# DESIGN DOCUMENTATION OF THE LCIMU SYSTEM

By Andrew .T .Omagba

MAY 1, 2017

THE UNIVERSITY OF NEW ENGLAND

# Table of Contents

# Table of Figures

# 1.0 Introduction

## 1.1 Purpose

The purpose of the LCIMU design documentation, is to document and highlight the necessary information required to effectively articulate and define the LCIMU system architectural design. This would then serve as a road map or guideline for future development or modification of a low cost inertial motion capture system.

# 2 General Overview

The LCIMU system is a device that captures the upper limb motion (two upper limbs) of a human subject over a giving period. It samples the data captured at a sample rate of +60Hz.

In its current specification, the LCIMU system is implemented by a simple User application that allows data to be captured and stored on a profile.

The literary wordings of this design specification were revised several times before this final draft. The graphics and layout of the device and display screen as shown here merely illustrates the underlying functionality of the device.

This specification gives a description of a device system that is more data-centric and useful to a high-level or a low-level user. It simply details how an affordable customized motion capture device can be used to capture motion data, which can be used for research analysis.

## 2.1 Basic Scenarios and User Profiles

Different scenarios would help give a mental visualization of how people can use the device.

**Scenario 1: Gates**

Gates is a post grad student in the department of Biomathematics at the University of South Australia. He is doing a research on upper limb functionality in a controlled environment and he needs to come up with a model to identify and classify the behavioural activities of groups of individuals from different demographics. He has a limited budget and needs an affordable motion capture device that can enable him to achieve his goal. With his background, Gates fits the profile of a high-level user and the LCIMU can easily allow him to capture the necessary data required for him to begin his data identification and analysis. He straps on the device to the test individuals and begins the data capture of the selected behavioural activities (eating, texting, sleeping, walking etc.). He runs the program, logs the data, collects the data and uses his preferred means of data modelling to analyse the captured data.

**Scenario 2: Bolt**

Bolt is an undergrad in the department of biomedicine at the University of Canberra. He is interested in undergoing a study of the power spectral analysis of a human subject with stage two Parkinson's disease. He has a minimal budget and he is in search of a cheap but duly functional device that can allow for the inertial motion capture of a target subject. The individual has stage two Parkinson's; therefore, it means that he/she undergoes relative tremors on both sides of the body. The LCIMU will come in handy for Bolt as it would fit his budgetary needs and due to the movement restrictions of the subject, would allow for the capturing of data in a controlled environment.

## 2.2 Assumptions and Constraints

The LCIMU system would be designed with some basic assumptions and constraints inherent. However, these factors do not significantly impact the overall functionality of the resultant system.

Some of the assumptions factored into the system design and development are:

- There will be no significant effect from powering the system from a laptop in the alpha/beta phases.
- The voltage drop along the length of the cable carrying the data and power is not enough to affect the overall functionality of the system.
- There would be no significant signal contradiction from all the individual components (from different manufacturers) that make up the overall system.
- The system will be able to function over a prolonged period without any risk of overheating.

Some of the constraints of the system are:

- The User application will not support a password or username reminder functionality.
- The data from the LCIMU system can only be accessed when the program is run on a suitable JAVA IDE such as Intelli J, Eclipse, Netbeans etc..
- The range of the system is limited to the length of its connecting cables.
- The memory storage of the system is subjected to the limit of the data capture platform, in this case the Intelli J IDE.
- The overall system functionality is dependent on multiple software platforms thus making it overly robust.

## 3.0 Architectural Design

This document presents the architecture as a series of views; use case view, logical view, hardware design schematics and software build structure.

## 3.1 Use Case View

A description of the use case view of the LCIMU software architecture will be centred on the simple set of centrally-functional scenarios applicable to the LCIMU system. It will also describe the set of scenarios or use cases that have a significant architectural coverage.

The LCIMU use cases are:

- User Registers
- User Signs In
- User Starts the Application
- User Captures Data
- User Displays Data
- User Stops the Application
- User Logs Out

These use cases are initiated by the user. It is also important to point out here that the last two use cases (Display Data and Store Data) as listed above are automatically initiated by the first (Capture Data).

## 3.2 Architecturally-Centric Use Cases

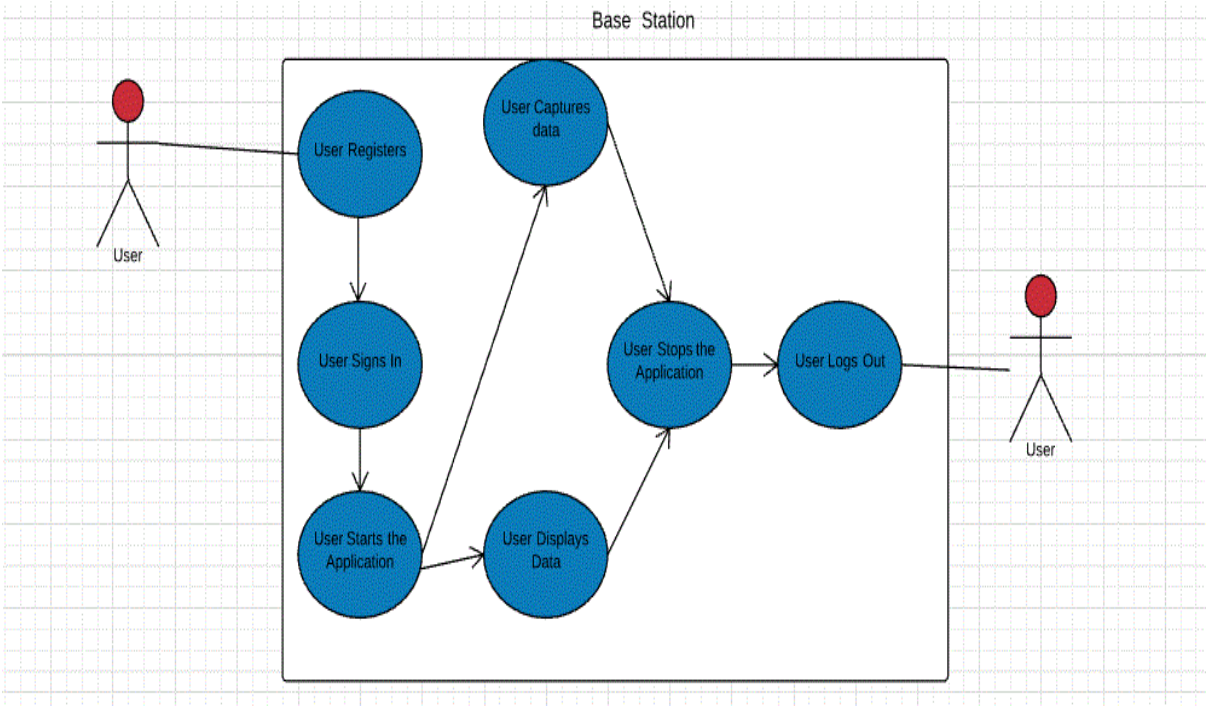


Figure 3.2. Use Case view of the LCIMU System

## 3.3 Use Case Descriptions

**User Registers**

This use case represents a user that registers an account profile to access the sensor data.

**User Signs In**

The use case represents a user that has successfully registered his account and signs in to the application.

**User Starts application**

This use case represents the user successfully logging in to the application and starting the data capture process.

**User Captures Data**

This use case represents a user, which can either be the LCIMU strapped user or another stand in user, capturing data from the LCIMU system. The data is captured as soon as the program is running on the software platform.

**User Displays Data**

This use case represents a user visualizing the data flow from the LCIMU device. It is important to point out here that this use case is automatically initiated by the Capture Data use case.

**User Stops the Application**

This use case represents a user terminating the data capture process on the application. When the User stops the application he/she will remain in his/her profile page and might decide to continue or exit the application.

**User Logs Out**

This use case represents a user terminating and logging out of the LCIMU application.

## 3.4 Logical View

The logical view of the LCIMU system gives a description of the important classes, subsystems and their organization into layers. It gives a view of a realistic use case materialization of the holistic system. The logical view of the LCIMU system is comprised of two main packages. The User Interface and the Motion Capture System package. While the User Interface contains a class for the real-time data display view, the Motion Capture System package contains control classes for communication between the LCIMU system and the base station (laptop).

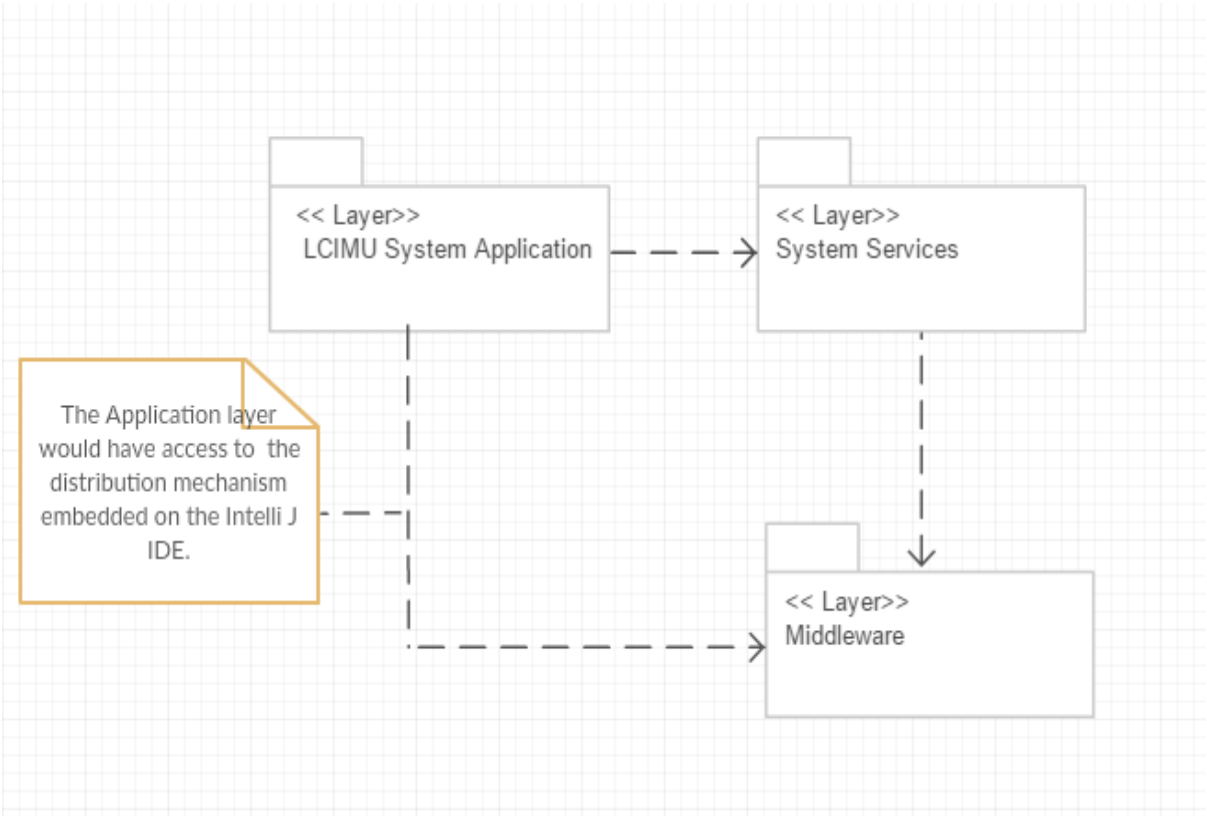


Figure 3.2. An overview of the LCIMU subsystem layers and packages

**System Application**

This application layer contains the classes that represents the application screens that the user views. This layer separates the user from the mid-tier layer of the system.

**System Services**

The System services layer contains all the individual control classes that represents the important use cases that dictates the behaviour of the system. This layer represents the border of separation between the user and the mid-tier.

**Middleware**

This layer represents the mid-tier section of the overall system. It supports access to the data base management and in this case, the storage system (Intelli J).

## 3.5 Process View

The process view gives a description of the process of execution of the tasks associated with the LCIMU system. This process model illustrates the data capture, display and storage process.

