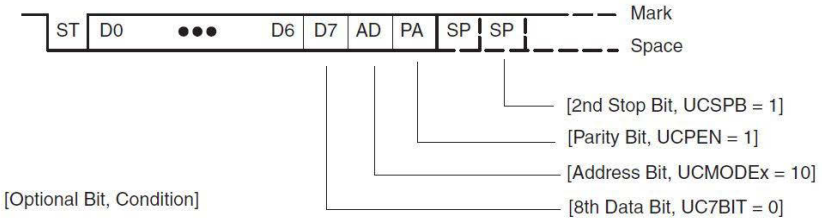


Figure 4-7 UART Protocol

In the code, the function *UART_Setup* indicates its configuration (No parity, LSB first, 8 bits data, 1 stop bit, asynchronous communication). It also configures the communication speed (115200 baud) and pins for the communication with Bluetooth. Finally, interruptions in the UART are available. That is, after receiving a character, an interrupt signal will be produced and an interrupt route will be called. The interrupt route defines all the data which are received from Bluetooth; these data will be stored in the FIFO buffer before other operations.

The Bluetooth protocol is a complex integration of many different protocols. Figure 4-8 and 4-9 are a brief diagram of Bluetooth protocol [25].

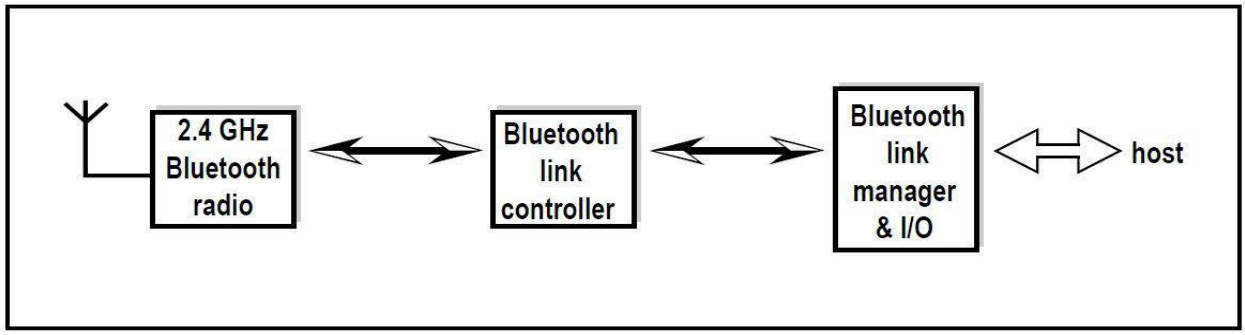


Figure 4-8 Brief Structure of Bluetooth Protocol

Generally, a Bluetooth protocol has integrated four kinds of protocol inside the Bluetooth protocol architecture： the Core protocols, the Cable replacement and telephony control protocols and the Adopted protocols.

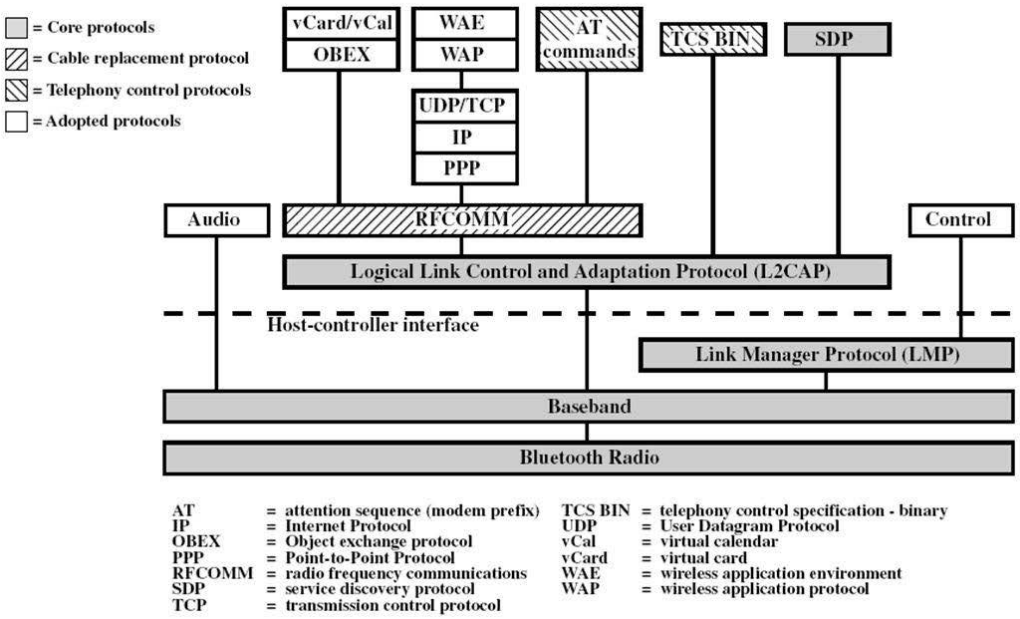


Figure 4-9 Structure of Bluetooth Protocol

The Baseband in figure 4-9 is equal to the Bluetooth link controller in figure 4-8. The functionality of the Baseband is to select the working mode of Synchronous Connection-Oriented (SCO) link or Asynchronous Connection-Less (ACL) link. The SCO link is a point-to-point link between a master and a single slave in the piconet while the ACL link is a point-to-multipoint link between the master and all the slaves which participates in the piconet [25].

The Radio works in the unlicensed ISM band at 2.4 GHz as shown in figure 4-8 [25]. A frequency hop transceiver is adopted to combat interference and fading; a shaped binary FM modulation is applied to minimize the transceiver's complexity. On the channel, information is exchanged through data packets which are transmitted on a different hop frequency. Slots which connect to PC or other devices can be reserved for synchronous packets while the Bluetooth can support an asynchronous data channel, up to three simultaneous synchronous channels, or a channel which simultaneously supports asynchronous data and synchronous signals. Each signal channel supports a 64 kb/s

synchronous channel in each direction. The asynchronous channel can support a maximal 723.2 kb/s asymmetric transmission or a 433.9 kb/s symmetric transmission.

The LMP messages are used for link setup, security and controlling. They are transferred in the payload rather than the L2CAP. The Link Manager messages have higher priority than user data. If the Link Manager needs to send a message, it shall not be delayed by the L2CAP data flow, although it can be delayed by many individual baseband packets.

Briefly, in the IMU, the MSP430 or the PC send a Control order to the LMP. The LMP receives this order and sends another order to the Baseband to select a working mode. After that, the packets are submitted to the Radio part.

4.5 FIFO and Packet

The FIFO and packet that is mentioned in the former part are necessary modules of the IMU. In particular, these two are the frontline of the communication between the PC and the IMU. The FIFO is actually a data buffer and the packet is a kind of data communication format that is used in the USB and the Bluetooth protocol. In our IMU code, there are two kinds of packets in the code. One is used for receive data from the PC; it is the order of operation and response, while another is used to transmit data from the MCU. The purpose of the packet is to display information in specific format by Hyper Terminal on the PC. The former one contains five bytes: these are command code, specific data relates to the command, checksum used for reliability. The later one is more complex and bigger and contains command code, timestamp, data, checksum, carriage return and line feed.

The Receive data packet is mainly used to read the current IMU configuration and write configuration information into the IMU as well as read data from the FRAM. The contents are as follows.

Receive Data Packets:

1. Read configuration from the IMU to the PC

- Byte 1 = ‘R’
- Byte 2 = 0xXX (Don’t care)

- Byte 3 = 0xXX (Don’t care)
- Byte 4 = 0xXX (Don’t care)
- Byte 5 = Checksum

Response packet:

- Byte 1 = ‘R’
- Byte 2 = |ASCII/Byte|AccelON|GyroON|MagON|AFS1|AFS0|GFS1|GFS0|
- Byte 3 = |FramON|Desired sample frequency in Hz (7 bits)|
- Byte 4 = 0x00 (Blank)
- Byte 5 = Checksum

2. Write configuration from the PC to the IMU

- Byte 1 = ‘W’
- Byte 2 = |ASCII/Byte|AccelON|GyroON|MagON|AFS1|AFS0|GFS1|GFS0|
- Byte 3 = |FramON|Desired sample frequency in Hz (7 bits)|
- Byte 4 = 0xXX (Don’t care)
- Byte 5 = Checksum

Response packet:

- Byte 1 = ‘W’
- Byte 2 = |ASCII/Byte|AccelON|GyroON|MagON|AFS1|AFS0|GFS1|GFS0|
- Byte 3 = |FramON|Sample frequency in Hz (7 bits)|
- Byte 4 = 0x00 (Blank)
- Byte 5 = Checksum

3. Read data from FRAM to PC (After reading the data, it would be erased in FRAM to give space to another new data)

- Byte 1 = ‘F’
- Byte 2 = 0xXX (Don’t care)
- Byte 3 = 0xXX (Don’t care)
- Byte 4 = 0xXX (Don’t care)
- Byte 5 = Checksum

Response packet:

- The response packet is actually multiple packets, where each packet output is a sensor sample that has been stored in the FRAM and the final packet indicates the end of data storage:
    - Packet(s) before last packet – The format of this packet is almost the same as the "Sensor Data Output" packet (Transmit Data Packets). The only difference between the FRAM packets and Transmit Data Packets is that the FRAM packet has byte 1 = 'F' at the start while the Transmit Data Packets has byte 1 = 'S'. For example, configuration: ***ASCII/Byte = 1, AccelON = 0, GyroON = 1 and MagON = 0 (ASCII, gyro only) – | 'F' | ',' | '####' (RelativeTimestamp, 4 bytes) |',' | '–#####,' (Gx, 7 bytes) | '–#####,' (Gy, 7 bytes) | '–#####,' (Gz, 7 bytes) | Checksum (1 byte) | CR | LF |***
    - Final "End of FRAM data" packet:
        - Byte 1 = 'F'
        - Byte 2 = 0x00
        - Byte 3 = 0x00
        - Byte 4 = 0x00
        - Byte 5 = Checksum

Transmit Data Packets:

The basic structure of Transmit Data Packets contains these six parts:

1. 'S' – Command code for Transmit Data
2. RelativeTimestamp (word for Byte mode, 4 bytes for ASCII mode) – The time (in ms) since the previous sample was output
3. SensorData – The size of this varies depending on the settings ASCII/Byte, AccelON, GyroON and MagON
4. Checksum of all the previous bytes in the packet (byte)
5. Carriage Return (byte) – Only included in ASCII mode
6. Line Feed (byte) – Only included in ASCII mode

The Packet and FIFO are a transceiver point between the IMU and the PC. Data sent or received by the PC first enters the FIFO buffer to balance the speed and make a packet

before it is received by the IMU or the PC. However, inside the IMU, all modules communicate with each other by their own protocol such as SPI, I2C or UART.

4.6 USB

4.6.1 USB Introduction

After the FIFO and Packet, data may be transmitted through a USB. USB is a very common and simple communication protocol, supporting a high speed of transmission. The USB line consists of 4 lines inside (VBUS, D+, D-, GND). In order to make it convenient in the MSP430, TI built an API (application programing interface) for users. The API cannot be modified by users. However, the sources that are contained in the API are flexible and can be edited if necessary. The IMU uses an API CDC (Communication Device Class) for working.

Before introduction of the CDC, the USB protocol needs to be explained simply. In the USB communication, data should be packeted to transmission. Unlike the UART or the SPI and the I2C which can only transfer or receive data 1 byte per time, the USB puts data into a packet and transmits a huge number of data each time. Similar to other protocols, the USB also has its own unit for transmission. The basic unit for the USB is a packet. There are four kinds of packet for USB communication. The first one is called the Token Packet which is used for operating. The Token Packet often contains input order, output order, setup order or start order. The Token Packet is to indicate how to operate a Data Packet. The second one is the Data Packet. This packet consists of probably thousands of bits of data. It is the core part of the USB communication. The third one is named the Handshake Packet. This packet only has one function to acknowledge to the host that the device has already received/transferred data successfully. The last one is the Special Packet. However, we do not use that in our program.

In the API, we cannot operate a packet inside the USB. What we can operate is a USB Event. An event in the USB is similar to a character in the UART. A basic Event consists of 3 packets which are the Token Packet, Data Packet and Handshake Packet. A

USB Event is a basic operational unit in the USB communication [14]. Thus all operations such as USB configurations are to handle the event. That is the basic USB protocol knowledge which is required in the code.

In the IMU, a function *USB_init* is used for the USB initializing. This function configures the USB data terminal D+ and D-, and also selects a LDO working voltage (3.3V and 1.8 available) [14]. It sends version information to the host for enumerating and initializing DMA (Direct Memory Access). After that, the code enables all the USB Event. Here is a figure to illustrate how USB Events work.