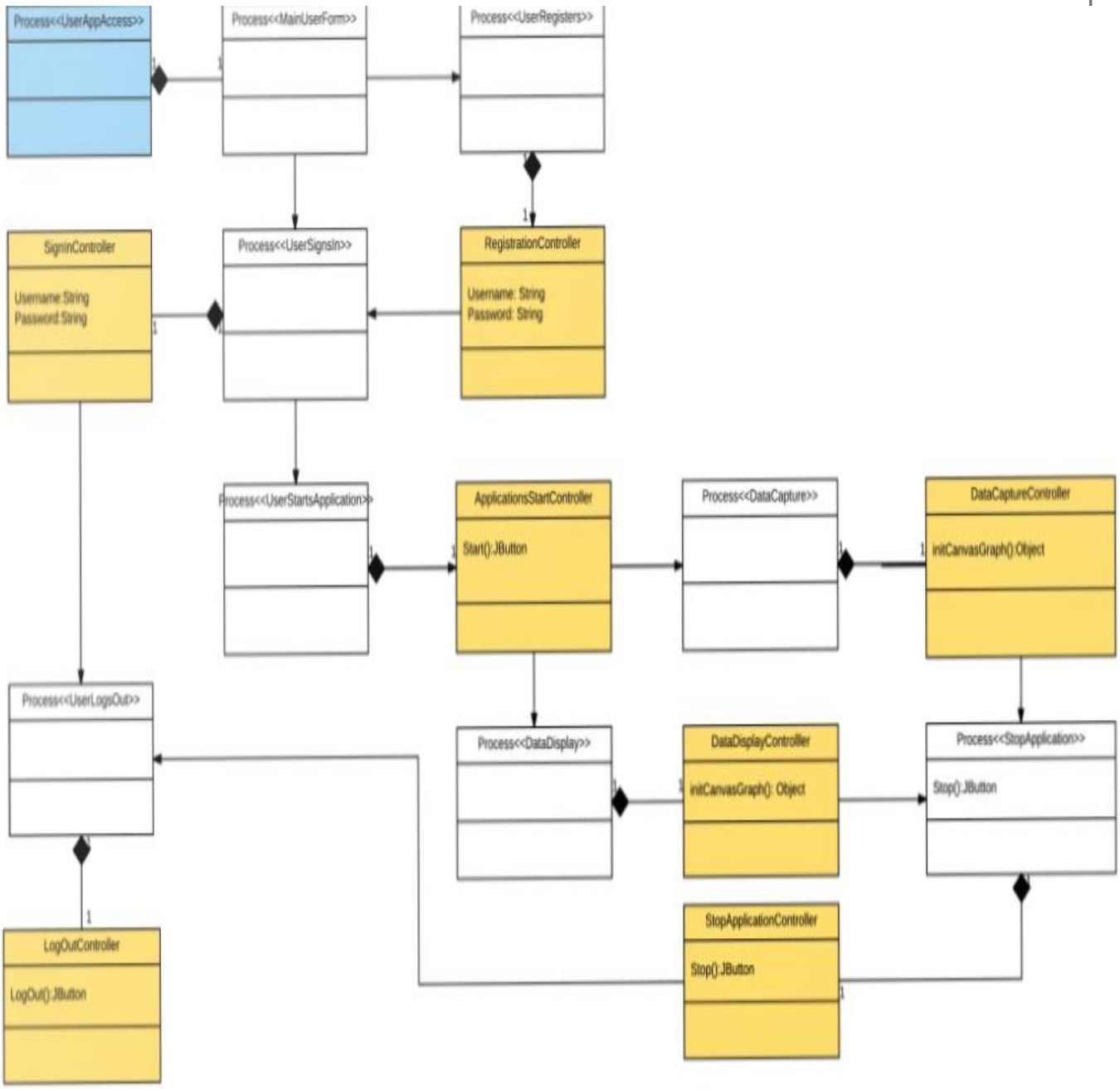


Figure 3.3. LCIMU System Process View

**UserAppAccess**

This process manages the user's access to the system's functionality. There is one instance of this process for each user that accesses the application.

**MainUserForm**

Controls the interface of the User application as well as the forms associated.

**UserRegisters**

There is one instance of this for every user that registers to use the application

**RegistrationController**

This process supports the use case that allows a user to register with a unique name and password to use the application.

**UserSignsIn**

This process allows a registered user to sign in

**SigninController**

This process supports the use case that allows a registered user to sign in with a username and a password

**StartApplication**

This process allows a user who has successfully signed in to start the application

**StartApplicationController**

This process supports the use case that allows a signed in user to start the data capture application

**DataCapture**

This process allows a user who has started the application to capture and store the data

**DataCaptureController**

This process manages all individual instances of the user initiating the data capture process.

**DataDisplayProcess**

This process allows the user to the display the captured data.

**DisplayDataController**

This process controls access to the data captured and displays the data in real time.

**StopApplicationProcess**

This process allows the user to stop the application stores.

**StopDataController**

This process supports the use case that allows the user to stop the application.

**UserLogsOut**

The process allows the user to log out of the application.

**LogOutController**

This process supports the use cases that allows the user to log out of the data capture application.

## 4.0 Hardware Architecture

The hardware architecture view of the system gives a detailed description of the LCIMU design and build from the ground up. The LCIMU system is comprised of different component parts chosen because of their characteristics and functionality. The components of the LCIMU are an accelerometer/gyroscopic device, a microcontroller, a logic level converter, a proto board, an RJ45 8-pin connector, connecting wires and resistors. The exact models of the components chosen will be highlighted as well as a brief description of their key characteristics. Other important tools used are; a soldering iron, solder lead, hand drills and heating insulators.

## 4.1 Overview of the accelerometer/gyroscope sensor

The accelerometer/gyroscope sensor used in this design was the MPU-6050 3-axis accelerometer/ gyroscope module. The salient feature of the MPU-6050 is that it's an affordable generic module. It is quite sensitive and has a range of ± 2, ± 4, ± 8, ± 16 g and 250, 500, 1000, 2000 °/s. It has a 16-bit analogue to digital interface. It's 2 x 1.6 x 0.1mm in size. It measures three axes; x, y and z axis. It has a power supply range between 3v – 5v. Figure 4.1 is an image and schematic of the MPU 6050

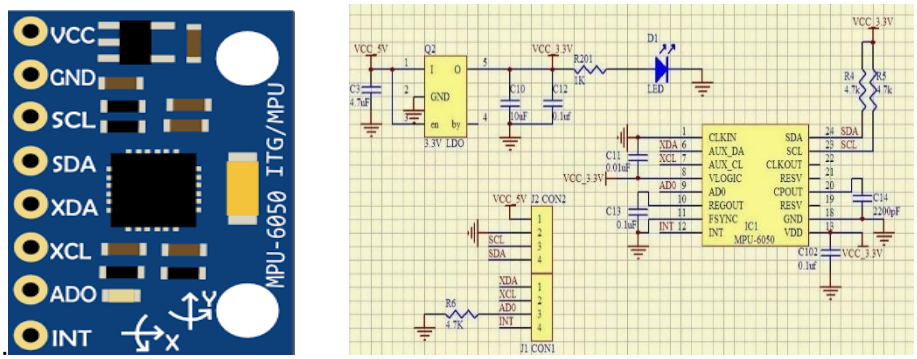


Figure 4.1. an MPU-6050 and its schematic (Invensense 2015)

## 4.2 The microcontroller boards

The microcontrollers used in the LCIMU design were the Atmega328 and AT91SAM3X8E microcontrollers of the Arduino Nano and Arduino Due boards respectively. Its dimensions (45mm x 18mm), which is relatively small and compact. Its communication protocol is the UART (used explicitly for this design). It has an operating voltage of 5V. It is cost effective as generic brands of it goes for as much as under $25.It has 14 digital and 8 analogue pins. It has external memory support, a flash memory of 32kb and a clock speed 16MHz. All the features of the Arduino Nano highlighted above were factored into its choice as the key microcontroller board for running the software program on the MPU6050. The Arduino Nano was used as a slave device. Figure 4.2 is an image of the Arduino Nano and its schematic diagram.

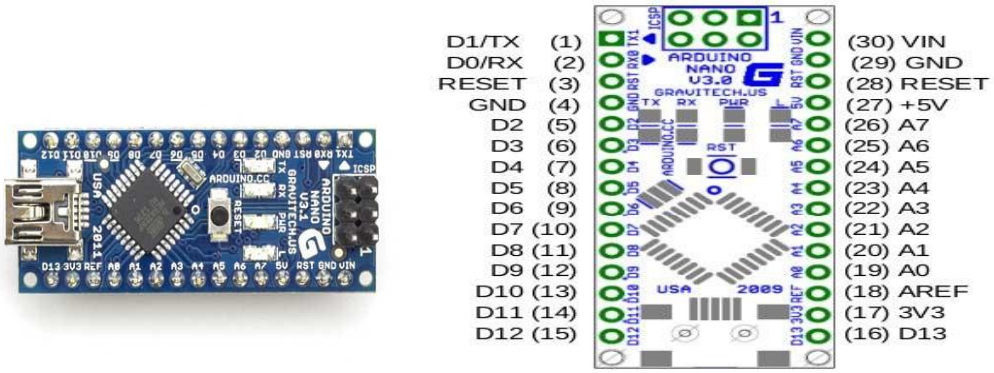


Figure 4.2 an Arduino Nano (Goutorbe 2015, Arduino 2016)

The salient features of the Arduino Due considered are:

- its dimensions (101.52mm x 53mm), which is compact and portable.
- Its communication protocol is the UART (used explicitly for this design).
- It has an operating voltage of 3.3V.
- It is cost effective as generic brands of it goes for as much as under $40.
- It has 54 digital and 12 analogue pins.
- It has external memory support, a flash memory of 512kb and an impressive clock speed of 84MHz.

The processing speed of the Arduino Due is one of the reasons why it was chosen for this design implementation. Its clock speed of 84MHz ensures that its sketches are processed at warp speed. The Arduino Due is used as the Master device.

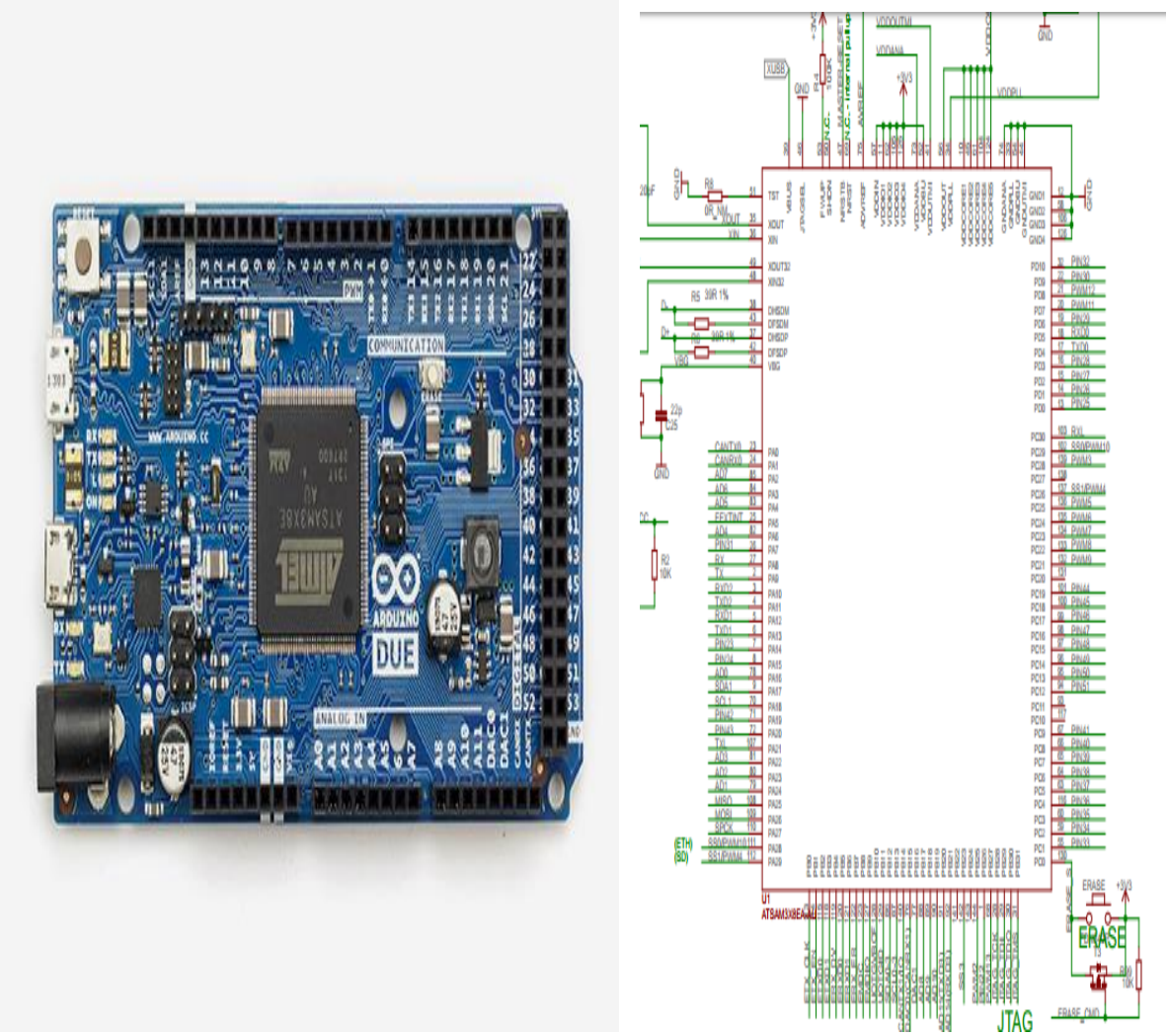


Figure 4.3. an Arduino Due and its schematic (Arduino 2016)

## 4.3 Other component devices

The other component devices are the logic level converters, the RJ45 8-pin connector, resistors and the ProtoBoard. These features of the components will be highlighted in this section

Logic Level Converter

A logic level converter is used to switch the logic levels of two different devices. It allows two devices od different operational voltages to be connected to one another. The logic level converters by Electronics (2014), has bidirectional voltage sources, it acts as a simultaneous step down ( converts a 5V signal to a 3.3V signal) and a step up voltage ( 3.3V signal to a 5V signal) device. It is relatively affordable and cost effective at under $20. It has a small dimension of 16.05 x 13.33mm. An image of a logic level converter is shown below.