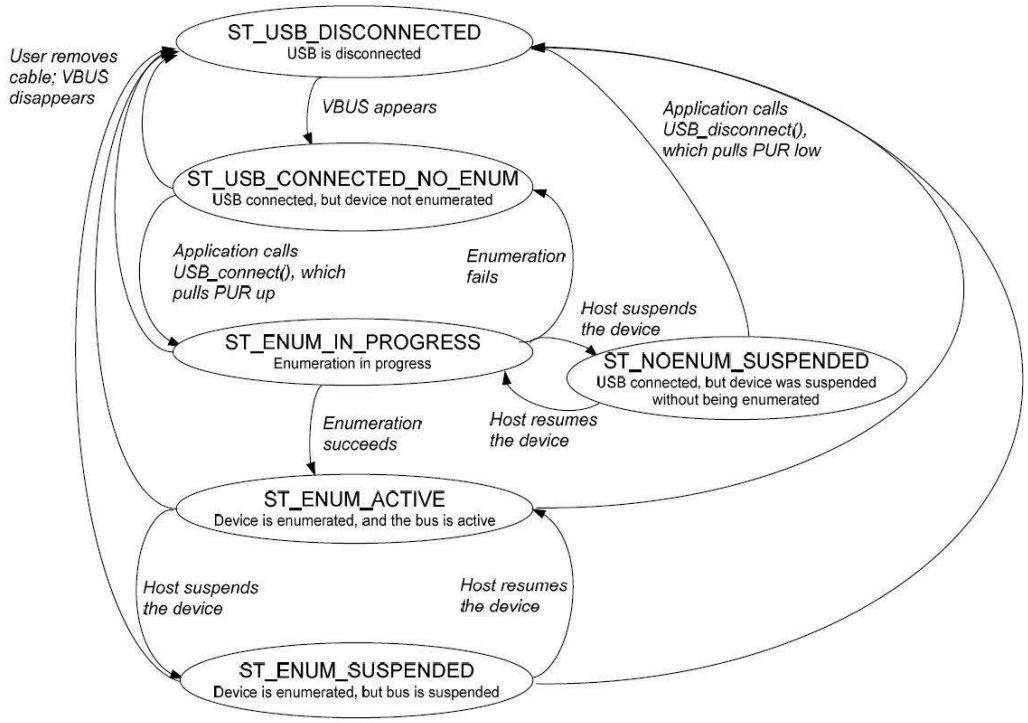


Figure 5-10 USB Events

At this time, all USB configurations have been finished. If USB has already been connected, using USB communication otherwise using Bluetooth communication.

4.6.2 USB API

The TI MSP430 serial microcontrollers apply USB through a complete development tool which is called an application programming interface (API). This API is a part of the suite of tools which TI provides to make USB simple on the MSP430. This API includes, firstly, *MSP430 USB Descriptor Tool* which is a code generation tool that generates reliable descriptors for using with this API automatically. Secondly, it includes *MSP430 USB API Stacks* which implements common USB device classes (CDC/HID/MSC). Thirdly, it has *Windows HID API* that will not be used in our project.

Finally, it also includes *MSP430 USB Field Firmware Updater* which downloads firmware to the MSP430 over the USB. The API implements four USB device classes: the Communications Device Class (CDC), the Human Interface Device (HID) class, the Mass Storage class (MSC) and the Personal Healthcare Device Class. These classes could be used simply in the corresponding devices. The structure of API is illustrated in figure 4-11.

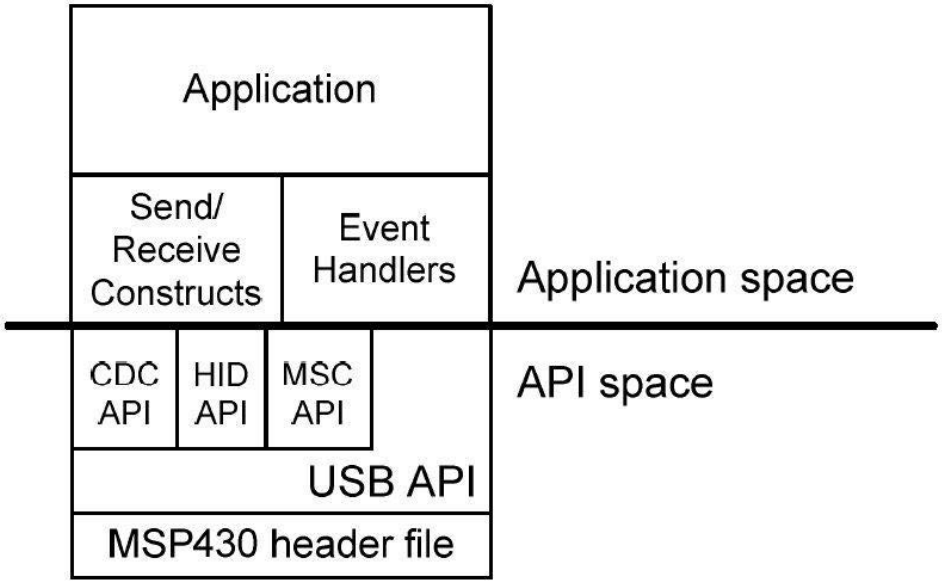


Figure 4-11 Structure of API

The API is divided into an *API space* and an *application space.* In most cases, TI recommends to only modify the application space instead of the API itself. This can preserve USB compliance and increase the chance of passing the USB certification and avoid complications. The *Send/Receive Constructions* and the *Event Handlers* are constructs which are in the *USB API* but are considered in the *Application Space*. The API files are shown in table 4-4 and these files are illustrated in figure 4-12.

Table 4-4 Files of API

	User Modify	File Name	Description
Application Space	Yes	Main.c/main.h	User application
		usbConstructs.c/h	Contains example constructs for send/receive operations.
		usbEventHandling.c/h	Event-handling placeholder functions.
API Space (Config)	Generate from Descripter Tool	*descriptors.c/h*	*Descriptors.c* contains data

			structures that define the USB descriptors the device will report to the host. A default descriptor set is provided, which can be customized with the *MSP430 USB Descriptor Tool*. *Descriptors.h* contains the configuration constants and additional descriptor information.
		UsbIsr.c	USB interrupt service routine handler, and related functionality
API Space (Core)	No	*Usbcdc.c/h*	CDC-related functionality
		UsbHid.c/h *UsbHidReportHandler.c/h* *UsbHidReq.c/h*	HID-related functionality
		UsbMscScsi.c/h *UsbMscStateMachine.c/h* *UsbMscReq.c/h*	MSC-related functionality
		usb.c/h *UsbIsr.h*	Functionality common to all USB applications.
		dma.c/h	Functions related to DMA transfers
Core Library	No	*hal_pmm.c/h* *hal_ucs.c/h* *hal_tlv.c/h*	MSP430's standard library for the F5 architecture. The USB API uses it to handle the PMM (power) and UCS (clocking) modules, and for reading the TLV structure (Tag-Length-Value) in MSP430 flash to obtain the device's

			unique die ID number.
Header Files	No	*device.h*	Controls the device derivative for which the stack is configured. In IAR/CCS, it receives its direction from the IDE's project settings.
		defMSP430USB.h	Definitions related to the MSP430 USB module
		types.h	Data type definitions
		msp430fxxxx.h	Standard header file for the MSP430 device derivative being used. This is included in the software development environment. Selection is automatically controlled by *device.h.*

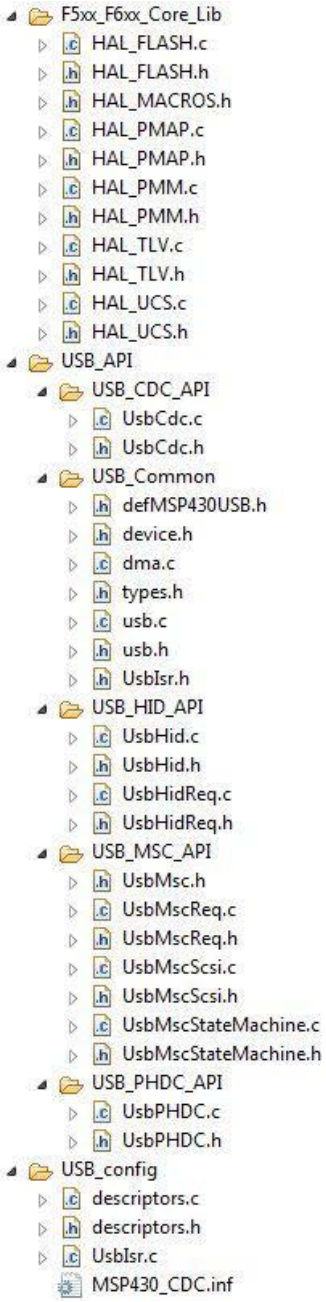


Figure 4-12 Files of API in CCS

4.6.3 API Communications Device Class

The IMU mainly uses the CDC API to communicate with the PC. The Communications Device Class (CDC) is one of the standard USB protocols. It is supported by most host operating systems such as Windows 2000, XP, Vista, and 7. In addition to a CDC driver, the host requires an interface to connect to the device. Any “terminal” application can be used to communicate with an MSP430 which is equipped by the CDC API stack.

The speed of the USB is quite fast. Full-speed USB is 12Mbps. However, this is just a theoretical maximum speed, it includes protocol overhead. Thus it is not possible for a practical application to achieve this rate for data transmission.
Furthermore, the USB is a system that integrates many components. Any component will reduce the bandwidth when transmitting. The bandwidth is affected by several factors. The first factor is the *Host Application*. The host must initiate all transmissions when sending or receiving. If the host is not initiating as often as needed, the speed will be slower. The second factor is *Bus Loading*. Because the USB adopts bulk transmitting, the consideration of the security factor will limit the speed. The third factor is software limitation. If all factors that are mentioned before are working perfectly, the USB still cannot reach the highest speed because the data receiving and sending also spend much time unavoidably. However, the USB CDC could reach 788 KB/sec (6.8Mbps) as long as it satisfies the following conditions:

- 8MHz CPU master clock (MCLK)
- Send data from Host to Device
- The USB calls function *USBCDC_rejectData(**BYTE intfNum**)*

The last factor is that when devices reject the data, this function would be called. And the application will not use any time to transmit the data. The interface *intfNum* is located at the end of the data blocks and buffers. If the function is called, the buffer for this interface is purged immediately and the data will be lost. In contrast, other data blocks must be handled; the transmission speed will be decreased to less than 788KB/sec. As a result, if the transmission speed is primarily limited by the MCU data blocks handling, it is useful running a MCLK at maximum frequency to get bandwidth close to 788KB/sec.

When the USB starts to transmit the data, the API provides a scheme for the exchanging work. This scheme is called *datapipe* which is used to send or receive data. The scheme is illustrated in figure 4-13.

```
USBCDC_sendData(addr, size, intfNum);
```

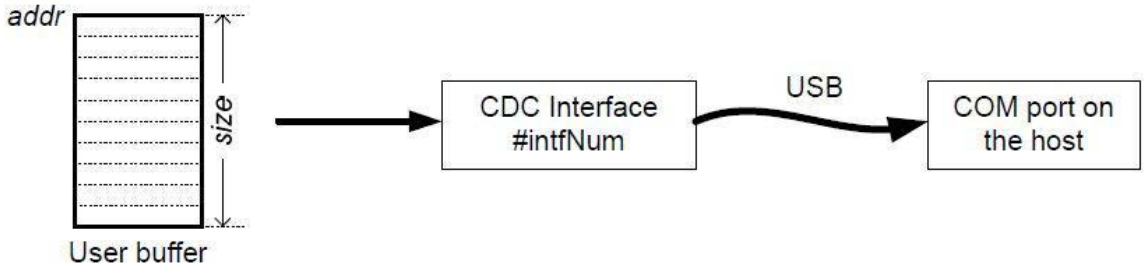


Figure 4-13 Datapipe

All transmission must start at a *User buffer* which can be located in any address in the memory unit. The data is stored in this buffer firstly as well as the number of bytes. After that, select an interface of CDC (order function), such as *USBCDC_sendData()* or *USBCDC_receiveData(),* to give the API a specific command.

The function that is used to initialize sending operation is *USBCDC_sendData()*. The API copies all data in the *User buffer* to the *endpoint buffer* in specific packetized blocks according to protocol. After the packets of data are formed, the host starts to read the packet from the *endpoint buffer*.

The receiving operation is similar to the sending operation. The API first receives an order, *USBCDC_receiveData(),* to make preparation for receiving. Now the API copies all data in the *endpoint buffer* to the *User buffer* until it is full.

The concept of "buffer" here implies a RAM source as well as Flash or peripherals. The size of buffers is unlimited. That means it can be any size, one byte or kilobytes. However, the *endpoint buffer* is different. The size of the *endpoint buffer* is up to 64 bytes. It should be regarded as a register inside the USB module which stores the USB data which is waiting for sending or receiving. The *endpoint buffer* is not a part of the API while the API handles all data packets through the *endpoint buffer* automatically. The *endpoint buffer* cannot be written during sending or receiving otherwise the operation will be interrupted.