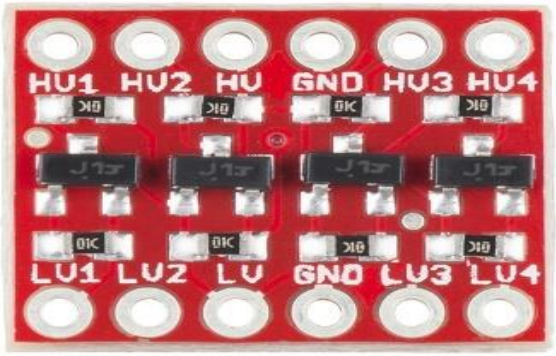


Figure 4.3. a logic level converter (Electronics 2014)

RJ45 8-Pin Connector

The RJ45 8-Pin connector is a generic product used to transmit data and power by Ethernet cables plugged into it over distances. The Ethernet cables includes the commonly used Cat5, Cat5e, and Cat6 cables. It is inexpensive, as it goes for under a $1 for a unit and its easy to use. Figure 4.4 is an image of an RJ45 connector and schematic

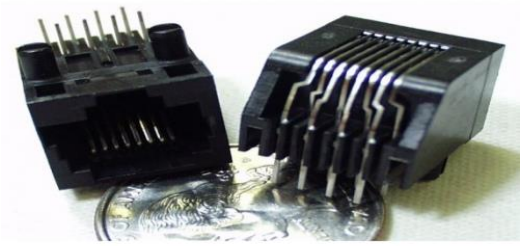

Figure 4.4 an Rj45 connector (Electronics 2016)

ProtoBoard

The ProtoBoard (DFROBOT 2016) is a pre-formed (perf) circuit board. It has a 0.1” spacing through the individual holes and four mounting holes. The mounting holes are 3mm in diameter. It has a diameter of 58mm x 78mm and it costs under $3 per unit. Figure 4.5 is an image of a ProtoBoard.

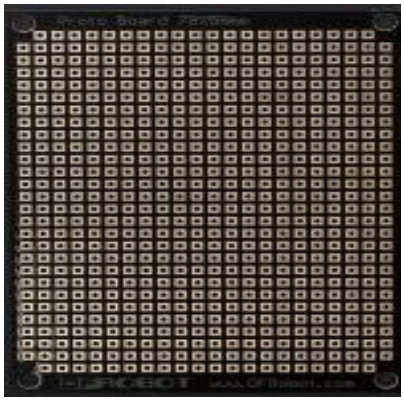

Figure 4.5. a ProtoBoard (DFROBOT 2016)

Ethernet cables, wiring and resistors

The Cat5e module of the commercial available Ethernet cable was chosen for data transmission between IMU devices and the base station. The Ethernet cables were of varying lengths ranging from 1 – 3m in length. Also, standard cheap commercial solid wires of 0.321mm in diameter were used in connecting the components on the ProtoBoard. The resistors were 10k ohm resistors used as a pull up for the A4 and A5 pins on the Arduino Nano. Figure 4.6 is an image of solid wires and a resistor.

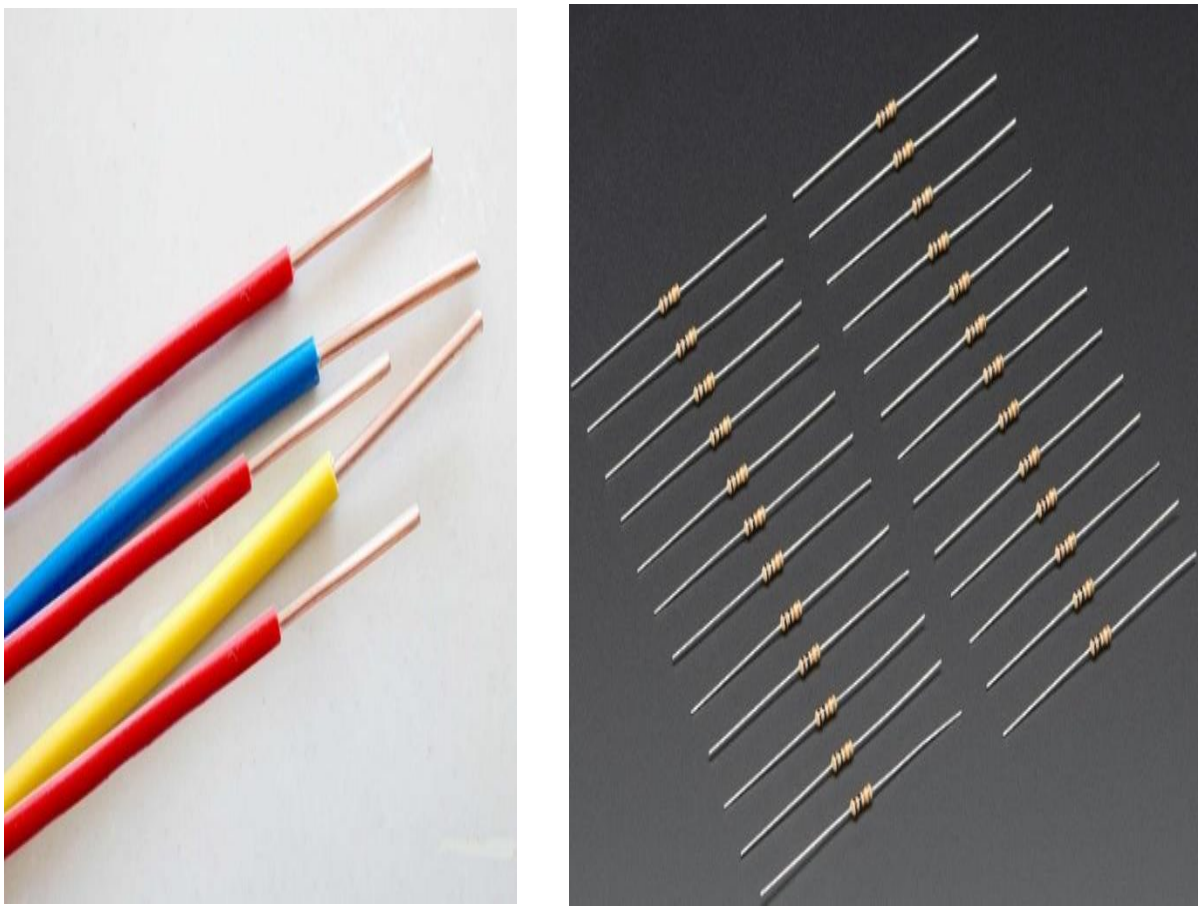

Figure 4.6. Solid wires and resistors (Adafruit 2014, SparkFun 2016)

## 4.3 Communication Protocol

The communication protocols that will be adopted for the transfer of the data between the individual components of the overall system will be reviewed in this section.

A communication protocol defines the rules necessary for data exchange between one or more interlinked circuits (Sparkfun 2014). Communication protocols are classified into 2 major categories; parallel and serial. Although the parallel communication protocol is fast and easy to implement, certain drawbacks such as the need for multiple input/output lines and its cost prevents it from being utilised in small scale projects. Serial communication protocol on the other hand, is cost effective and allows for the transfer of single bits of data across one and a maximum of four lines from data buses. The data being transferred via this protocol is in binary digits and it is channeled across data buses. Figure 4.7 is a schematic of a serial communication.

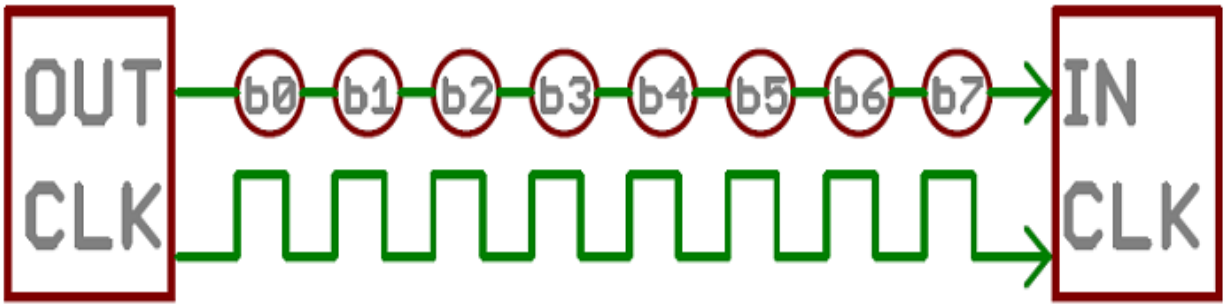


Figure 4.7. Serial communication across 2 lines (Sparkfun 2014)

Some examples of serial communication protocols are the UART, SPI and the I²C.

### 4.3.1 UART communication protocol

The UART is responsible for interfacing both parallel and serial communication data across their data buses (Rouse 2011, Sparkfun 2014). The UART is responsible for sending and receiving serial data. The UART receives data by sampling at a given baud rate from its data receiving end (RX), creating a data packet (comprising of the data bits, synchronized and parity bits) and sending the data packets from its transmitting end (TX) across transmission lines. Some UART's can store the transmitted data in a buffer, from where it can be released unto a microcontroller in a first-in-first-out (FIFO) basis. Microcontrollers such as the Arduino Uno and Yun have an inbuilt UART but others do not have.

Figure 4.8 is a schematic of a UART, interfacing a parallel and serial communication data buses. The asynchronous serial interface encounters some drawbacks when trying to interface data between two systems that have a slightly different clock rates as it does not have control over when data is sent across the systems as they are not synchronized.

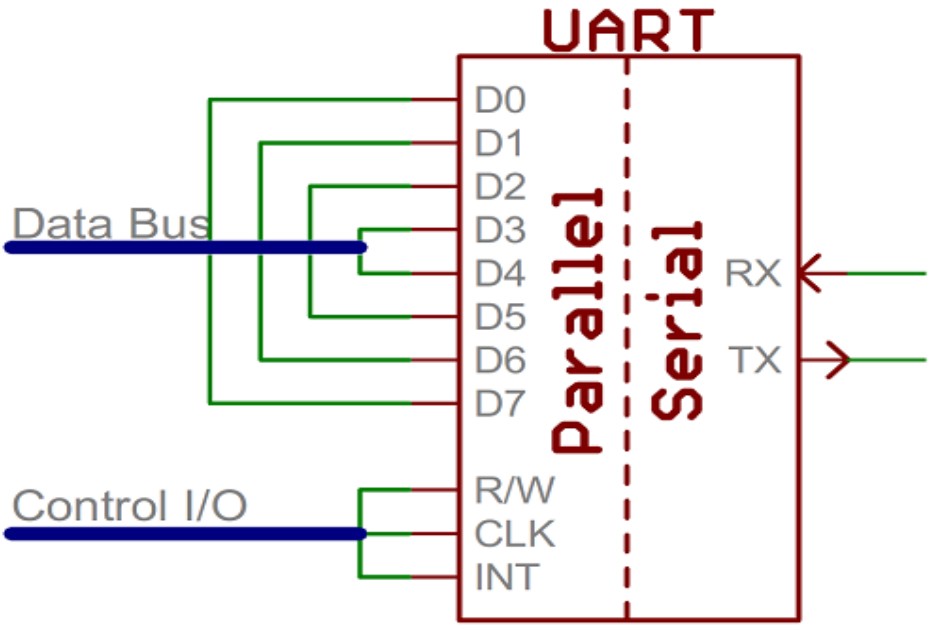


Figure 4.8. A schematic of a UART (Sparkfun 2014)

### 4.3.2 I²C communication protocol

The inter-integrated circuit protocol (I²C) is a protocol that allows multiple slave digital slave integrated circuits to communicate with one or more master circuits. It only makes use of 2 signal lines but any number of slaves and masters can be connected unto these two signal lines (Paradigm 2014). The inter-integrated circuit protocol is serial communication bus protocol intended for short distances within a single device. It requires only two signal lines to exchange data. An I²C device is recognized by a 7-bit address. The I²C interface has a 7-bit address space with each bit having 16 reserved addresses, providing 112 available nodes for communication on the same bus interface. The maximum number of nodes in an I²C device bus (Paradigm 2014) is dependent on the address space and the total capacitance of the bus interface (400pf). The speed of the transfer of data can be chosen from three ranges; 100kps (for standard mode), 400kps (for low speed mode) and 3.4Mbps (for high speed mode).

The two signal lines used in the transfer of data between the devices connected to the I²C bus interface is known as Serial Data line (SDA) and Serial Clock line (SCL). The SDA contains the signal of the data been transferred while the SCL contains the clock signal. The operating voltages are either +5V or +3.3V. Like in the SPI protocol, the device that initiates the data transfer and generates the clock signal is known as the master while the device been addressed is known as the slave. Figure 4.9 is a schematic of the block structure of the connection between two masters and two slave devices.

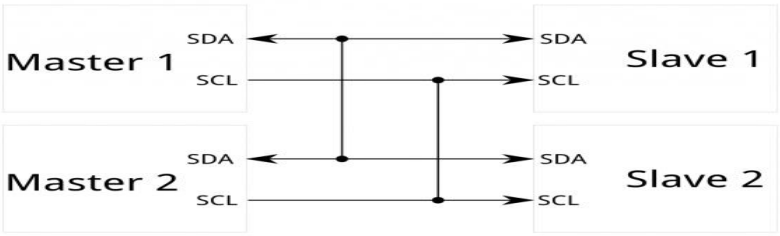


Figure 4.9. Master and Slave Connections (Sparkfun 2014)

According to Sparkfun (2014), The I²C bus can drive a signal line low but cannot drive it high, which means that they are open drain. This essentially eliminates the possibility of signal obstructions between conflicting signal lines of devices with opposing line states (high and low). Each signal line restores the state of the line to high using its individual pull up resistor, when not under the influence of a low drive. Although, resistor selection varies with devices on an I²C bus, it is advisable to begin with a 4.7k ohms and adjust down when necessary. Figure 4.10 is a schematic of a master and slave line connection with the pull up resistor on each line.