The sending and receiving work is executed in the back ground. The User can only obtain the information that the transmission is started or completed. The advantage of this is that the efficiency is quite high because there is no need for the API to stay in idle. Other useful activities could be executed during this period. All transmission in the

CDC API is operated by the CDC data handling functions and the CDC event handling functions:

Table 4-5 CDC Data Handling Functions

Function	Description
BYTE USBCDC_sendData()	Begins a send operation to the USB host
BYTE USBCDC_receiveData()	Begins a receive operation from the USB host
BYTE USBCDC_bytesInUSBBuffer()	Returns the number of bytes residing in the USB endpoint buffer awaiting a receive operation to move them to a user buffer.
BYTE USBCDC_abortSend()	Aborts an active send operation
BYTE USBCDC_abortReceive()	Aborts an active receive operation
BYTE USBCDC_rejectData()	Rejects payload data residing in the USB buffer, for which a receive operation has not yet been initiated
BYTE USBCDC_intfStatus()	Returns status information specific to a particular CDC interface
void USBCDC_setCTS()	Toggle state variable for CTS in USB Stack (Bridge only)

BYTE USBCDC_sendData(BYTE * data, WORD size, BYTE intfNum)

This function is used to initiate sending data through the CDC interface *intfNum* and to define data size (*size)* and address (*data).* In most cases, if a sending operation is started successfully, the function will return *kUSBCDC_sendStarted.* If the USB is not connected, *kUSBCDC_busNotAvailable* will be returned to the system and no operations will be continued. If the data size is 0, the function will return *kUSBCDC_generalError.* If there are other operations underway, the function returns with *kUSBCDC_intfBusyError.*

Table 4-6 Parameters for USBCDC_sendData()

BYTE* data	passed in	An array of data to be sent
WORD size	passed in	Number of bytes to be sent, starting from address *data*.
BYTE intfNum	passed in	Which data interface the data should be transmitted over
Returns		kUSBCDC_sendStarted*:* a send operation was successfully started kUSBCDC_intfBusyError*:* a previous send operation is underway kUSBCDC_busNotAvailable*:* the bus is either suspended or disconnected

	kUSBCDC_generalError*:* size was zero, or other error

BYTE USBCDC_receiveData(BYTE * data, WORD size, BYTE intfNum)

Similar to the last function, *USBCDC_receiveData* defines an interface *intfNum* and data size *"size"* and its address *"data"*. If the receiving operation is underway, the function will return *kUSBCDC_receiveStarted.* If the receiving operation is completed, the function will return *kUSBCDC_receiveCompleted.* If the bus is not connected while the function is called, the function returns *kUSBCDC_busNotAvailable.* If *size* is 0, the function returns *kUSBCDC_generalError.* If there is another function underway, the function returns *kUSBCDC_intfBusyError.*

Table 4-7 Parameters for USBCDC_receiveData()

BYTE* data	passed in	An array to contain the data received.
WORD size	passed in	Number of bytes to be received
BYTE intfNum	passed in	Which data interface to receive from
Returns		kUSBCDC_receiveStarted*:* A receive operation has been succesfully started. kUSBCDC_receiveCompleted*:* The receive operation is already completed. kUSBCDC_intfBusyError*:* a previous receive operation is underway kUSBCDC_busNotAvailable*:* the bus is either suspended or disconnected kUSBCDC_generalError*:* size was zero, or other error

BYTE USBCDC_bytesInUSBBuffer(BYTE intfNum)

This function can return the number of bytes that are waiting in the USB endpoint buffer for interface *intfNum*. If the return value is non-zero, the application can either open a receive operation in which the data can be moved out of the *endpoint buffer*, or the data should be rejected.

Table 4-8 Parameters for USBCDC_bytesInUSBBuffer()

BYTE intfNum	passed in	The data interface whose buffer is to be checked
Returns		The number of bytes waiting in this buffer

BYTE USBCDC_abortSend(WORD* size, BYTE intfNum)

This function can stop an active sending operation on data interface *intfNum* and return the number of bytes (*size*) which have been sent.

If a surprise removal of the bus or a USB suspend event or any send operation that extends longer than desired has occurred, the API will call this function.

Table 4-9 Parameters for USBCDC_abortSend()

WORD* size	passed out	Number of bytes that were sent prior to the abort action.
BYTE intfNum	passed in	The data interface for which the send should be aborted
Returns		kUSB_succeed

BYTE USBCDC_abortReceive(WORD* size, BYTE intfNum)

This function can stop an active receiving operation on the CDC interface *intfNum* and return the number of bytes which have already been received and transferred to the data location.

This function is called by the API if it no longer wants to receive data from the USB host. It should be noticed that if a continuous stream of data is being received from the host, stopping the operation is similar to pressing a "pause" button. The receiving will be suspended until another receive operation is opened.

Table 4-10 Parameters for USBCDC_abortReceive()

WORD* size	passed out	Number of bytes that were received and are waiting at the assigned address.
BYTE intfNum	passed in	The data interface for which the send should be aborted
Returns		kUSB_succeed

BYTE USBCDC_rejectData(BYTE intfNum)

This function can reject data that has been received from the host for interface *inftNum.* It is located in the USB *endpoint buffer* and blocks further data until a new receiving operation is opened. When this function is called, the buffer for this interface is purged as well as the data.

Table 4-11 Parameters for USBCDC_rejectData()

Returns	kUSB_succeed

BYTE USBCDC_intfStatus(BYTE intfNum, WORD* bytesSent, WORD*bytesReceived)

This function is used to describe the status of the CDC interface *intfNum*. If a sending operation is active for the interface, the function returns the number of bytes that have been transmitted to the host. At the same time, if a receiving operation is active for the interface, the function also returns the number of bytes that have been received from the host and waits at the assigned address.

Table 4-12 Parameters for USBCDC_intfStatus()

BYTE intfNum	passed in	Interface number for which status is being retrieved.
WORD* bytesSent	passed out	If a send operation is underway, the number of bytes that have been transferred to the host is returned in this location. If no send operation is underway, this returns zero.
WORD*bytesReceived	passed out	If a receive operation is underway, the number of bytes that have been transferred to the assigned memory location is returned in this location. If no receive operation is underway, this returns zero.
Returns		kUSBCDC_waitingForSend: Indicates that a send operation is open on this interface kUSBCDC_waitingForReceive: Indicates that a receive operation is open on this interface kUSBCDC_dataWaiting: Indicates that data has been received from the host for this interface, waiting in the USB receive buffers, lacking an open receive operation to accept it. kUSBCDC_busNotAvailable: Indicates that the bus is either suspended or disconnected. Any operations that had previously been underway are now aborted.

USBCDC_setCTS(BYTE state)

This API is available in a Bridge Application.

Table 4-13 Parameters for USBCDC_setCTS()

BYTE state	passed in	State of CTS – an external input.
Returns		void

Table 4-14 CDC Event Handling Function

Function	Description
BYTE USBCDC_handleDataReceived()	Data has been received for CDC interface with no receiving operation underway
BYTE USBCDC_handleSendCompleted()	Sending operation on CDC interface has just been completed
BYTE USBCDC_handleReceiveCompleted()	Receiving operation on CDC interface has just been completed
BYTE USBCDC_handleSetLineCoding()	SetLineCoding request has been received from the host and new values for baud rate are available
BYTE USBCDC_handleSetControlLineState()	SetControlLineState request has been received from the host and new values for RTS are available

BYTE USBCDC_handleDataReceived(BYTE intfNum)

This event is used to express that data has been received from the CDC interface *intfNum* with no other receiving operations underway. A receive operation can be started indirectly out of this event because *USBCDC_receiveData()* cannot be called from the event handlers. However, the handler must set a flag in *main()* to begin the receiving operation. After this function has been finished, *USBCDC_intfStatus()* for this CDC interface will return *kUSBDataWaiting* to indicate that the receiving work has been done and USB enters idle. However, if a receiving operation is always executed that prior to data arriving from the host, this event will never occur.

BYTE USBCDC_handleReceiveCompleted(BYTE intfNum)

This event indicates receiving operation on the CDC interface *intfNum* has already been completed. Data is available in the user buffer when calling the function *USBCDC_receiveData()*. However, if the event occurs, it means that the buffer is full.

BYTE USBCDC_handleSetLineCoding (BYTE intfNum, ULONG lBaudrate)

This event indicates that a *SetLineCoding* request has been received from the host and a new baud rate are available.

BYTE USBCDC_handleSetControlLineState(BYTE intfNum, BYTE lineState)

This event indicates that a *SetControlLineState* request has been received from the host. New values for RTS are available now. The application could use the new RTS value in the UART.

4.7 Chapter Conclusion

This chapter introduces 5 communication protocols and 4 core electronic components. The USCI_B0 module supports I2C protocol communicating with MPU9150. The FRAM communicates with the MCU under SPI master mode. The Bluetooth communicates with the MCU through UART protocol. The three core components use three different communication modules and protocols inside the IMU. Two protocols are adopted by the IMU to communicate with an outside device. The USB Communications Device Class (CDC) plays a wired communication role while Bluetooth is wireless communication. FIFO and Packet is a customized communication protocol for Bluetooth and USB in order to format the output data.

V. CALIBRATION METHOD FOR ACCELEROMETER

5.1 Introduction

The former chapters have demonstrated the hardware and the software inside the IMU. This chapter will focus on the calibration of the IMU. The Inertial measurement unit (IMU) has been widely used in inertial navigation of moving machines and posture detection of portable devices and observation of human movement [7], [15]. It is commonly composed of an orthogonally tri-axial accelerometer and a gyroscope which are applied to measure acceleration and angular velocity [7]. For further application, the IMU sensors show a excellent performance in improving flight stabilization or autonomous hovering of helicopters or quad rotors [8]. The IMU has also been integrated on cloth or shoes to track human motion [8]. Due to its simple and convenient operation and high reliability, the IMU is used to create a low-cost, high-accuracy, high sensitivity measurement system [2], [3]. Because of its outstanding advantages, the IMU has shown rapid growth in mobile device markets since 2005 [9], [10], [11]. Because of its practicability and popularity, we developed a new device that integrates the IMU system for capturing the motion of patients.

The IMU is frequently used in applications of posture and motion research, the output of an accelerometer, however, is unfortunately always inaccurate [13]. The accelerometer accuracy is affected by 3 main reasons which are tolerances during manufacturing procedures [12], temperature and external environment elements [1]. At the same time, coupled axes have a great impact on the accuracy of the whole system. Therefore, a calibration method can affect the performance of the IMU directly. Generally, most calibration methods require technical laboratory equipment such as a vibration generator or optical tracking systems or a robotic actuation to assist the calibration process [13]. The most commonly used calibration method is to align each axis of the accelerometer to a known reference acceleration, for example gravity acceleration, and to calculate the best fitting parameters according to the reference acceleration [13]. Making a comparison of the selected methods thus has a significant meaning in our research. Consequently, two low equipment dependence and high

accuracy calibration methods which are an auto calibration and a classical calibration method have been selected for comparison.

In this chapter, the first section contains details of the two calibration methods and their principles of operation [2]. The IMU features and relevant experiments will be introduced in the second section. The third section will be focused on analysis of the comparison results. The fourth part describes how to rebuild the code. The fifth part explains how to improve the accuracy. The last part gives the conclusion of the calibration.

5.2 Rough Calibration Method

5.2.1 Auto Calibration Method

Before starting the calibration procedure, an auto calibration method which is used by Frosio, Stuani and Borghese is selected as a reference method. Frosio states that the most reliable auto-calibration method is based on the basic fact that the acceleration module in which acceleration is measured by a 3-axis accelerometer should equal to the local gravity acceleration "1g" in the static condition [2], [4], [13]. He establishes a model to find the best fitting parameters and compare the angle between the X axis and XY plane, Y axis and the XY plane which is illustrated in Figure 1 to assess the performance of their calibration. The basic principle of their auto-calibration is:

$$\mathrm{g} = \sqrt{\mathrm{a_x^2} + \mathrm{a_y^2} + \mathrm{a_z^2}} \tag{5-1}$$

And their model is:

$$\begin{cases} \mathrm{a_x} = \mathrm{S_{xx}} \cdot (\mathrm{V_x} - \mathrm{O_x}) \\ \mathrm{a_y} = \mathrm{S_{yy}} \cdot (\mathrm{V_y} - \mathrm{O_y}) \\ \mathrm{a_z} = \mathrm{S_{zz}} \cdot (\mathrm{V_z} - \mathrm{O_z}) \end{cases} \tag{5-2}$$

In order to successfully determine the 6 fitting parameters,$S_{xx}, S_{yy}, S_{zz}, O_x, O_y, O_z$, the experiment records the 3-axis readings of the IMU under N different random postures [2]. These readings are regarded as known parameters V_x, V_y, V_Z in equation (5-2). Frosio uses the Nonlinear Least Squares Method to determine the other six unknown parameters. The error e_i and the sum of squares of error *E* are defined as follows:

$$\mathrm{e_i} = \sum_{\mathrm{j=x,y,z}} \left\{ [\mathrm{S_{jj}} \cdot (\mathrm{V_{j,i}} - \mathrm{O_j})]^2 \right\} - \mathrm{g}^2 \tag{5-3}$$

$$E = \sum_{i=1}^{N} e_i^2 \quad , \tag{5-4}$$

where S_{jj} is the sensitivity of each direction and O_j is the offset and $V_{j,i}$ is the ADC values that is obtained from the IMU. Frosio points out that the Newton Method could be performed in the minimising procedure [2]:

$$X_{n+1} = X_n - \alpha \cdot H^{-1}(X_n) \cdot J(X_n), \tag{5-5}$$

where X_n is the vector of iteration for n^{th} that contains $S_{xx}, S_{yy}, S_{zz}, O_x, O_y, O_z$ as parameters. $H(X_n)$ and $J(X_n)$ are the Hessian matrix and Jacobian vector respectively and α is a damping coefficient that is less than 1 [2].

$$J(X_n) = \left[\frac{\partial E}{\partial X_1} \quad \frac{\partial E}{\partial X_2} \quad \cdots \quad \frac{\partial E}{\partial X_n}\right] \tag{5-6}$$

$$H(X_n) = \begin{bmatrix} \frac{\partial^2 E}{\partial X_1^2} & \frac{\partial^2 E}{\partial X_1 \partial X_2} & \cdots & \frac{\partial^2 E}{\partial X_1 \partial X_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 E}{\partial X_n \partial X_1} & \frac{\partial^2 E}{\partial X_n \partial X_2} & \cdots & \frac{\partial^2 E}{\partial X_n^2} \end{bmatrix}. \tag{5-7}$$

This procedure will terminate until the condition satisfies:

$$\max\left\{\left|\frac{x_k^n - x_k^{n-1}}{(x_k^n + x_k^{n-1})/2}\right|\right\} < \varepsilon, \tag{5-8}$$

where x_k^n is k^{th} element in x_n, such as S_{xx} is x_1^n, and ε equals to 1.5×10^{-6} [2]. After the completion of the minimization, Frosio returns these six parameters to the equation (5-2) and calculates accelerations, a_x, a_y, a_z, through experimental ADC values, V_x, V_y, V_z. After the process of calculation of fitting parameters, Frosio detects orientation using the following equation (5-9)

$$\begin{cases} \varphi = \arctan\dfrac{a_x}{\sqrt{a_y^2 + a_z^2}} \\ \rho = \arctan\dfrac{a_y}{\sqrt{a_x^2 + a_z^2}} \end{cases}. \tag{5-9}$$

Figure 5-1 reports the meaning of φ and ρ [2].