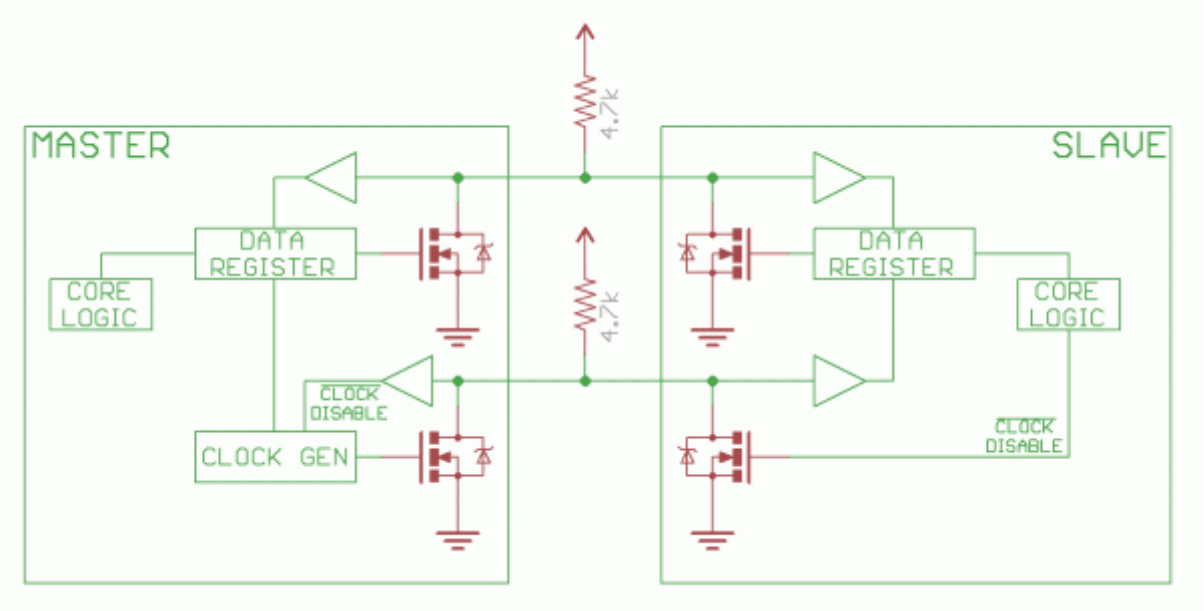


Figure 4.10. A master/slave connection with pull-up resistors (Sparkfun, 2014a)

The I²C protocol offers the possibility of connecting devices with different input and output voltages. This is done by connecting a pull-up resistor to the lower of the two different voltages. The data to be exchanged between the devices is broken down into two different frames: an address frame and a data frame(s). The address frame is the frame where the master identifies the slave to which the data is been sent to while the data frame is the frame through which the data passes from the master to the slave and vice-versa. The data is placed on the SDA line after the SCL line goes low and is sampled when the SCL line is driven high. The master device drives the SDA line low and keeps the SCL line high, in order to initiate the address frame. This informs all connected slave devices about an incoming transmission from the master device. When more than one master device tries to take control of an I²C bus interface at a time, whichever master device drives the SDA low first takes control of the bus at that point in time (Sparkfun 2014). The master device first sends a start bit (which is the most significant bit (MSB)) and a unique binary address (7-bit address) of the slave device it wants to access, this is then followed by a read or write bit (1-bit data or the least significant bit (LSB)) informing the slave device that it wants to either receive or send data to it. After this happens, depending on the number of devices connected to the master, each slave device will then compare the binary address with its own to check if it's a match, with the resulting matching slave device sending an acknowledgement signal to the master device. The acknowledgement signal (1-bit data known as ACK or NACK (Not Acknowledge)) completes the address frame. When the data moves from the address frame to the data frame, the slave device then takes control of the SDA line and pulls it down indicating that it has received the data. Once all the data frames have been sent (as we can have more than one data frames), the master device generates a stop condition by transitioning the SDA line from low to high.

## 4.4 The design of the slave IMU device in the LCIMU

The key aspects of the design of the slave IMU are the interaction between the component devices which are highlighted below:

- Arduino Nano and the MPU6050
- Arduino Nano and the Logic Level Converter
- Arduino Nano and the RJ45 Connector
- RJ45 connector and the Logic Level Converter
- All Components and the Protoboard.

These various interactions will be highlighted in sub sections below.

### 4.4.1 Interaction 1: Arduino Nano and The MPU6050

The Arduino Nano interacts and communicates with the MPU6050 by the $I^2C$ protocol. This is made possible by the MPU6050 having dedicated SDA and SCL pins installed. The SDA pins allow for Serial data to be transferred while the SCL supports the transfer of the corresponding clock signals. The Arduino Nano was designed with dedicated pins that support both $I^2C$ and serial communication. In this case, it's dedicated $I^2C$ pins were A4 and A5. The A4 pin receives the data from the MPU6050 SDA line while the A5 pin receives the clock signals from the SCL pin. Figure 4.11 is a schematic of the interaction between this two components on the slave device. The interrupt pin is D3 on the Arduino Nano, this ensures that whenever sensor data is being sent through from the MPU6050, any action or current state of the Arduino Nano would be suspended temporarily until the incoming data is received. In the schematic, two 10kohms pull up resistors were used to control the logic levels of the MPU6050 when the sensor is turned off and on. This was necessary to ensure that the Arduino Nano communicates smoothly with the MPU6050.

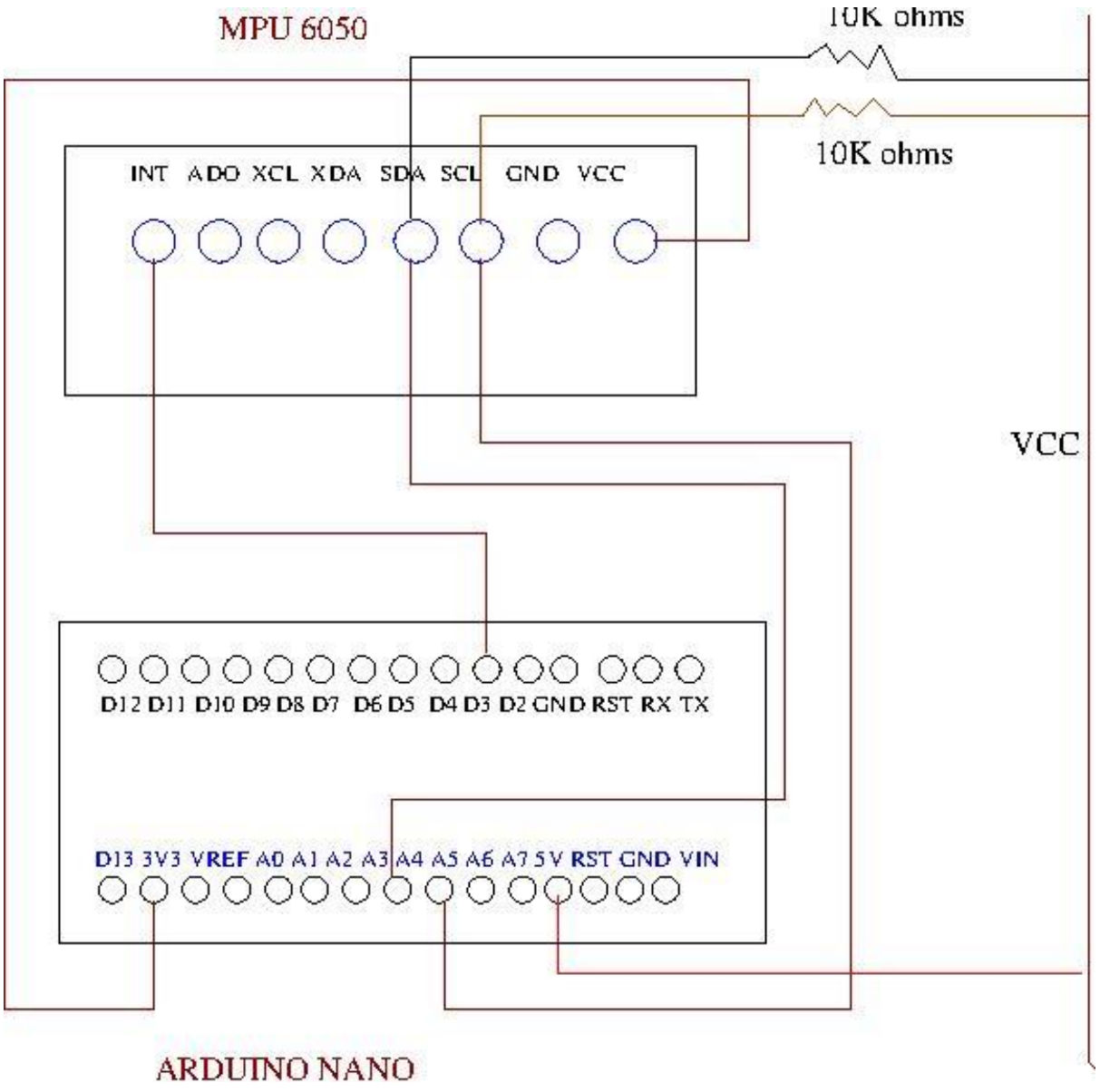


Figure 4.11. Interaction between the Arduino Nano /MPU6050

### 4.4.2 Interaction 2: Arduino Nano and Logic Level Converter

The Logic Level converter (LLC) is used to allow the Arduino Nano communicate with the Arduino Due on the master device as they both are powered by different voltage sources (The Arduino Nano by a 5V source while the Arduino Due by a 3.3V source). The LLC switches the logic states of both devices to enable data transmission to be possible by pulling the logic levels of the Arduino Nano from high to low. This interaction does not require a communication protocol. Figure 4.12 is a schematic of the interaction between the Arduino Nano and the LLC. The RX, TX pins are connected to both the HV3 and HV4 pins respectively of the LLC. While the GND pin and the 5V voltage from the Nano is connected to the GND and HV pins of the LLC.

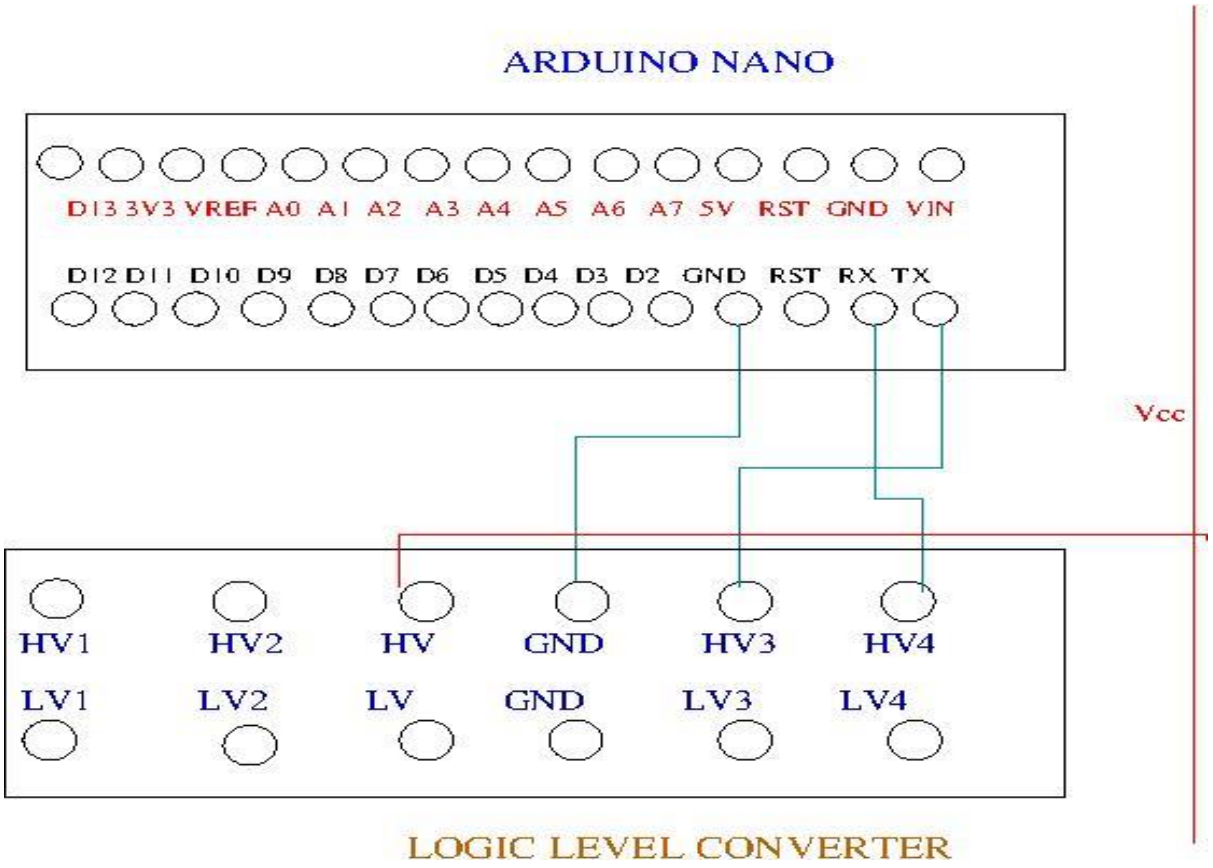


Figure 4.12. Interaction between Arduino Nano and the LLC

### 4.4.3 Interaction 3: Arduino Nano and the RJ45 Connector

Data transfer between the master and slave devices on the LCIMU system was carried out by transmission cables as a means of mitigating the effects of data lagging while transmitting data wirelessly through a medium such as a Bluetooth. The RJ45 Connectors were used to integrate the connecting cables with the slave device. It was mounted on the protoboard for stability of access. The RJ45 connector has 8 pins as indicated in Electronics (2016). Pins 4 and 8 are the GND and Vin lines respectively and these were connected to the GND and Vin pins of the Arduino Nano.