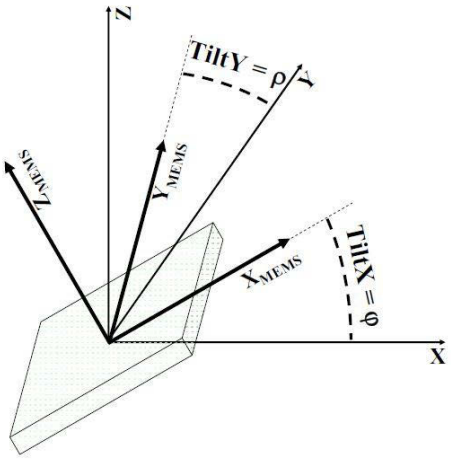


Figure 5-1 ϕ is the angle of MEMS accelerometer x direction with absolute XY plane. ρ is the angle of MEMS accelerometer y direction with absolute XY plane

In the final part of the paper, Frosio analyzes the results of the experiment. He believes that this approach is feasible because φ and ρ never exceed 2° [2].

Because Frosio's model equation (5-2) is a representative auto calibration method, this model is selected as an auto calibration method reference in the comparison research. A new but more standard and simpler linear auto-calibration model can be established, based on Frosio's model:

$$\begin{cases} a_x = S_x \cdot V_x + O_x \\ a_y = S_y \cdot V_y + O_y. \\ a_z = S_z \cdot V_z + O_z \end{cases} \tag{5-10}$$

Define the error and global error, under unit "g", based on equation (5-3) and (5-4):

$$e_i = \sum_{j=x,y,z}\{(S_{jj} \cdot V_{j,i} - O_j)^2\} - 1^2 \tag{5-11}$$

$$E = \sum_{i=1}^{N} e_i^2 \tag{5-12}$$

Convert equation (5-11) and (5-12) to a more intuitive format:

$$e_i = \left(\begin{bmatrix} V_{xi} & 0 & 0 \\ 0 & V_{yi} & 0 \\ 0 & 0 & V_{zi} \end{bmatrix}\begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} + \begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix}\right)^T \cdot \left(\begin{bmatrix} V_{xi} & 0 & 0 \\ 0 & V_{yi} & 0 \\ 0 & 0 & V_{zi} \end{bmatrix}\begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} + \begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix}\right) - 1^2 \tag{5-13}$$

$$E = \sum_{i=1}^{N} \left(\left(\begin{bmatrix} V_{xi} & 0 & 0 \\ 0 & V_{yi} & 0 \\ 0 & 0 & V_{zi} \end{bmatrix} \begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} + \begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix} \right)^{T} \cdot \left(\begin{bmatrix} V_{xi} & 0 & 0 \\ 0 & V_{yi} & 0 \\ 0 & 0 & V_{zi} \end{bmatrix} \begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} + \begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix} \right) - 1^2 \right)^2 .$$

(5-14)

The best way to minimize is to execute the Nonlinear Least Square Method, because this global error function (5-14) is a non-linear system. The basic principle of the Nonlinear Least Square Method is to minimize the unconstrained problem [6]. The mathematical format is:

$$\min f(x) = \sum_{i=1}^{n} f_i(x)^2. \tag{5-15}$$

In the model, $f(x)$ is E and $f_i(x)$ is e_i which are illustrated in equation (5-11) and (5-12). The order *lsqnonlin* in Matlab solves the nonlinear least square problem and the nonlinear data fitting problem. Using the function order and set initial parameters to operate optimization.

5.2.2 Classical Calibration Method

Classical calibration method is every direction of accelerometer axis aligns to gravity direction alternately, such as aligning X negative direction to gravity direction and record the ADC value as "-1g" and aligning X positive direction to gravity direction and record the ADC value as "1g" later. The models are as follows:

$$\begin{cases} V_{xp} S_x + O_x = 1 \\ V_{xn} S_x + O_x = -1 \end{cases} \tag{5-16}$$

$$\begin{cases} V_{yp} S_y + O_y = 1 \\ V_{yn} S_y + O_y = -1 \end{cases} \tag{5-17}$$

$$\begin{cases} V_{zp} S_z + O_z = 1 \\ V_{zn} S_z + O_z = -1 \end{cases}. \tag{5-18}$$

Every function group represents different axis, *X, Y, Z*. The matrix formats of these functions are:

$$\begin{bmatrix} V_{xn} & 1 \\ V_{xp} & 1 \end{bmatrix} \begin{bmatrix} S_x \\ O_x \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \tag{5-19}$$

$$\begin{bmatrix} V_{yn} & 1 \\ V_{yp} & 1 \end{bmatrix} \begin{bmatrix} S_y \\ O_y \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \tag{5-20}$$

$$\begin{bmatrix} V_{zn} & 1 \\ V_{zp} & 1 \end{bmatrix} \begin{bmatrix} S_z \\ O_z \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}. \tag{5-21}$$

In this case, all ADC values which are obtained from every direction should be converted to the mean value. That is to say, for example, all ADC values recorded in X positive direction will be calculated as a mean value, recorded as V_{xp}.Generally, the matrices $\begin{bmatrix} V_{xn} & 1 \\ V_{xp} & 1 \end{bmatrix}$, $\begin{bmatrix} V_{yn} & 1 \\ V_{yp} & 1 \end{bmatrix}$ and $\begin{bmatrix} V_{zn} & 1 \\ V_{zp} & 1 \end{bmatrix}$ are square and non-singular matrices and the equations (5-19), (5-20), (5-21) are the linear systems. Therefore, to obtain S and O through a Matlab order *pinv* which is used to calculate Moore-Penrose pseudoinverse of the matrix is the simplest method.

5.3 Device and Experiment

The IMU device that used for our research is shown in figure 5-2.

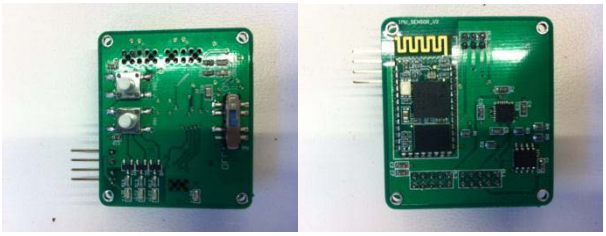


Figure 5-2 Top side (left) and bottom side (right) of IMU device

Lines through three SMD (Surface Mounted Devices) connectors between two boards build a reliable communication from the MCU (Micro Control Unit) to periphery devices. On the top board that integrates MSP430F5528, a powerful communication module named the USCI (Universal Serial Communication Interface) module is the main factor which is responsible for communicating with the bottom board. However, there are also many other small modules which contains different communication protocols, integrated in the USCI module [14]. In our code, three different protocols are used for the three periphery devices that have already been introduced in Chapter 3.

In the experiment, a table lamp is a support of the IMU. Figure 5-3 shows the details of the equipment.

Figure 5-3 Experiment for Auto Calibration Method

The support is quite similar with a manipulator with six degrees of freedom which has 4 joints. Number 1 that is showed in Figure 5-3 can rotate around absolute Z axis. Number 2 and 3 are used to adjust the height of support. Number 4 can rotate around the joint. The IMU that is fixed on the support can reach to any position and posture and remain static. Number 5 is a plummet which is used to indicate the gravity direction. In the auto-calibration method, the device should be set in N different random postures and be used to record the accelerometer data which is received from the sensor. That is quite simple to achieve through this support.

In the classical method, the experiment requires 6 orthogonal positions. That is, every different axis direction of the accelerometer aligns to the gravity direction in each measurement. Due to this restricted requirement, the classical calibration method needs to be applied using high accuracy equipment. Unfortunately, such equipment is not available. Instead, a paper roll which is made by a whole A4 printing paper can be the IMU support. Figure 5-4 shows the details of the classical calibration experiment. The paper roll can hold the edge of the IMU. That guarantees the axis which is calibrated is vertical to the ground. After that, a plummet with a line can ensure the paper roll is parallel to the gravity direction.

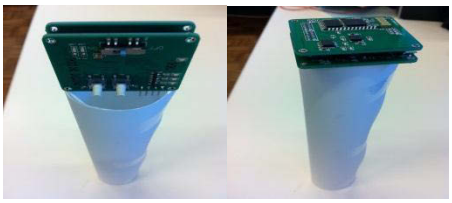


Figure 5-4 Experiment for classical calibration method

5.4 Rough Calibration Result

The results of two methods are shown on following table:

Table 5-1 RESULTS OF OFFSET AND SENSITIVITY

Parameter	Auto Method	Classical Method
S_x	0.6217E-04	0.6075E-04
S_y	0.6170E-04	0.6058E-04
S_z	0.6077E-04	0.6162E-04
O_x	0.0421	0.0374
O_y	-0.0148	-0.0144
O_z	-0.0748	-0.0729

The results shown above are the offset and the sensitivity which are calculated through the Method A (Auto-calibration) and the Method B (Classical Method). The results, however, are totally different. In order to verify the performance of the results, we re-enter these 6 parameters to the IMU. Firstly, these results are operated to re-build the code. Secondly, the new experiment which executes 84 random postures obtains some new data of accelerations. Finally, the calculation of the root mean square value (RMS) of error by the equation (5-22) and the comparison of these two methods by the RMS is performed.

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(e_i)^2} \tag{5-22}$$

Here are the results:

Table 5-2 ERROR RMS OF TWO METHODS

	Auto Method	Classical Method
RMS	$0.074g^2$	$0.069g^2$

It is clear that the Method B has higher accuracy than the Method A. The most important reason to cause this difference is coupling. Every axis of the accelerometer affects each

other in any time and position. In the Method A, the experiment is based on the N random positions and their data. Every position has a different coupling effect. In the Method B, however, 6 orthogonal positions have minimal coupling effect to each other. That is the main reason that the Method B performs better than the Method A. The second reason is that although the two models use the same format of the definition in the acceleration, the Method A's algorithm is much more complex than that of Method B. A more complex algorithm requires more procedures for the curve fitting. Hence, the error will be increased during the calculation.

The advantage of the Method A is more general. It is unnecessary to guarantee the orthogonality of the 3 axes and each direction aligns to the gravity direction or not. As long as the device can be held statically in some positions, calibration could be done successfully. The disadvantage is that the algorithm is more complex and the accuracy is lower. On the other hand, the Method B is simpler and has a lower equipment demand and also has higher accuracy than the Method A. However, the disadvantage is that if the experiment table and the ground are not flat enough, the Method B will cause extra error.

5.5 Build Calibration Mode

After experiment and calculation of these parameters, it is time to build the calibration mode. In the calibration mode, the system will convert the raw data to a readable and calibrated data. When users press button 1, the system will enter the calibration mode. If users press button 1 again, the system will return to the original status. The calibration mode code is demonstrated in figure 5-5. The frame of the calibration mode is first obtaining raw data; second calibrating data; third transmitting data.

```
537 void calibrationMode(void)
538 {
539
540 if (GetSample)
541 {
542     UINT8 size = 0;                 // The amount of bytes being output
543     UINT8 data[MAX_STR_LENGTH];     // Data to output
544     TBOOL ASCIImode = (TBOOL)((Config[2] >> 3) & 0x01);
545     UINT8 i=0;
546     GetSample = bFALSE;             // Reset flag
547 if(SENSOR_GetSamples(CurrentAccelSamples, CurrentGyroSamples, CurrentMagSamples, (Config[2] & 0x04)))
548     {
549
550     if (ASCIImode)
551         { // output bytes in ASCII
552             UINT8 tempData[MAX_STR_LENGTH] = "";
553             const UINT8 sensorStringSize = 21; // 21 is the size of '-#####,-#####,-#####,'
554
555             if ((Config[2] >> 2) & 0x01) // Output accel readings?
556             {
557                 Cal_SENSOR_Reading2String(CurrentAccelSamples, (char *)tempData);
558                 for (i = 0; i < sensorStringSize; i++)
559                 {
560                 data[size++] = tempData[i];
561                 }
562             }
563         }
564
565     else// output raw bytes
566         {
567             if ((Config[2] >> 2) & 0x01) // Output accel readings?
568             {
569                 for (i = 0; i < 6; i++)
570                 {
571                     data[size++] = CurrentAccelSamples[i];
572                 }
573             }
574         }
575     if (ASCIImode)
576         { // Add
577             size++;
578             data[size++] = '\r';
579             data[size] = '\n';
580         }
581
582     size++;
583     // data packet is finished being constructed
584     sendData(data, size);
585     }
586 }
587 }
```

Figure 5-5 Calibration Mode

Thus, the central procedure in the calibration mode is 3 functions. The first one is *SENSOR_GetSamples()* (Line 547) which is used to obtain the raw accelerometer data. The second one is *Cal_SENSOR_Reading2String()* (Line 557) which is to modify raw data to a readable and calibrated data. The last one is *sendData()* (Line 584) which is used to transfer data from the IMU to the PC.