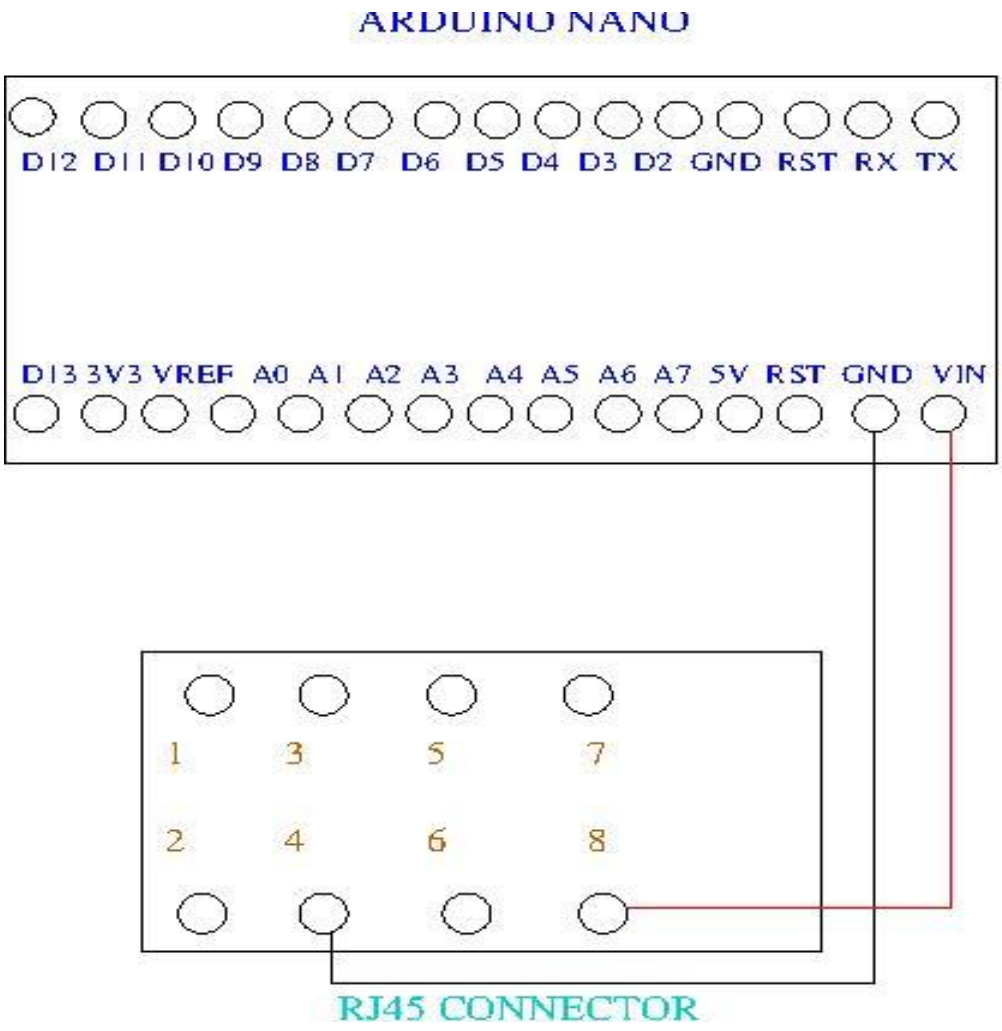


Figure 4.13. Interaction between the Arduino Nano/RJ45 Connector

### 4.4.4 Interaction 4: RJ45 Connector and the Logic Level Converter

The LLC (Electronics 2014) transfers the converted data lines and power lines from the LV4, LV3, GND and LV pins through to the RJ45 connector. Pin 7 on the RJ45 connector corresponds with LV4 (carrying data on the RX line from the Arduino Nano), Pin 6 corresponds with LV3 (carrying data on the TX line from the Arduino Nano). Pin 5 to GND and lastly Pin 2 to LV. Figure 4.14 is a schematic of the interaction described above.

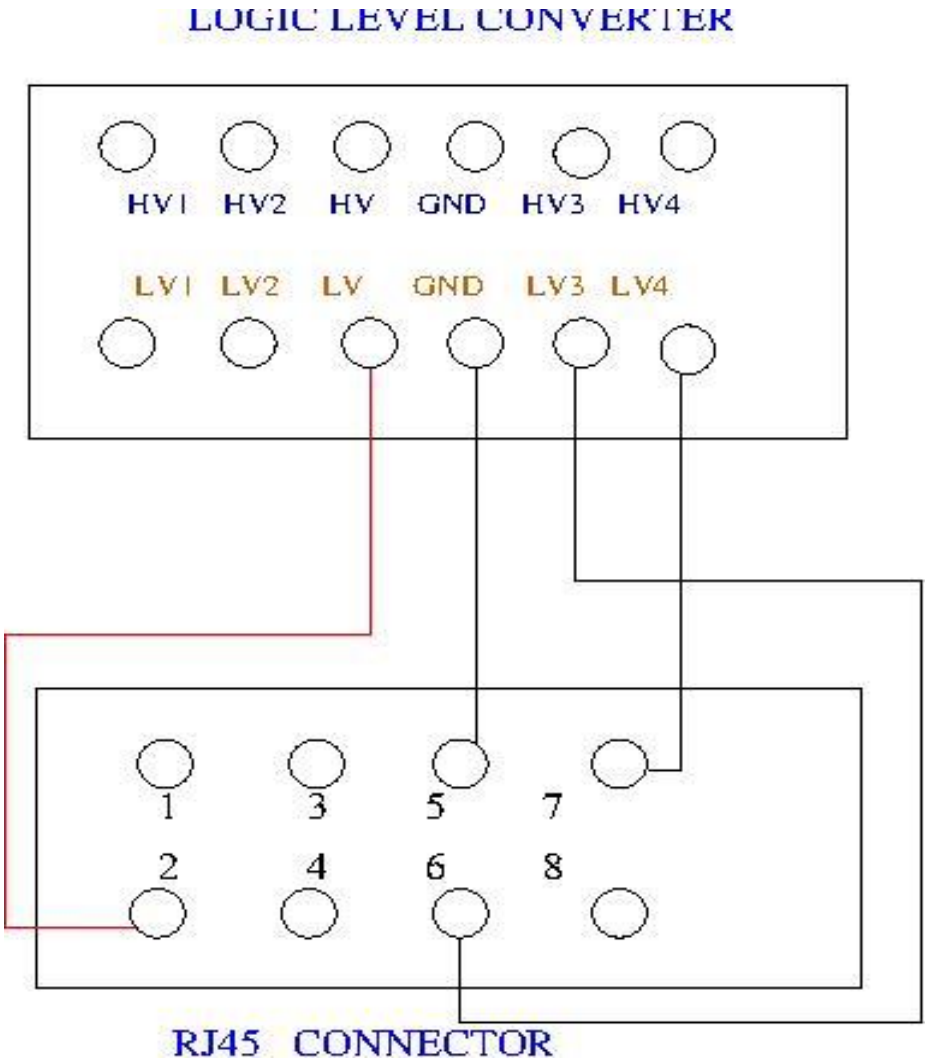


Figure 4.14. Interaction between the LLC/RJ45 Connector

## 4..4.5 Interaction 5: All Components and the Protoboard

The protoboard (DFROBOT 2016) has a dedicated voltage line (Vcc) that run parallel one side of the board. This allows for multiple components to be powered from a source connected to the line. In other words, if a 5V source is connected to the line, it could power multiple 5V components without having to connect multiple components to one pin on the line carrying the source voltage as seen in some generic boards available in the technology market today. All the component parts highlighted in the previous sections are integrated to form the LCIMU slave design in Figure 4.15. The positioning of all components on the protoboard was key to reducing challenges of crossing lines together and the interference from closely soldered pins. Figure 4.16 is an actual image of all components mounted on the protoboard. It highlights the advantage of closely placing the components together while maintain spacing integrity to ensure that the wires do not interfere with one another and that they rest properly on the board.

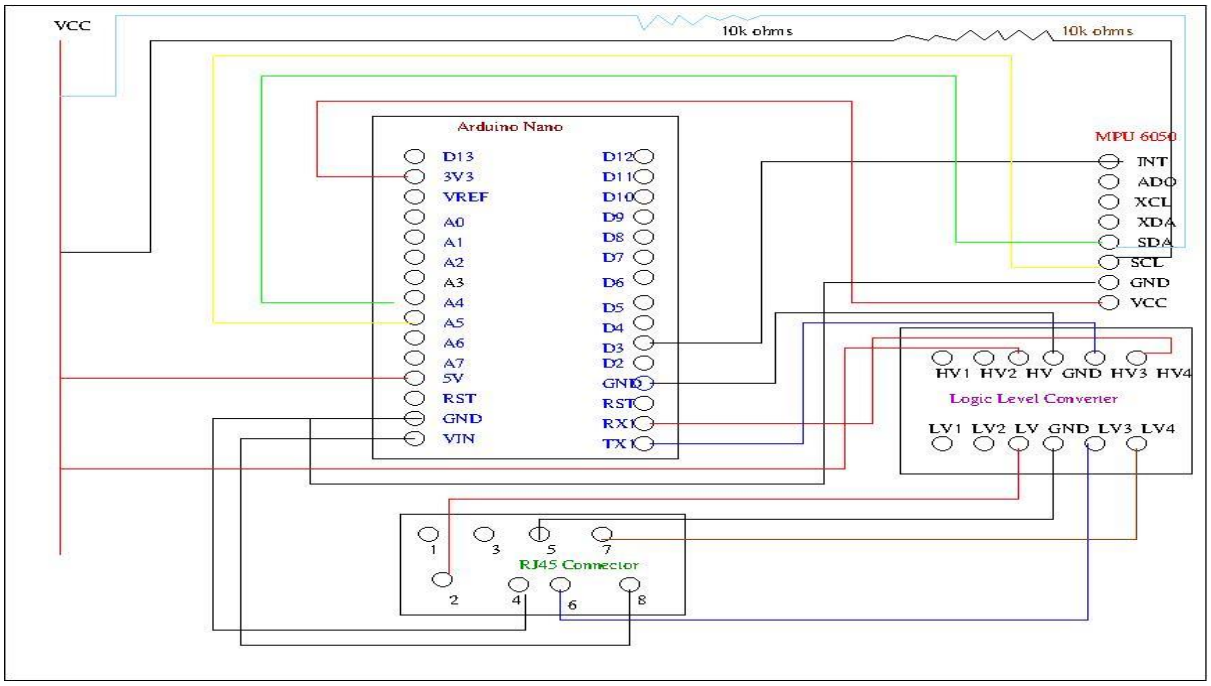


Figure 4.15. The schematic of the slave device design in the LCIMU

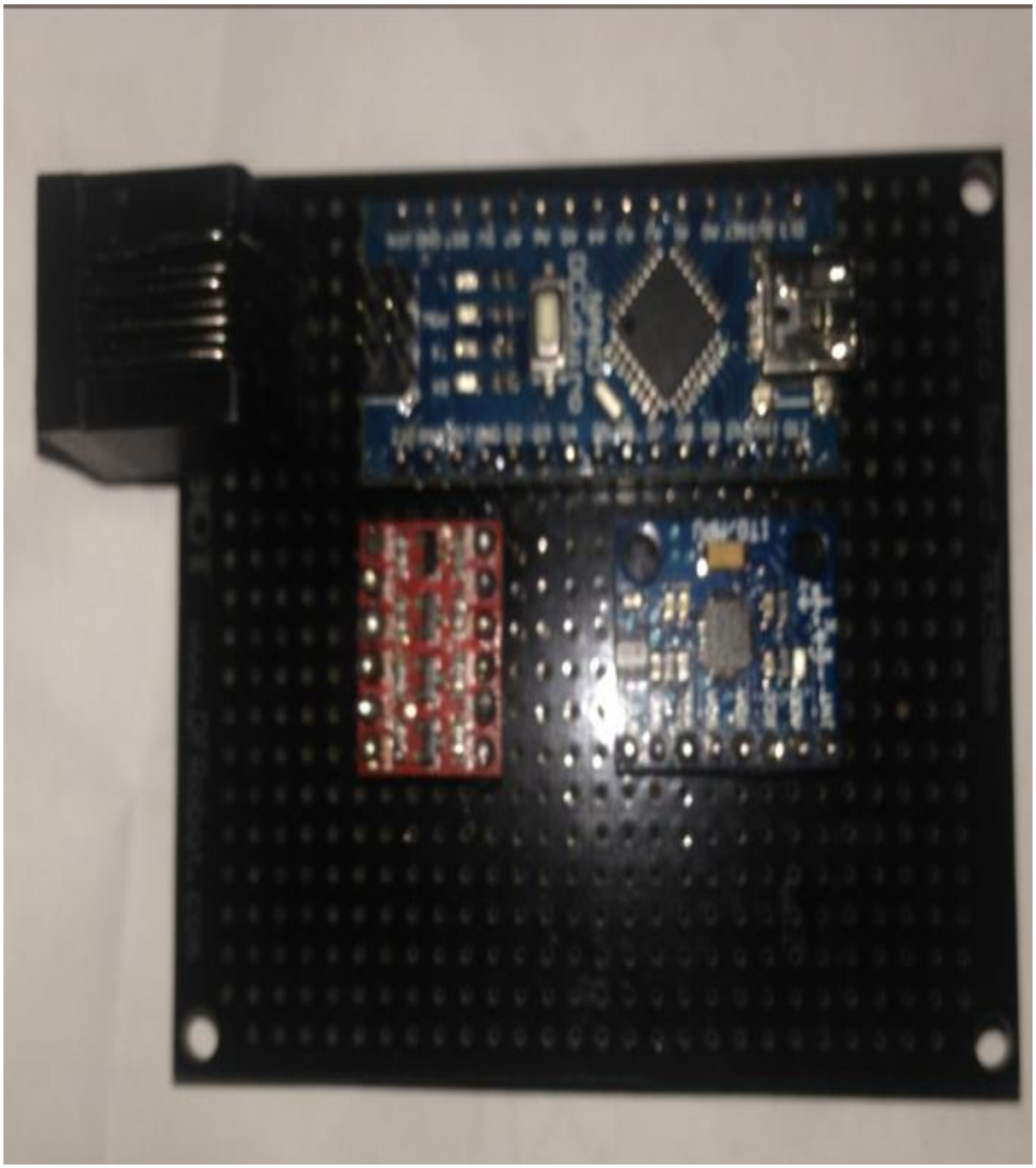

Figure 4.16. An image of the slave device in the LCIMU

## 4.5 The design of the master device in the LCIMU

The design of the master IMU is mostly quite repetitive in nature, as multiple slave devices will be connected to the one master device across three RJ45 connectors. This could best be surmised in some basic interactions.