The definition of *SENSOR_GetSamples()* is illustrated in figure 5-6.

```
422 TBOOL SENSOR_GetSamples(UINT8 * const accelSamples, UINT8 * const gyroSamples, UINT8 * const magSamples, const UINT8 settings)
423 {
424
425   if (settings & 0x01) // Get mag readings?
426   { // Set Magnetometer to Single-Measurement Mode, NOTE: This is Done first as the magnetometer takes time to
427     // create a sample. The acquiring of the sample from the magnetometer happens at the end of GetSamples
428     if (!(SENSOR_WriteByte(MPU9150_MAG_ADDRESS, MPU9150_RA_MAG_CNTL, 0x01))) // Single-Measurement Mode
429     {
430       return (bFALSE); // Write failed
431     }
432   }
433   if ((settings >> 2) & 0x01) // Get accel readings?
434   { // Get Accelerometer readings: X_H, X_L, Y_H, Y_L, Z_H and Z_L (in that order)
435
436
437
438
439     if (!(SENSOR_ReadArray(MPU9150_ADDRESS, MPU9150_RA_ACCEL_XOUT_H, accelSamples, 6)))
440     {
441       return (bFALSE); //Read failed
442     }
443
444
445   }
446   if ((settings >> 1) & 0x01) // Get gyro readings?
447   { // Get Gyroscope readings: X_H, X_L, Y_H, Y_L, Z_H and Z_L (in that order)
448     if (!(SENSOR_ReadArray(MPU9150_ADDRESS, MPU9150_RA_GYRO_XOUT_H, gyroSamples, 6)))
449     {
450       return (bFALSE); // Read failed
451     }
452   }
453   if (settings & 0x01) // Get magnetometer readings?
454   { // Get Magnetometer readings: X_L, X_H, Y_L, Y_H, Z_L and Z_H (in that order)
455     // NOTE: Magnetometer readings are stored in MPU-9150 as X_L, X_H... (ie reverse of gyro and accel)
456     UINT8 tempMagSamples[6];
457     INT16 temp;
458     UINT8 dataReady = 0x00;
459
460     // Wait until data is ready
461     while (dataReady != 0x01)
462     {
463       if (!(SENSOR_ReadByte(MPU9150_MAG_ADDRESS, MPU9150_RA_MAG_STATUS_1, &dataReady))) // Get data ready status
464       {
465         return (bFALSE); // Read failed
466       }
467     }
468     // Data is ready, hence get Magnetometer sample
469     if (!(SENSOR_ReadArray(MPU9150_MAG_ADDRESS, MPU9150_RA_MAG_XOUT_L, tempMagSamples, 6)))
470     {
471       return (bFALSE); // Read failed
472     }
473
```

Figure 5-6 Sensor_GetSamples

This function has 4 parameters. The first three are pointers and the last one is an 8-bit constant integer. This function can obtain data from all the sensors in the IMU (three parameters are 3 data correspondingly) while in the calibration mode, the only useful parameter is the first one which appears in line 439 and the last one which is used as a mark. The only useful function in *Sensor_GetSamples* in calibration mode is from line 439 to line 442. The line 439 *SENSOR_ReadArray()* is a function that reads data from the accelerometer. This function has 4 parameters. The first one defines the address of the sensor (which sensor should be read). The second one defines the register address (which register should be read inside the sensor). The third one defines the address of the data (where should be read inside register). The fourth is the data size. Therefore, this function clearly finds the data that is wanted. In the corresponding library, the accelerometer address is 0x69 and re-defines as *MPU9150_ADDRESS*. The register address is started from 0x3B (*MPU9150_RA_ACCEL_XOUT_H*) to 0x40 (*MPU9150_RA_ACCEL_ZOUT_L)*. This function is defined in figure 5-7. The useful

line is from 682 to 685. Because the MPU9150 communicates with the microcontroller through I2C, this needs another function (*I2C_ReadArray*) inside MPU9150 to read the data. The Function *I2C_ReadArray* is illustrated in figure 5-8.

```
677 TBOOL SENSOR_ReadArray(const UINT8 slaveAddress, const UINT8 regAddress, UINT8 * const dataRx, const UINT8 size)
678 {
679   // Ensure that over 1.3 us (t_buf min) has passed since previous I2C function call
680   __delay_cycles(20); // 20/8MHz = ~2.5us delay
681   // Check for valid input conditions
682   if ((slaveAddress == MPU9150_ADDRESS) && (regAddress <= MPU9150_MAX_RA))
683   { // Valid inputs for gyroscope/accelerometer
684     return (I2C_ReadArray(MPU9150_ADDRESS, regAddress, dataRx, size));
685   }
686   if ((slaveAddress == MPU9150_MAG_ADDRESS) && (regAddress <= MPU9150_MAG_MAX_RA))
687   { // Valid inputs for magnetometer
688     return (I2C_ReadArray(MPU9150_MAG_ADDRESS, regAddress, dataRx, size));
689   }
690
691   return (bFALSE); // Invalid input conditions
692 }
```

Figure 5-7 SENSOR_ReadArray

In the calibration mode, the useful part is from line 682 to line 685. These lines define the default reading address which is the accelerometer address. Also register address is the memory unit in the accelerometer. This function is actually a pointer function.

```
261 TBOOL I2C_ReadArray(const UINT8 slaveAddress, const UINT8 regAddress, UINT8 * const dataRx, const UINT8 size)
262 {
263   UINT8 count; // counts bytes that have been read
264
265   if (size <= 0) // Remove the assumption that size will have to be > 0
266   {
267     return (bFALSE); // size is not valid for use
268   }
269
270   UCB0I2CSA_L = slaveAddress;    // Configure the I2C address to use
271   SET(UCB0CTL1, UCTR);           // Transmit mode
272
273   // Follow the I2C protocol:
274   // MSP430 -> S-AD+W-----RA-----S-AD+R----------ACK------NACK-P
275   // SLAVE  -> -------ACK----ACK--------ACK-DATA-----DATA-------
276   // Step:     1 2    3   4  5   6 7    8   9****10* 11   12   13, steps 9 & 10 are repeated "size - 1" times
277
278   SET(UCB0CTL1, UCTXSTT);    // 1) Send start bit
279                              // 2) AD+W automatically sent
280   if (!(ackReceived()))
281   {
282     return (bFALSE); // NACK was received, therefore transfer failed
283   }
284                              // 3) ACK automatically received
285   UCB0TXBUF = regAddress;    // 4) Send register address
286   if (!(ackReceived()))
287   {
288     return (bFALSE); // NACK was received, therefore transfer failed
289   }                            // 5) ACK was automatically received
290   CLEAR(UCB0CTL1, UCTR);     // Change to Receive mode
291   SET(UCB0CTL1, UCTXSTT);    // 6) Send RESTART (ie START) bit
292   CLEAR(UCB0IFG, UCTXIFG);   // Clear Transmit flag
293                              // 7) AD+R automatically sent
294   while (UCB0CTL1 & UCTXSTT) // UCTXSTT is cleared when an ACK or NACK has been received
295   { }
296   if (UCB0IFG & UCNACKIFG) // Check if it was an ACK or a NACK
297   { // NACK was received, therefore send STOP and exit
298     SET(UCB0CTL1, UCTXSTP);    // Send STOP bit and release SCL (which is held low after receiving a NACK)
299     CLEAR(UCB0IFG, UCTXIFG);   // Clear Tx interrupt flag
300     CLEAR(UCB0IFG, UCNACKIFG); // Clear NACK flag
301     return (bFALSE);
302   }
303                              // 8) ACK automatically received
304   // Read "size - 1" bytes of data as the last read will be done after the STOP bit is sent
305   for (count = 0; count < (size - 1); count++)
306   {
307     while(!(UCB0IFG & UCRXIFG))  // Wait for DATA to finish being received
308     { }                          // 10*) ACK is sent when UCRXIFG is set
309     *(dataRx+count) = UCB0RXBUF; // 9*) Receive DATA
310   }
311   SET(UCB0CTL1, UCTXSTP);        // 12) & 13) Send NACK & STOP                          // NOTE: DATA will finish being received before stopping
312   CLEAR(UCB0IFG, UCTXIFG);       // Clear Tx interrupt flag
313   while(!(UCB0IFG & UCRXIFG))    // Wait for DATA to finish being received
314   {}
315   *(dataRx+(size-1)) = UCB0RXBUF;// 11) Receive DATA
316
317   return (bTRUE);            // NOTE: Steps 12) and 13) were done above step 11)
318 }
```

Figure 5-8 I2C_ReadArray

The function *I2C_ReadArray* is the direct function that obtains information from the accelerometer. The line 270 in figure 4 "*UCB0I2CSA_L = salveAddress"* defines the reading address of the sensor. The line 285 *"UCB0TXBUF = regAddress"* defines the reading address of the registers. The line 315 *"*(dataRx + (size-1)) = UCB0RXBUF"* defines the location where the data comes from. Therefore, when the user calls this function *I2C_ReadArray,* the system will return these 4 parameters which indicate the address of the data. The registers of I2C are demonstrated in figure 5-9. It is simple to find register *UCB0TXBUF, UCB0I2CSA, UCB0RXBUF* and their functionalities.

Offset	Acronym	Register Name	Type	Access	Reset
00h	UCBxCTLW0	USCI_Bx Control Word 0	Read/write	Word	0101h
00h	UCBxCTL1	USCI_Bx Control 1	Read/write	Byte	01h
01h	UCBxCTL0	USCI_Bx Control 0	Read/write	Byte	01h
06h	UCBxBRW	USCI_Bx Bit Rate Control Word	Read/write	Word	0000h
06h	UCBxBR0	USCI_Bx Bit Rate Control 0	Read/write	Byte	00h
07h	UCBxBR1	USCI_Bx Bit Rate Control 1	Read/write	Byte	00h
0Ah	UCBxSTAT	USCI_Bx Status	Read/write	Byte	00h
0Bh		Reserved - reads zero	Read	Byte	00h
0Ch	UCBxRXBUF	USCI_Bx Receive Buffer	Read/write	Byte	00h
0Dh		Reserved - reads zero	Read	Byte	00h
0Eh	UCBxTXBUF	USCI_Bx Transmit Buffer	Read/write	Byte	00h
0Fh		Reserved - reads zero	Read	Byte	00h
10h	UCBxI2COA	USCI_Bx I2C Own Address	Read/write	Word	0000h
12h	UCBxI2CSA	USCI_Bx I2C Slave Address	Read/write	Word	0000h
1Ch	UCBxICTL	USCI_Bx Interrupt Control	Read/write	Word	0200h
1Ch	UCBxIE	USCI_Bx Interrupt Enable	Read/write	Byte	00h
1Dh	UCBxIFG	USCI_Bx Interrupt Flag	Read/write	Byte	02h
1Eh	UCBxIV	USCI_Bx Interrupt Vector	Read	Word	0000h

Figure 5-9 Registers of I2C

Generally, the parameters of *SENSOR_GetSamples(), SENSOR_ReadArray()* are pointer functions (indirect function). When the function *SENSOR_GetSamples()* is called, the system can visit the registers *UCB0TXBUF, UCB0I2CSA, UCB0RXBUF* to obtain the accelerometer data through the parameter pointers.

The second function in calibration mode is *Cal_SENSOR_Reading2String().* This function converts the raw accelerometer data to a calibrated and readable string data. The function is defined as follow:

```
536 void Cal_SENSOR_Reading2String(UINT8 * const samples, char * const stringOut)
537 {
538   char fullString[25] = "";
539   // store samples (signed 16-bit 2's complement)
540   INT16 Xsample = ((samples[0] << 8) | samples[1]); // X sample
541   INT16 Ysample = ((samples[2] << 8) | samples[3]); // Y sample
542   INT16 Zsample = ((samples[4] << 8) | samples[5]); // Z sample
543
544       float CalX = 0.6075E-4 * Xsample + 0.0374;   // Calibrate X axis
545       float CalY = 0.6058E-4 * Ysample - 0.0144;   // Calibrate Y axis
546       float CalZ = 0.6162E-4 * Zsample - 0.0729;   // Calibrate Z axis
547
548
549   // Append X part
550       Calreading2String(CalX, fullString);
551   // Append Y part
552       Calreading2String(CalY, fullString);
553   // Append Z part
554       Calreading2String(CalZ, fullString);
555
556   // Copy to output string
557   strcpy(stringOut, fullString);
558 }
```

Figure 5-10 Cal_SENSOR_Reading2String

This function has two steps. The first step is calibration and the second step is converting binary data to a readable string data. The MPU9150 accelerometer is an 8-bit sensor. Therefore, the data collected from the accelerometer is an 8 bits binary data. Each axis's acceleration consists of 2 bytes which is high byte and low byte. Such as X direction consists of X high and X low. Each byte is 8 bits. Therefore, it is necessary to consist the two byte to a 16 bits data. This 16 bits data is the real acceleration (binary). The line 540 to line 542 consists low byte and high byte in each axis. After that, the raw accelerations are stored in the variables *Xsample, Ysample, Zsample* in 16 bits binary style.