- Arduino Due and the first RJ45 Connector.
- Arduino Due and the second RJ45 Connector.
- Arduino Due and the third RJ45 Connector.
- Master device components integrated together.

### 4.5.1 Interaction 7: The Arduino Due and the first RJ45 Connector

In this interaction, the first RJ45 connector is connected to the Arduino Due via Pins 7 and 6 which goes into TX3 and RX3 respectively on the Arduino Due. An important connection here to note would be, Pin 4 connected to the GND as well as Pin 8 to VIN, as both the master and the slave device needs to have a common ground and voltage to facilitate the correct voltage differential between the two devices. This will be repeated on the other two interactions as well. Figure 4.17 is a schematic of this interaction.

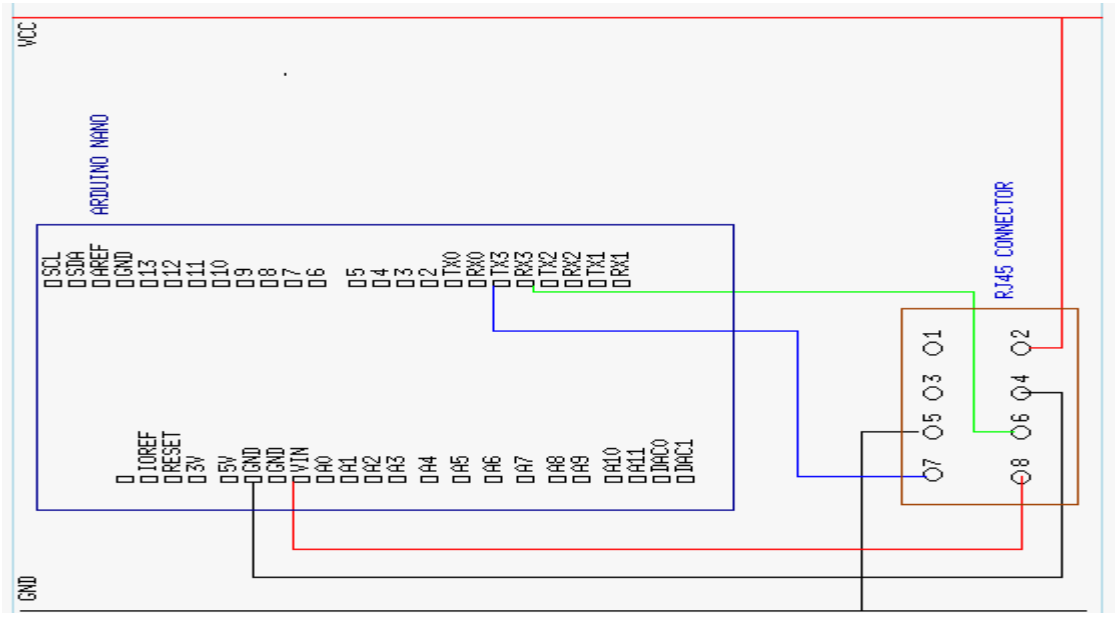


Figure 4.17. Interaction between the Arduino Due/RJ45 connector

## 4.5.2 Interaction 8: The Arduino Due and the Second RJ45 Connector

Like interaction 7 in the previous section, pin 4 is connected to GND while pin 8 to VIN. The difference here is that, the data pins (Pins 7 and 8) on the RJ45 connector goes into TX2 and RX2 of the Arduino Due. The TX2 and RX2 pins represent the transmission and receiver lines of serial port 2 on the Arduino Due ( see Arduino (2016) for more details). Figure 4.18 is a schematic of this interaction.

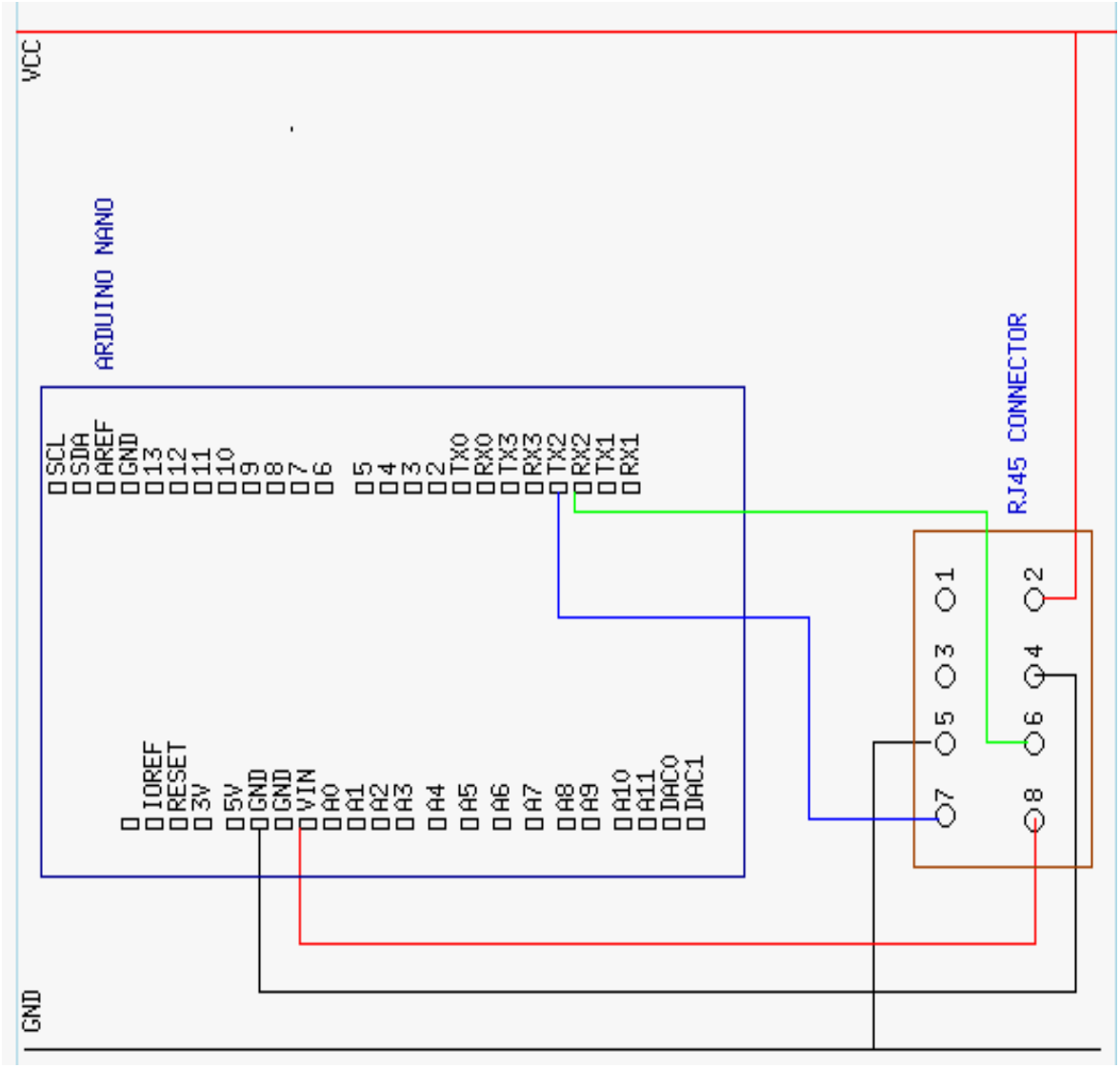


Figure 4.18. The Schematic of Interaction 8

### 4.5.3 Interaction 9: The Arduino Due and the Third RJ45 Connector

This interaction also retains the concept of its predecessors, the only difference here is that pins 7 and 8 are connected to TX1 and RX1. Figure 4.19 is a schematic of this interaction.

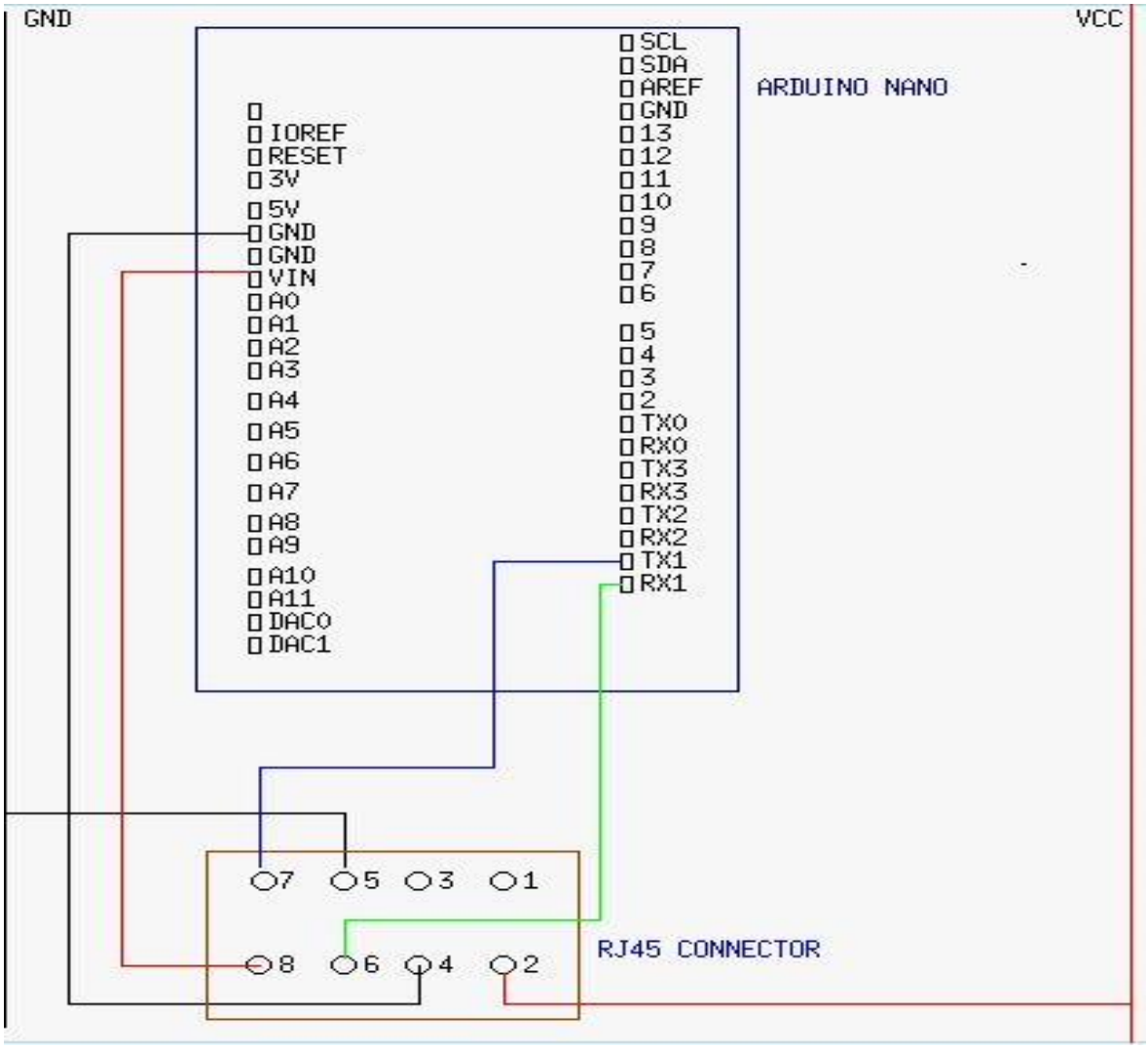


Figure 4.19. A Schematic of Interaction 9

### 4.5.4 Interaction 10: Master device Components integrated together

This interaction comprises of the integration of Interactions 7, 8 and 9 into a whole unit representing the master device as shown below.

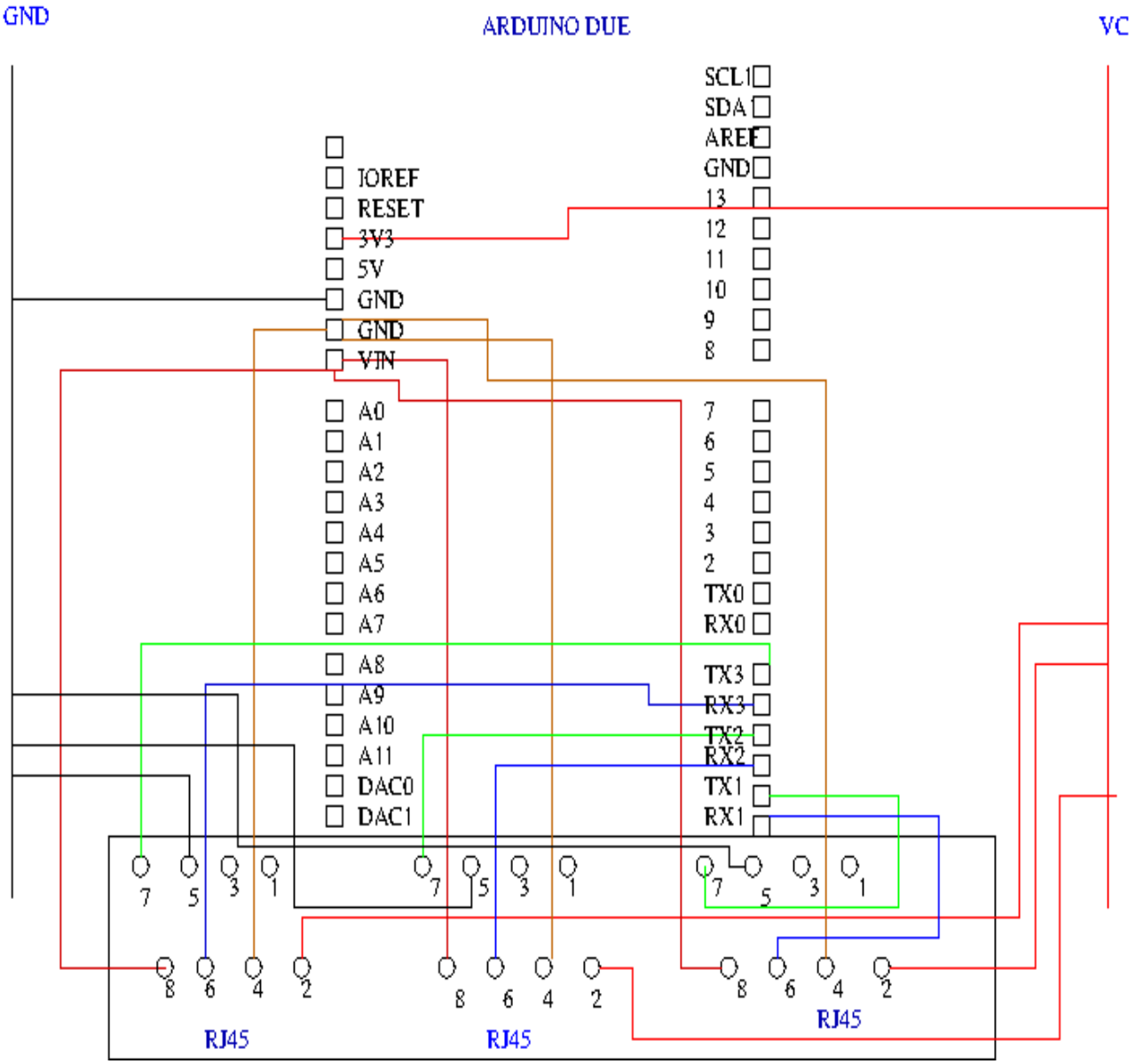


Figure 4.20. A schematic of the Integrated Master Device.

From Figure 4.20, the placement of the RJ45 connectors at the base of the IMU was necessary to allow for easy accessibility of the connecting cables. For this to be effectively implemented the three RJ45 connectors had to be soldered on to a protoboard which was then super-imposed on the Arduino Due. Physically, this would lead to an increase in the bulkiness of this design but nonetheless ensure the reusability of the device components. Figure 4.21 is an image of the superimposed RJ45 soldered protoboard on the Arduino Due.

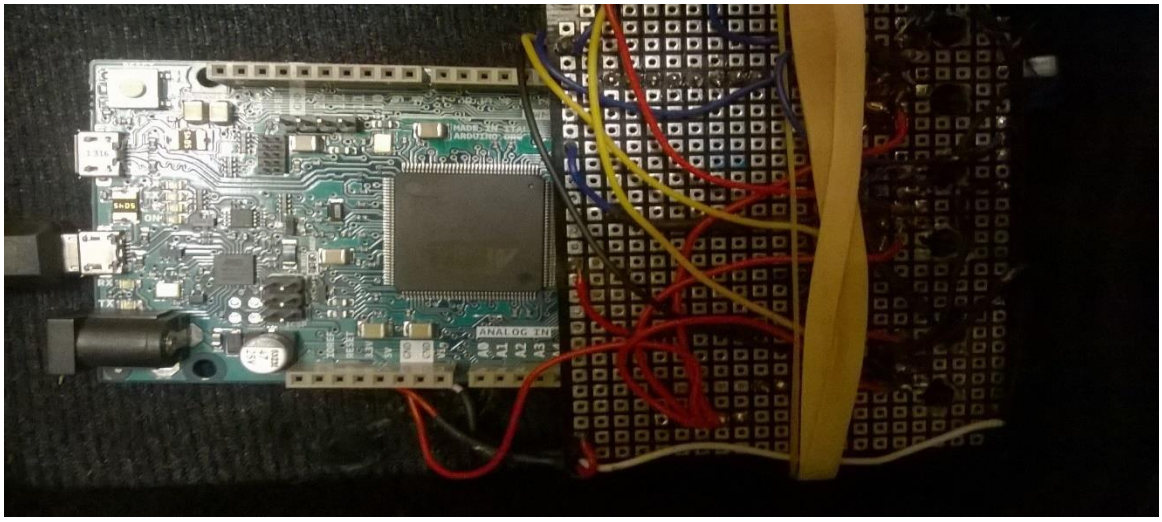

Figure 4.21. an image of the master device in the LCIMU

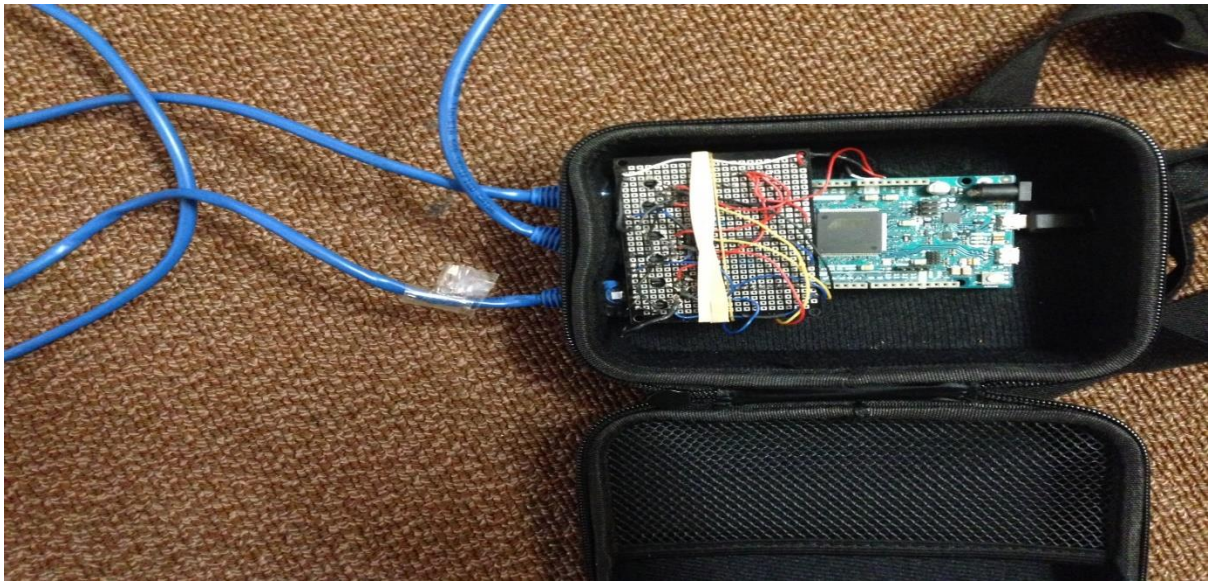

Figure 4.22. an image of the Master device in its casing.

## 4.5.5 Interaction 11: The overall schematic of the LCIMU system

This interaction gives a simplified overview of the schematic of the entire system after final integration. On the schematic, SLAVE DEVICE 11,12, and 13 represent the three slave IMU's tethered to the Master device. They are all tethered to the Master device by three cables for each unit.

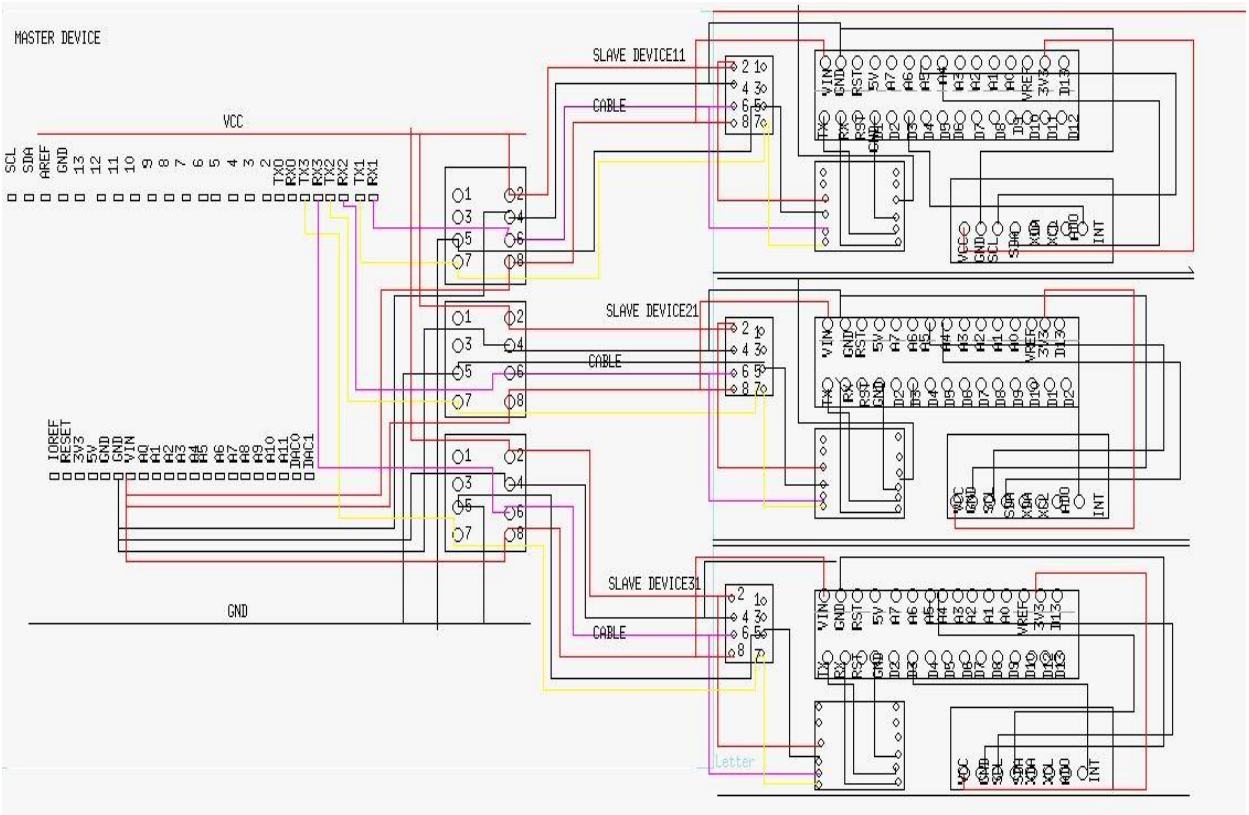


Figure 4.23. A schematic of the overall integrated LCIMU system

An actual image of the overall LCIMU in its integrated state is shown below in Figure 4.24 below.