It is time to adopt the calculated results in table 5-2. The Method B performs better and thus it supports the calibration system in the IMU. Next, the function *Calreading2String* converts 16 bits data to a string data. This function is defined in figure 5-11.

```
90 void Calreading2String(float reading, char * const fullString)
91 {
92   char tempString[8] = "";
93   char digit[2] = "0"; // Used to append individual characters by changing the digit, then using strcat()
94   float tempNum;
95   float a,b,c;
96
97   if (reading < 0) // Negative number
98   { // Append - sign to string and make number positive
99     digit[0] = '-';
100    strcat(tempString, digit);
101    if (reading == -32768)
102    {
103      reading = 32767; // This is the maximum positive number for an INT16, hence -32768 is treated as -32767
104    }
105    else // Any other negative number just needs a multiplication of -1
106    {
107      reading *= -1;
108    }
109  }
110  else // Positive number
111  { // Append a + sign to string to keep readings in line when output
112    digit[0] = '+';
113    strcat(tempString, digit);
114  }
115
116  tempNum = (float)reading;
117
118  digit[0] = (UINT8)(tempNum + 0x30); // The integer part
119  strcat(tempString, digit);          // add to string
120
121  digit[0] = '.';                      // Decimal point
122  strcat(tempString, digit);
123
124  a=tempNum;
125  digit[0] = (UINT8)((a-(UINT8)a)*10 + 0x30);    // The first decimal place
126  strcat(tempString, digit);
127
128  b=(a-(UINT8)a)*10;
129  digit[0] = (UINT8)((b-(UINT8)b)*10 + 0x30);    // The second decimal place
130  strcat(tempString, digit);
131
132  c=(b-(UINT8)b)*10;
133  digit[0] = (UINT8)((b-(UINT8)b)*10 + 0x30);    // The third decimal place
134  strcat(tempString, digit);
135
136  digit[0] = ',';
137  strcat(tempString, digit); // The string is complete
138
139  strcat(fullString, tempString); // Append created string
140 }
141
```

Figure 5-11 Calreading2String

If the results of the calibration on the screen is "-6000", this kind of data string is defined by this function. At the beginning of this function, two variables constitute the frame of the string. *"tempString[8]"* defines the length of the string is 8 bits. The *digital[2]* defaults the string is "00000" (*"S00000,\0"* where *"S"* is the sign of data ("+" or "-") and *"\0"* is an end label). The variable *"tempNum"* is the accelerometer binary raw data and *a,b,c* are used to define the decimal places. After these variables definition, 2 judgement sentences are used to add a positive or negative mark before the numbers.

At the end of this function, it defines the format of the data string. The first bit, integer place, is defined by *"digital[0] = (UNIT8)(tempNum + 0x30)"*. *tempNum* is the raw accelerometer data and *"0x30"* is "0" in the ASCII code. Therefore, this line is to convert a 16 bits raw data to an ASCII code which can be displayed on the screen. The second bit is a decimal point ".". The third bit is the first decimal place. This bit is defined by *"a=tempNum"* and *"digital[0] = (UNIT8)((a-(UNIT8)a)*10 + 0x30)"* . *"a=tempNum"* is simple to understand but notice that the variable *"a"* is a float type. Thus the *"a-(UNIT8)a"* is using float *"a"* to minus its integer part. For example, if *"a"*

is “1.256”, this *“a-(UNIT8)a”* is “1.256-1=0.256”. And thus “*a-(UNIT8)a)*10”* equals to “2.56”. And *“(UNIT8)((a-(UNIT8)a)*10 + 0x30)”* equals to string “2”. The fourth bit and fifth bit are similar with the third bit definition. Finally, this function *Calreading2String()* is to convert its first parameter float type data to a second parameter readable string.

The last function in the calibration mode is *sendData().* This function is used to transmit data from USB or Bluetooth. Its definition is illustrated in figure 5-12.

```
929 void sendData(const UINT8 * data, const UINT8 size)
930 {
931   if (UsbConnected)
932   {
933     cdcSendDataInBackground((UINT8*)data, size, CDC0_INTFNUM, 0);  // Send data over USB
934   }
935   else
936   {
937     UART_SendData(data, size);  // Send data over Bluetooth
938   }
939 }
```

Figure 5-12 sendData

In this function block, there are two functions. One is *cdcSendDataInBackground().* The other is *UART_SendData().* The parameter *UsbConnected* in line 931 is a mark that is used to indicate whether the USB has already been connected. If the USB has connected, using the USB to transmit data, otherwise, using the Bluetooth to transmit data.

5.6 Fine Calibration

5.6.1 Fine Classical Calibration

The classical calibration method is achieved by an A4 paper roll which is demonstrated in figure 5-4. Although its accuracy is enough to use, the need to improve of its accuracy as much as possible is considerable. Therefore, a new program has been designed. The program adopts the table lamp as the IMU support. However, this time the program will not use random positions to collect data. The procedures are as follows:

1. Fix the IMU on the table lamp
2. Press button 1 on the IMU to open Calibration Mode
3. Rotate table lamp joint and make sure that the Z positive axis is aligning to gravity direction approximately. The Z accelerometer that is demonstrated on the PC screen should be around 1g.

4. At this time, adjust IMU position slightly around current position many times and observe Z accelerometer changing. Find the highest Z accelerometer data in these positions and stop adjusting the position.
5. Press button 1 again to return Sensor Mode.
6. Record the highest ADC value of Z axis.
7. This highest ADC value equals to “1g”.
8. Rotate table lamp joint and make Z negative axis align to gravity direction approximately. Z accelerometer that is demonstrated on the PC screen should be around “-1g”. Repeat procedure 4, 5, 6. The highest ADC value equals to “-1g”.

The program is illustrated in figure 5-13. The axis X and axis Y have the same procedure as axis Z. When the number reaches to the highest or lowest, it means that this number should represent 1g or -1g because the accelerometer measurement range is 2g and the highest acceleration is 1g.

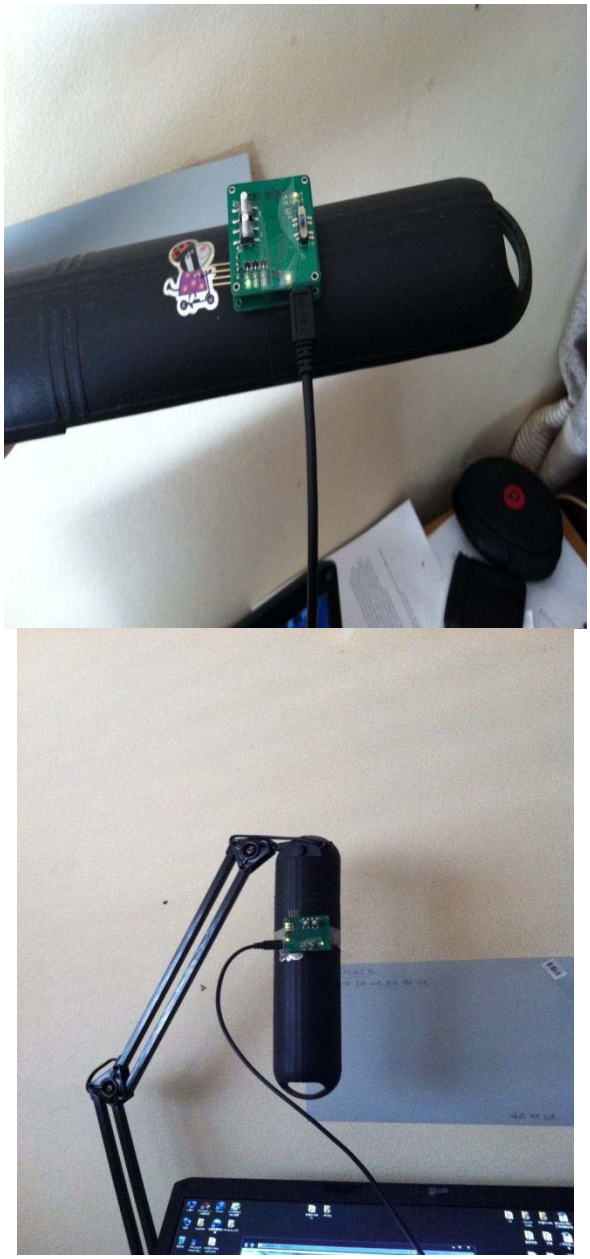


Figure 5-13 Refined Calibration

Table 5-3 ADC Value in 3 Axes

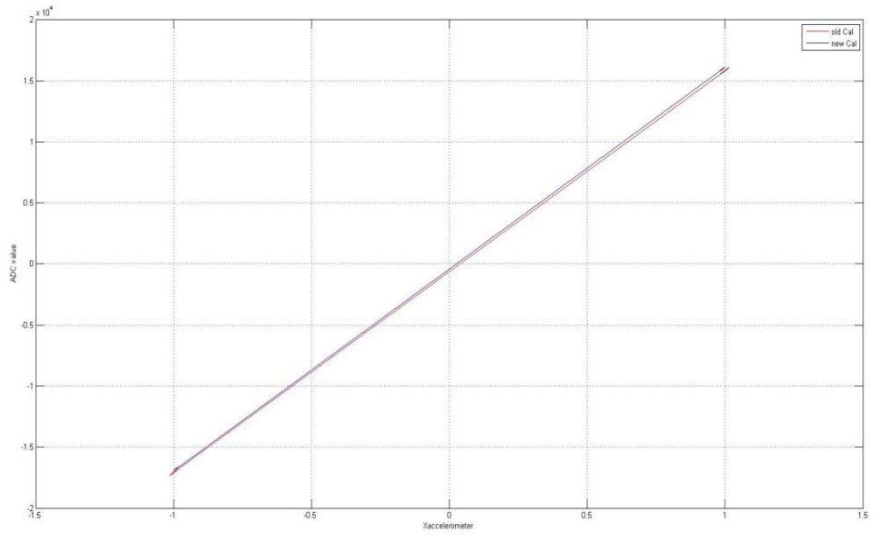
The highest ADC value in 3 axes	
$X_n = -16916$	$X_p = 16136$
$Y_n = -16520$	$Y_p = 16756$
$Z_n = -15472$	$Z_p = 17556$

Adopt equation (5-19), (5-20), (5-21) to obtain$S_x, S_y, S_z, O_x, O_y, O_x$. Use equation (5-13), (5-14), (5-22) to calculate error and RMS. The results are as follows:

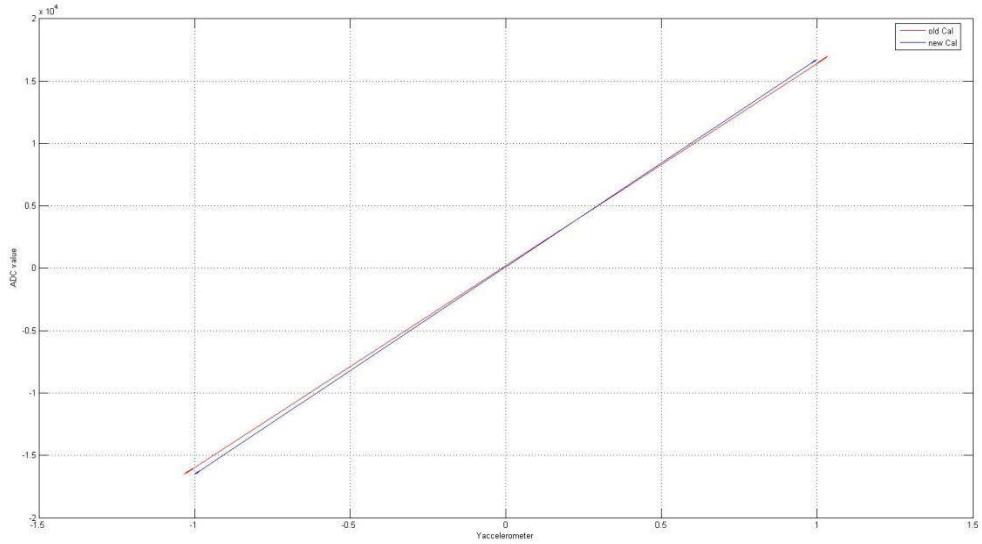
Table 5-4 Results of Refined Calibration

$S_x = 6.0511E-05$	$O_x = 0.0236$
$S_y = 6.0103E-05$	$O_y = -0.0071$
$S_z = 6.0555E-05$	$O_z = -0.0631$
$RMS = 0.0343$	

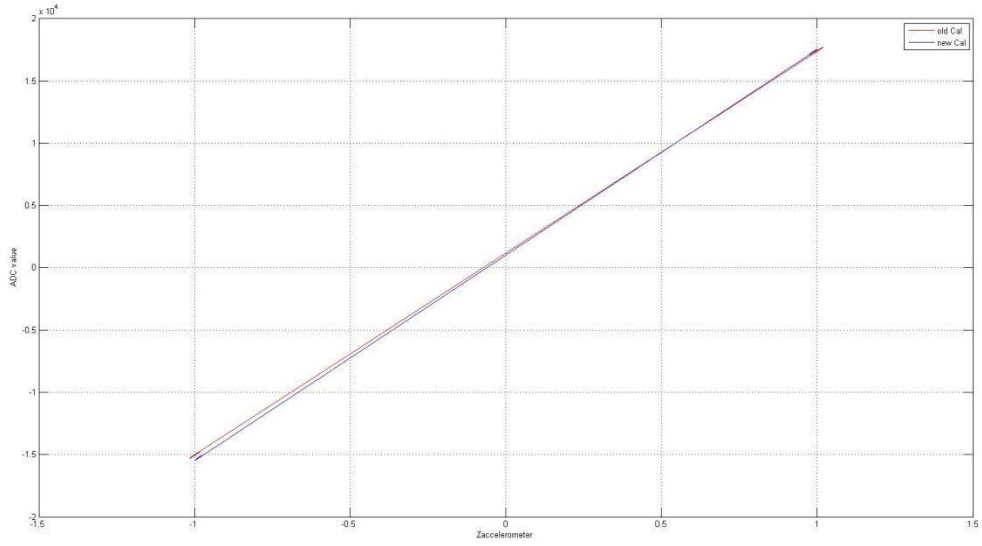
It is noticed that the refined classical calibration RMS is almost half of the rough Classical Calibration RMS. This means that the error has decreased 49.7%. A comparison of rough Classical Calibration and Refined Classical Calibration is illustrated in figure 5-14.



a. X axis



b. Y axis



c. Z axis

Figure 5-14 A comparison of rough calibration and refined calibration

The blue line is the refined calibration and the red line is the rough calibration. The blue line is more close to horizontal axis. This means the refined calibration method actually improve accuracy.

5.6.2 Fine Auto-Calibration Improvement

The rough auto calibration adopts the nonlinear-least square method. However, the initial value of nonlinear-least square method is not a hard rule. Therefore, the initial value is set to $5 \times [1,1,1,1,1,1]$ and the following result s are obtained:

Table 5-5 Rough Auto-calibration Method Results

Parameter	Auto Method
S_x	0.6217E-04
S_y	0.6170E-04
S_z	0.6077E-04
O_x	0.0421
O_y	-0.0148
O_z	-0.0748

These results are used to re-build the code and find that the RMS (Root Mean Square) is $0.074g^2$.

In order to improve the auto-calibration accuracy, this thesis will try to use a different initial value and find the relationship between the initial value and the RMS. After the initial value being set, the rough Auto-calibration experiment is processed again in order to obtain ADC value under this initial value. After that, the calculated parameters would be used in converting ADC value to a "g" value according to Frosio's model.

The experiments are illustrated as follow:

1. Use different initial value to obtain S and O through the functions in figure 5-15.

```
function Err=acc(X,Vnew)
kx=X(1);
ox=X(2);
ky=X(3);
oy=X(4);
kz=X(5);
oz=X(6);

Vx=Vnew(:,1);
Vy=Vnew(:,2);
Vz=Vnew(:,3);

Err=sum(((kx*Vx+ox).^2+(ky*Vy+oy).^2+(kz*Vz+oz).^2-1).^2);
return
```

```
clear all
close all
load Vnew

gain=16384;
gain=max(max(Vnew))

%x0 = [6e-5 0.01 6e-5 0.01 16e-5 0.01 ];
x0 = 4*ones(1,6);


X=lsqnonlin(@(x)acc(x,Vnew/gain),x0);

kx=X(1)/gain
ky=X(3)/gain
kz=X(5)/gain

ox=X(2)
oy=X(4)
oz=X(6)
% to recover the real k and o
```

Figure 5-15 Functions of Nonlinear Least Square Method

Table 5-6 Initial Values and their S, O values

Initial value	S_x	S_y	S_z	O_x	O_y	O_z	RMS
10	7.9740e-05	7.9562e-05	5.6433e-05	0.0913	-0.0049	-0.0826	0.4481
9.5	7.8027e-05	7.7809e-05	5.6151e-05	0.0878	-0.0060	-0.0836	0.2393
9	7.6352e-05	7.6093e-05	5.5952e-05	0.0841	-0.0071	-0.0846	0.1615
8.5	7.4720e-05	7.4418e-05	5.5844e-05	0.0805	-0.0082	-0.0858	0.1404
8	7.3134e-05	7.2787e-05	5.5838e-05	0.0767	-0.0094	-0.0869	0.1609
7.5	7.1600e-05	7.1207e-05	5.5947e-05	0.0728	-0.0106	-0.0881	0.1629
7	7.0127e-05	6.9686e-05	5.6183e-05	0.0687	-0.0119	-0.0892	0.1556
6.5	6.8728e-05	6.8239e-05	5.6555e-05	0.0646	-0.0131	-0.0900	0.1379
6	6.7420e-05	6.6886e-05	5.7065e-05	0.0603	-0.0142	-0.0903	0.1556
5.5	6.6227e-05	6.5651e-05	5.7699e-05	0.0561	-0.0151	-0.0897	0.1262
5	6.5168e-05	6.4562e-05	5.8415e-05	0.0520	-0.0158	-0.0879	0.1107
4.5	6.4251e-05	6.3632e-05	5.9140e-05	0.0483	-0.0160	-0.0847	0.0819
4	6.3463e-05	6.2855e-05	5.9794e-05	0.0453	-0.0158	-0.0806	0.0449
3.5	6.2774e-05	6.2206e-05	6.0339e-05	0.0430	-0.0154	-0.0767	0.0576
3	6.2164e-	6.1670e-	6.0784e-	0.0414	-0.0149	-0.0745	0.0321

	05	05	05				
2.5	6.1639e-05	6.1260e-05	6.1106e-05	0.0401	-0.0143	-0.0728	0.0233
2	6.1170e-05	6.1035e-05	6.1337e-05	0.0395	-0.0137	-0.0714	0.0191
1.5	6.0427e-05	6.0554e-05	6.1677e-05	0.0373	-0.0132	-0.0686	0.0177
1.4	6.0481e-05	6.0546e-05	6.1718e-05	0.0373	-0.0136	-0.0710	0.0168
1.3	6.0534e-05	6.0551e-05	6.1698e-05	0.0374	-0.0136	-0.0711	0.0150
1.2	6.0599e-05	6.0550e-05	6.1676e-05	0.0374	-0.0137	-0.0712	0.0154
1.15	6.0704e-05	6.0525e-05	6.1599e-05	0.0373	-0.0135	-0.0697	0.0156
1.13	6.0704e-05	6.0530e-05	6.1636e-05	0.0373	-0.0137	-0.0711	0.0152
1.12	1.0396e-07	1.4037e-05	1.3740e-05	0.9804	2.1163e-04	-0.0054	0.0175
1.1	9.4367e-08	1.3840e-05	1.3549e-05	0.9821	2.0108e-04	-0.0053	0.0200
1	8.8846e-08	1.3475e-05	1.3191e-05	0.9831	1.9894e-04	-0.0052	0.0174
0.5	3.2658e-09	-1.8305e-09	2.3434e-08	0.6796	0.6332	0.3704	0.0013
0	-1.0714e-10	-1.0041e-10	-5.9121e-12	0.5772	0.5776	0.5773	0.0012
-0.5	-3.2658e-09	1.8305e-09	-2.3434e-08	-0.6796	-0.6332	-0.3704	0.0011
-1	-8.8846e-08	-1.3475e-05	-1.3191e-05	-0.9831	-1.9894e-04	0.0052	0.0210
-1.5	-6.0427e-05	-6.0554e-05	-6.1677e-05	-0.0373	0.0132	0.0686	0.0287
-2	-6.1170e-05	-6.1035e-05	-6.1337e-05	-0.0395	0.0137	0.0714	0.0317

2. Use these S and O to re-build the code and make some tests. Calculate RMS through these tests.

Table 5-7 Initial Values and their RMS values

Initial value	RMS
10	0.4481
9.5	0.2393
9	0.1615
8.5	0.1404
8	0.1609
7.5	0.1629
7	0.1556
6.5	0.1379
6	0.1556

5.5	0.1262
5	0.1107
4.5	0.0819
4	0.0449
3.5	0.0576
3	0.0321
2.5	0.0233
2	0.0191
1.5	0.0177
1.4	0.0168
1.3	0.0150
1.2	0.0154
1.15	0.0156
1.13	0.0152
1.12	0.0175
1.1	0.0200
1	0.0174
0.5	0.0013
0	0.0012
-0.5	0.0011
-1	0.0210
-1.5	0.0287
-2	0.0317

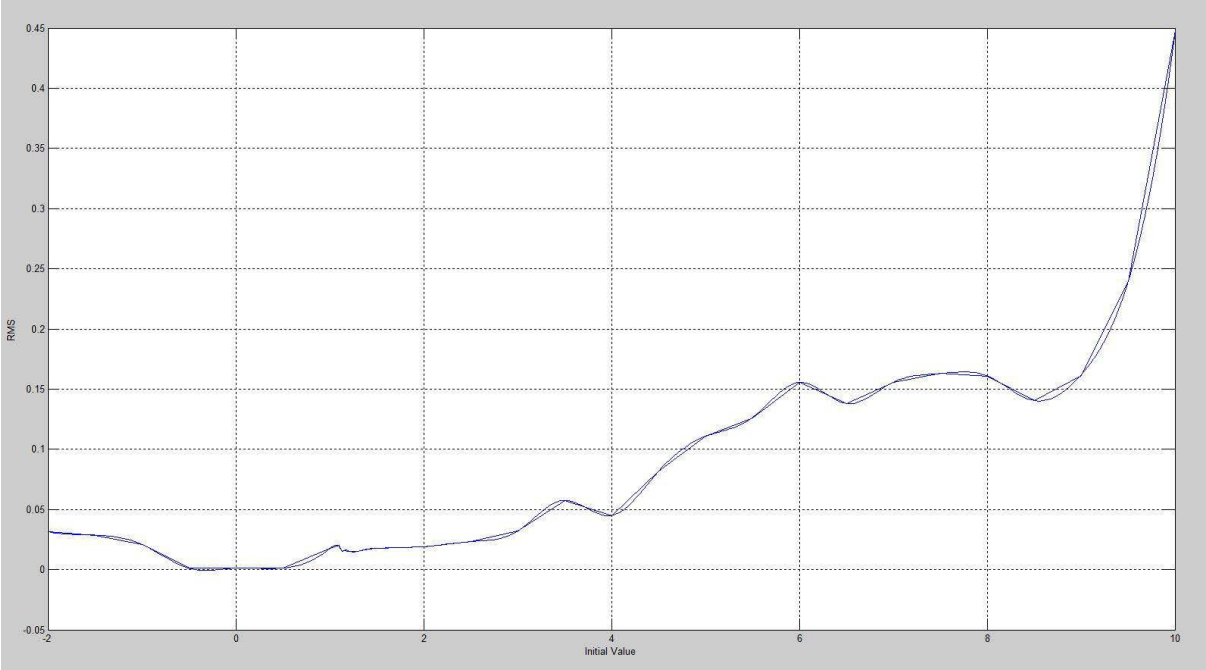


Figure 5-16 Relationship between Initial Values and their RMS values

From the experiments results in figure 5-16, when initial value decreases, the RMS value decreases. However, in the experiment, the function in figure 5-16 is not working if the initial value is lower than 1.123. At the same time, the RMS value of 1.2 is lower

than 0.0123. Therefore, a possible situation is that this curve converges to a value between 1.123 and 1.2. Nevertheless, this thesis will not discuss this problem in deepth. This thesis will use 1.2 as a best solution. Its RMS value is $0.0106g^2$.

5.7 Mixed Calibration Exploration

In the previous introduction, the auto calibration and the classical calibration has been discussed. However, the question occurs about how a mixed calibration performs. A mix calibration is combining the auto-calibration and the classical calibration. In other words, using the auto-calibration's gain and the classical calibration's offset or using the classical calibration's gain and the auto-calibration's offset. However, there is a problem that should be noted. In the auto calibration method, an initial value has a corresponding calibration result and also a corresponding calibration accuracy. The accuracy that mixes different auto-calibration results and classical calibration results must be different. Therefore, it is necessary to discuss which initial value should be chosen before mixing.

Because of this circumstance, the auto calibration parameters can be decided under an initial value 1.2 and the classical calibration method parameters are the fine classical calibration results. The solutions of Mixed-calibration are:

Table 5-8 Parameters of Auto-Calibration

$S_x = 6.0599e-05$	$O_x = 0.0374$
$S_y = 6.0550e-05$	$O_y = -0.0137$
$S_z = 6.1676e-05$	$O_z = -0.0712$

Table 5-9 Parameters of Classical Calibration

$S_x = 6.0511e-05$	$O_x = 0.0236$
$S_y = 6.0103e-05$	$O_y = -0.0071$
$S_z = 6.0555e-05$	$O_z = -0.0631$

It is viable to combine these two solutions and make a test. The first mixed calibration parameters are using auto-calibration's gain value and classical calibration's offset value. The test follows the previous auto-calibration method. The results are:

Table 5-10 Mix Calibration 1

$S_x = 6.0599e-05$	$O_x = 0.0236$
$S_y = 6.0550e-05$	$O_y = -0.0071$
$S_z = 6.1676e-05$	$O_z = -0.0631$
$RMS = 0.0174g^2$	

The second mix calibration combines auto-calibration's offset value and classical calibration's gain value. The results are:

Table 5-11 Mix Calibration 2

$S_x = 6.0511e-05$	$O_x = 0.0374$
$S_y = 6.0103e-05$	$O_y = -0.0137$
$S_z = 6.0555e-05$	$O_z = -0.0712$
$RMS = 0.0184g^2$	

These two mixed calibration can reach an acceptable accuracy. This RMS is in the range between the auto-calibration RMS and the classical calibration RMS. If the classical calibration RMS can be regarded as a reference rule, the first mixed calibration accuracy rises from 0.0343 to 0.0174 (49.27%) while the second one rises from 0.0343 to 0.0184 (46.35%). Although the mixed calibration accuracy cannot exceed the auto calibration accuracy, it is found that the gain values have more effect on RMS than the offset value.

5.7 Chapter Conclusion

The classical method performs with higher accuracy than the auto-calibration method in the rough calibration. However, in the fine calibration, the accuracy of the classical method improves 50.2%. The RMS value decreases from 0.069 to 0.0343. On the other hand, the fine auto calibration method accuracy improves 85.6% from 0.074 to 0.0106. As a result, the auto calibration method can be selected in the research.

This chapter discusses two different calibration methods. In the rough calibration experiments, the classical calibration method performs better while in the fine calibration experiments, the auto-calibration method performs better. This thesis gives

the best solution to the argument about which method is better. And also, this chapter provides a useful auto-calibration method and its matlab solution which can be used by future research directly in a similar process. In the last part, this chapter tries to explore viability and possibility of mixed calibration for future research. In conclusion, this chapter has justified the selection of the auto-calibration method as the first choice for the accelerometer calibration.

VI. CONCLUSION AND FUTURE RESEARCH

6.1 Conclusion

The thesis designs and improves an uncompleted medical device that is used in monitoring of patients movements. In the thesis, the system of the device is developed and the integrated sensor is calibrated. The thesis mainly discusses and analyses different results that are obtained by different calibration methods and makes a comparison between those methods. Based on this, a fine calibration is performed to improve the calibrating accuracy and an evaluation of the fine calibration is given. Furthermore, parameters which affect the accuracy are explored at the end of the thesis. The algorithm and experimental method that are presented in the calibration process have a number of advantages and, most importantly, provide a strong basis of knowledge for further improvement and research in the commercial product. Finally, the calibration method is completed adopted by our IMU.

This thesis has three sections in total. The first section introduces the preparation work, such as basic knowledge collection, accumulation and relevant information of selecting and organizing. This section includes chapter 1 and chapter 2. In this section, the background of the IMU, current application of the IMU, the necessity of calibration, the feasibility of current calibration methods, the inspiration from other similar research, the basic knowledge of the IMU structure, micro-controllers and the principles of the accelerometer calibration have been presented carefully. This section provides a strong theoretical support for the purpose of the research.

The second section which includes chapter 3 and 4 is the deeper research on the IMU. The IMU is dismantled in this section, not only the hardware but also the software. The physical structure and the electronic components of the IMU have been illustrated in chapter 3 in detail for the purpose of understanding its functionalities and features. Moreover, the core components and the communication protocols in the IMU are illustrated in chapter 4 to provide an understanding of its software structure. In this

chapter, Texas Instruments (TI) serial microcontrollers and its development tool Code Composer Studio relevant knowledge is the most important information. This knowledge is the structure and panel of the calibration.

The last section of the thesis is chapter 5. This section describes the success of the calibration, the accuracy improvement, completing the IMU system, comparing two calibration methods and the discussion of future research. Most significantly, two different calibration methods are discussed. In the fine calibration of the classical method, its accuracy is improved by 50.2% and its RMS value decreases from 0.069 to 0.0343. The fine auto calibration method improves accuracy by 85.6%. Its RMS decreases from 0.074 to 0.0106. This result can be adopted in the final IMU application as well as the future calibrating operation. At the same time, the IMU system is developed based on this achievement. In the last part of this section, viability and possibility of mixed calibration is explored in order to find a relationship between the initial values and the parameters of the IMU accelerometer for preparations for future research.

This thesis describes the procedures of calibrating the accelerometer in the IMU critically and progressively. In particular, this simple, inexpensive, high accuracy and high efficiency calibration method is strongly recommended. The IMU accelerometer calibration method which is introduced in the thesis can serve in many similar commercial products. Furthermore, the comparison between the classical calibration and the auto-calibration will provide evidence to select the calibration method in future research.

6.2 Future Research

In future research, effects of the axes coupling will be a possible direction of the calibration research to improve the IMU accuracy. The three accelerometer axes have affected on each other. This is the axes coupling problem which also causes some error. During the experiments, the orthogonal axes have less error while the non-orthogonal axes lead to more error. Therefore, a better calibration method which has better equipment or algorithm can be one of the research directions.

Secondly, noise influences in the thesis are not discussed. However, during the experiments, the noise which comes from inside current or outside disturbance can also cause some error. In future research, a development algorithm should consider the noise

problem to improve its accuracy.

Finally, more initial values will be tested to find a better solution. The relationship between different initial values and the IMU accuracy will be another possible research direction.

REFERENCES

[1] M. Glueck, A. Buhmann, Y. Manoli, "*Autocalibration of MENS accelerometers*," in Instrumentation and Measurement Technology Conference(I2MTC),IEEE International Conference on, 2012, pp. 1788 - 1793.

[2] I. Frosio, S. Stuani, N.A. Borghese, "Autocalibration of MENS accelerometers," *Instrumentation and Measurement Technology Conference*, *Proceedings of the IEEE*, 2006, pp. 519-523.

[3] D. Hu, C. Zeng, H. Liang, "Autocalibration method of MEMS accelerometer," in *Mechatronic Science, Electric Engineering and Computer, International Conference on,* 2011, pp. 1348-1351.

[4] I. Frosio, F. Pedersini, N.A. Borghese, "Autocalibration of Triaxial MEMS Accelerometers With Automatic Sensor Model Selection," *IEEE Sensors J.*, vol. 12, no. 6, pp. 2100-2108.

[5] MPU9150 Product Specification Revision 4.0.PDF.http://www.invensens.com

[6] http://math.gmu.edu/~igriva/book/Appendix%20D.pdf

[7] M. Hwangbo, J. Kim, T. Kanade, " IMU Self-Calibration Using Factorization," *IEEE Transactions on Robotics, vol. 29, no. 2, Apr. 2013, pp 493-507.*

[8] F. Höflinger, J. Müller, R. Zhang, L. M. Reindl, W. Burgard, "A Wireless Micro Inertial Measurement Unit (IMU)," *IEEE Transactions on Robotics, vol. 62, no. 9, Sep. 2013, pp 2583-2595.*

[9] J. Marek, "Trends and challenges in modern MEMS sensor packages," in *Proc. Symp. DTIP*, 2011, pp. 1–3.

[10] G. Retscher and T. Hecht, "Investigation of location capabilities of four different smartphones for LBS navigation applications," in *Proc. Int.Conf. IPIN*, Nov. 2012, pp. 1–6.

[11] R. Dixon. (2012, Oct.). *Combo Motion Sensors Strike Gold as Revenue Rises More Than 700 Percent This Year*, iSuppli, Englewood, CO, USA [Online]. Available: http://www.isuppli.com/MEMS-and-Sensors/MarketWatch/Pages/Combo-Motion-Sensors- Strike-Gold-as-Revenue-Rises-More-than-700-Percent-This-Year.aspx

[12]H. Trah, J. Franz, and J. Marek, "Physics of semiconductor sensors," in *Advances in Solid State Physics*, vol. 39. New York, NY, USA: Springer-Verlag, 1999, pp. 25–36.

[13] M. Glueck, D. Oshinubi, P. Schopp, Y. Manoli, "Real-Time Autocalibration of MEMS Accelerometers," *IEEE Transaction on Instrumentation and Measurement, vol. 63, no. 1, Jan. 2014, pp.96-105.*

[14]MSP430x5xx and MSP430x6xx Family User's Guide.pdf. http://www.ti.com/lit/ug/slau208m/slau208m.pdf

[15]M. S. Conover, "Using accelerometers to quantify infant general movements as a tool for assessing motility to assist in making diagnosis of cerebral palsy, " *Master thesis*, available at http://scholar.lib.vt.edu/theses/available/etd-09232003-153717/, 2003.

[16]Ceramic Surface Mount Low Profile Quartz Crystals ABMM2 datasheet.pdf. http://www.alldatasheet.net/datasheet-pdf/pdf/193540/ABRACON/ABMM2.html

[17] 1.5A USB-FRIENDLY Li-Ion BATTERY CHARGER AND POWER-PATH MANAGEMENT IC BQ24074 datasheet.pdf. http://www.alldatasheet.net/datasheet-pdf/pdf/249608/TI/BQ24074.html

[18] J. Kelley, "How to Create a Printed Circuit Board (PCB)", *Department of Electrical & Computer Engineering Michigan State University.*

[19]JTAG Tutorial, available at http://www.corelis.com/education/JTAG_Tutorial.htm

[20]MMS228T datasheet.pdf available at http://www.farnell.com/datasheets/490883.pdf

[21]MTMM Serial modified through-hole header.pdf available at https://www.samtec.com/ftppub/pdf/mtmm_th.pdf

[22]MSP430™ Programming Via the JTAG Interface User Guide.pdf available at http://www.ti.com/lit/ug/slau320m/slau320m.pdf

[23]150 mA, Low-Noise LDO Voltage Regulator SPX5205 available at http://www.alldatasheet.net/datasheet-pdf/pdf/46048/SIPEX/SPX5205.html

[24]USB6B1 datasheet.pdf available at http://www.alldatasheet.net/datasheet-pdf/pdf/25576/STMICROELECTRONICS/USB6B1.html

[25] Bluetooth 11 Specifications Book.pdf available at http://www.bluetooth.com

[26]BT0417C datasheet.pdf available at http://mdfly.com/Download/Wireless/BT0417C_datasheet.pdf

[27]FM25W256 Datasheet.pdf available at http://www.alldatasheet.net/datasheet-pdf/pdf/151112/ETC1/FM25W256.html

[28]MPU9150A Product Specifications.pdf available at http://www.invensense.com/mems/gyro/documents/PS-MPU-9250A-01.pdf

[29]MSP430F5528 Datasheet.pdf http://www.alldatasheet.com/datasheet-pdf/pdf/390608/TI/MSP430F5528.html

[30] S. Sabatelli, M. Galgani, L. Fanucci, and A. Rocchi "A Double-Stage Kalman Filter for Orientation Tracking With an Integrated Processor in 9-D IMU", *IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT, VOL. 62, NO. 3, MARCH 2013*

[31] F. Höflinger, J. Müller, R. Zhang, L. M. Reindl and W. Burgard, "A Wireless Micro Inertial Measurement Unit (IMU)", *IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT, VOL. 62, NO. 9, SEPTEMBER 2013*

[32] S. Bakhshi, M. H. Mahoor and B. S.Davidson, "DEVELOPMENT OF A BODY JOINT ANGLE MEASUREMENT SYSTEM USING IMU SENSORS", *33rd Annual International Conference of the IEEE EMBS Boston, Massachusetts USA, August 30 - September 3, 2011*

[33] C. Hide, T. Botterill, M. Andreotti, "Low cost vision-aided IMU for pedestrian navigation".

[34]M. Glueck,R. B. GmbH, A. Buhmann, Y. Manoli "Autocalibration of MEMS accelerometers".

[35]A. Krohn, M. Beigl, C. Decker, U. K. orfer, P. Robinson and T. Zimmer, "Inexpensive and Automatic Calibration for Acceleration Sensors".

[36]M. Glueck, D. Oshinubi, P. Schopp and Y. Manoli, "Real-Time Autocalibration of MEMS Accelerometers*", IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT, VOL. 63, NO. 1, JANUARY 2014.*

[37] B. M. Scherzinger, "Precise robust positioning with intertial/GPS RTK," in Proc. 13th Tech. Meeting Satellite Division Inst. Navigat., Salt Lake City, UT, USA, 2000, pp. 1–9.

[38] A. Umeda, M. Onoe, K. Sakata, T. Fukushia, K. Kanari, H. Iioka, and T. Kobayashi. (2004). "Calibration of three-axis accelerometers using a three-dimensional vibration generator and three laser interferometers," Sens. Actuators A, Phys. [Online]. 114(1), pp. 93 101. Available: http://www.sciencedirect.com/science/article/pii/S0924424704001700

[39] M. Stakkeland, G. Prytz, W. Booij, and S. Pedersen, "Characterization of accelerometers using nonlinear Kalman filters and position feedback," IEEE Trans. Instrum. Meas., vol. 56, no. 6, pp. 2698–2704, Dec. 2007.

[40] The economic impact of stroke in Australia National Stroke Foundation 13 March 2013

[41] S.S.Doshi "Applications of inertial measurement unit in monitoring rehabilitation progress of arm in stroke supvivors" , Colorado State University, Master Thesis, 2011.

[42] S. Badwal, G. Turley," Measuring Human Hip Activity Using Inertial Measurement Unit Sensors", University of Warwick.