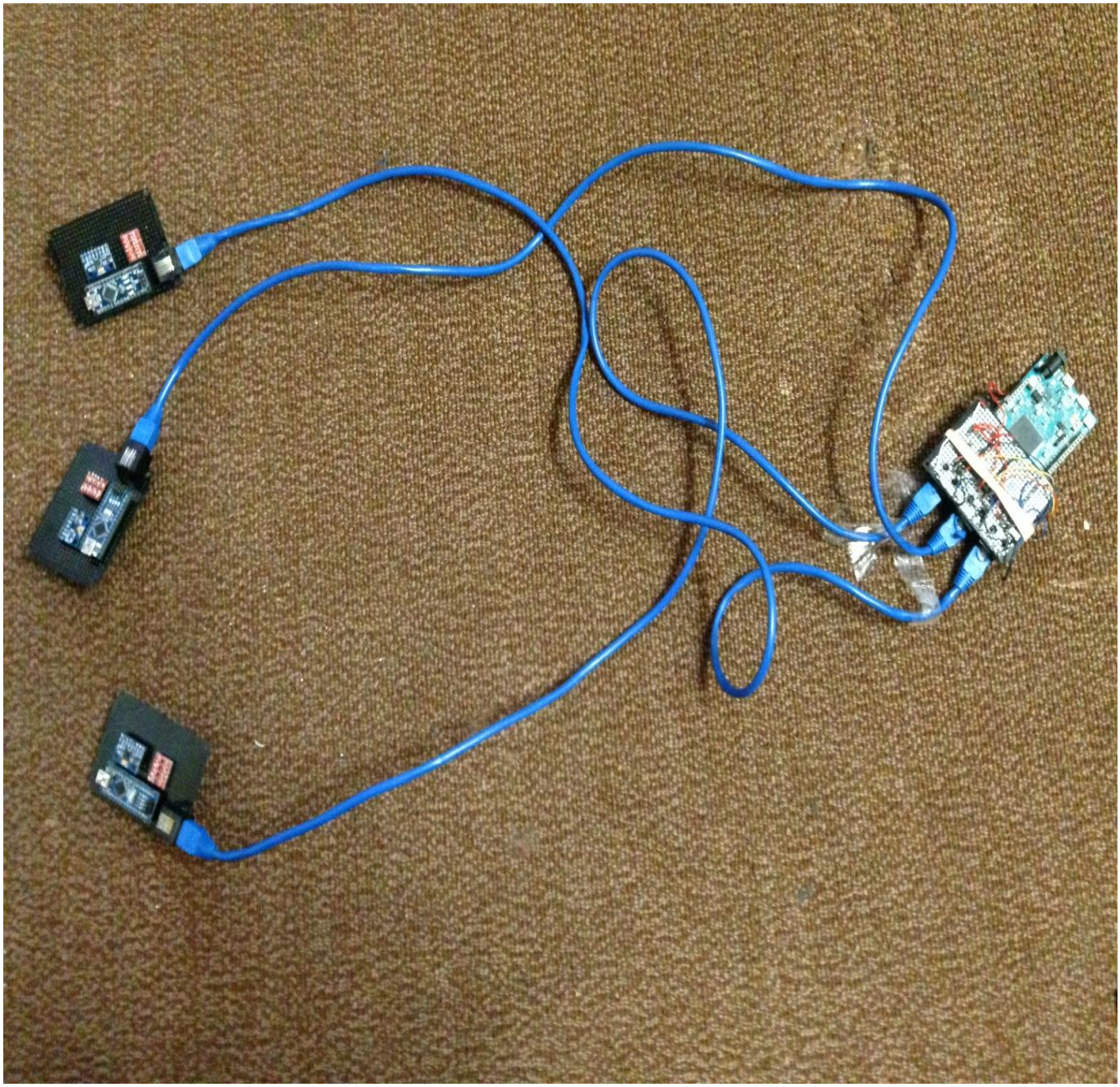

Figure 4.24. An image of the Integrated LCIMU system

## 4.6 The System Software architecture

The architecture of the software used in designing the system follows a flow of data from the individual microcontroller components down to the base station. One of the most important aspects of the software architecture is the Libraries. The Libraries are key to ensuring that the hardware device function as it should.

### 4.6.1 LCIMU Libraries

There are some important libraries instrumental to ensuring that the LCIMU system function properly. Notable among these on the Arduino side are;

- I²Cdevlib and Wire.
- MPU6050.
- Kalman.

On Intelli J, the libraries used were;

- RXTXComm.
- BeautyEye.
- Dom4j.

### 4.6.1.1 I²Cdevlib and Wire libraries

The I²Cdevlib created by Rowberg (2015) was used to facilitate the I²C communication protocol on the Arduino IDE between the Arduino Nano and the MPU6050 sensor. The Wire library on the other hand, is one of the default library packages on the Arduino IDE. These libraries both allow the ease of communication between the MPU6050 and the Arduino Nano, through dedicated pins. On the Arduino Nano, the dedicated I²C pins are A4 (SDA) and A5(SCL). These libraries contain simplified classes implemented by including the I²C header function on the sketch and calling the library by writing *#include "I2Cdev.h"* at the top of the sketch. Also, the Wire function is called by writing *#include "Wire.h"*. Although, this library can be edited, this project does not require any edit of this library.

### 4.6.1.2 MPU6050 library

The MPU6050 library was also created by Rowberg (2015). This library enabled the calibration and configuration of the MPU6050 device. The device contains numerous registers which should properly be configured for the device to function efficiently. Rowberg (2015), did an excellent job of wholly configuring the device and the only thing left to do would be to choose the range of sensitivity of the device. By default, the device is configured to the sensitivity range of ±2g for the accelerometer and ±250 degrees/sec for the gyroscope. Changing the configuration to a higher range can be done by simply editing the MPU6050.cpp file as shown in Figure 4.25 and replacing *line 65 and line 66* with any of the defined configured registers on the header file as shown in Figure 4.25b and 4.25c. To calibrate the Accelerometer range to within ±8g sensitivity, replace *line 66* with "*setFullScaleAccelRange(MPU6050_ACCEL_FS_8)*". On the LCIMU system the range of sensitivity is left at the default setting, the same was done for the gyroscope.

```
/** Power on and prepare for general usage.
 * This will activate the device and take it out of sleep mode (which must be done
 * after start-up). This function also sets both the accelerometer and the gyroscope
 * to their most sensitive settings, namely +/- 2g and +/- 250 degrees/sec, and sets
 * the clock source to use the X Gyro for reference, which is slightly better than
 * the default internal clock source.
 */
void MPU6050::initialize() {
    setClockSource(MPU6050_CLOCK_PLL_XGYRO);
    setFullScaleGyroRange(MPU6050_GYRO_FS_250);
    setFullScaleAccelRange(MPU6050_ACCEL_FS_2);
    setSleepEnabled(false); // thanks to Jack Elston for pointing this one out!
}
```

Figure 4.25a. Calibrating the MPU6050 sensitivity range

```
#define MPU6050_GYRO_FS_250         0x00
#define MPU6050_GYRO_FS_500         0x01
#define MPU6050_GYRO_FS_1000        0x02
#define MPU6050_GYRO_FS_2000        0x03
```

Figure 4.25b. Calibrating the MPU6050 sensitivity range

```
#define MPU6050_ACCEL_FS_2          0x00
#define MPU6050_ACCEL_FS_4          0x01
#define MPU6050_ACCEL_FS_8          0x02
#define MPU6050_ACCEL_FS_16         0x03
```

Figure 4.25c. Calibrating the MPU6050 sensitivity range

#### 4.6.1.3 Kalman library

The Kalman library is one of the default library packages on the new release of the Arduino IDE. It is used in filtering the noise that accompany the raw sensor signals by calculating the angle, rate and bias of the MPU6050 sensor signals. The linearity of the signal from the MPU6050 allows the implementation of the Kalman filter which relies on the usage of the last state of the sensor and a covariance probability matrix to arrive at the closest correct estimate of the sensor data. The Kalman function is called by writing *#include "Kalman.h"* to the top of the sketch.

#### 4.6.1.4 RXTXComm library

The RXTXComm library supports serial communication through the UART communication protocol. It allows serial data to be read in and out from serial ports on a computer. The library is written in java and is implemented by importing the 'gnu' class by writing the call function, *"import gnu.io. *"*.

#### 4.6.1.5 BeautyEye library

This library developed by Jiang (2011) is a generic adaptation of the java swing package. It contains similar classes contained in the java swing package but looks different in appearance. It was chosen simply for its unique look when applied to the data capture application on Intelli J.

#### 4.6.1.6 Dom4j library

On Intelli J, this library was used to parse the xml pages required to propagate the data capture application. The dom4j package enables the creation of xml documents, styling the document, writing to a document and navigating to a document. The LCIMU user account profile frame was created using this library.

## 4.6.2 The Arduino Software Architecture

The LCIMU Arduino software architecture would best be articulated using a flow chart as the data is processed across the IDE. Figure 4.26 is a flowchart of the software architecture and data flows on the Arduino IDE.

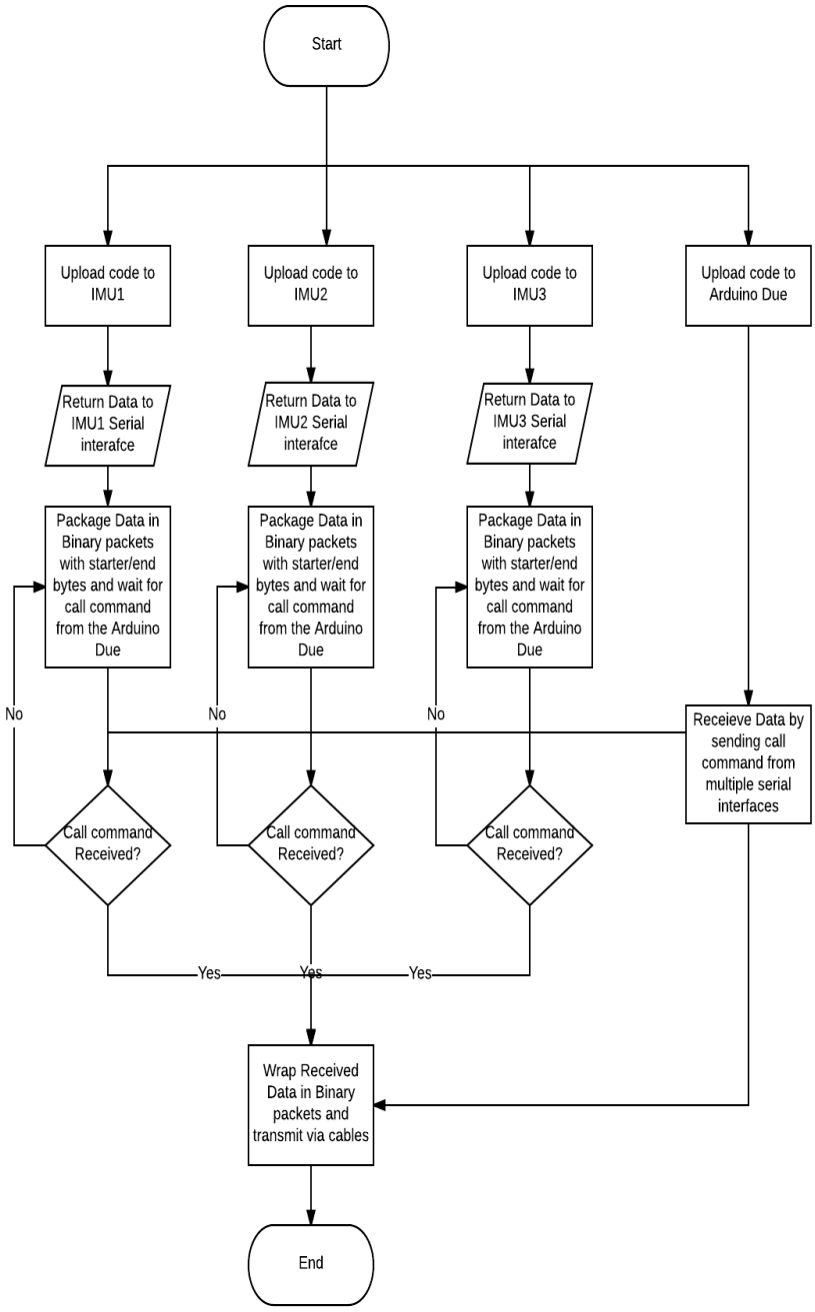


Figure 4.26. Flow Chart of the Software Architecture on Arduino IDE

**Upload code to Arduino IMU1, IMU2, IMU3 and Due**

Here, the individual IMU slave devices (Arduino Nanos) are programmed as well as that of the master device IMU (Arduino Due).

**Return Data to the Serial Interface**

The action on the first step would cause the data derived from the IMU devices to be read into the serial interface of the Arduino for temporary storage using the SRAM memory. On the Arduino Sketch uploaded to the slave IMU, after the variables have been declared as shown in figure 4.27, they are returned to the serial interface loop (for recurring data assignment) as shown in figure 4.28.

```
/* IMU Data */
double accX, accY, accZ;
double gyroX, gyroY, gyroZ;
int16_t tempRaw;
int ax, ay, az, rollx;
int gx, gy, gz, pitchx;
uint32_t tm;

// packet structure for InvenSense teapot demo
uint8_t PitchPacket[31];
uint8_t pitchBuffer[31];

double gyroXangle, gyroYangle; // Angle calculate using the gyro only
//uint16_t gyroXanglex, gyroYangley; // Angle calculate using the gyro only
double compAngleX, compAngleY; // Calculated angle using a complementary filter
//uint16_t compAngleXx, compAngleYy; // Angle calculate using the gyro only
double kalAngleX, kalAngleY; // Calculated angle using a Kalman filter
//uint16_t kalAngleXx, kalAngleYy; // Calculated angle using a Kalman filter

uint32_t timer;
uint8_t i2cData[14]; // Buffer for I2C data
```

Figure 4.27. Slave IMU variable declaration

```
/* Update all the values */
while (i2cRead(0x3B, i2cData, 14));
accX = ((i2cData[0] << 8) | i2cData[1]);
ax = (int)accX;
accY = ((i2cData[2] << 8) | i2cData[3]);
ay = (int)accY;
accZ = ((i2cData[4] << 8) | i2cData[5]);
az = (int)accZ;
tempRaw = (i2cData[6] << 8) | i2cData[7];
gyroX = (i2cData[8] << 8) | i2cData[9];
gx = (int)gyroX;
gyroY = (i2cData[10] << 8) | i2cData[11];
gy = (int)gyroY;
gyroZ = (i2cData[12] << 8) | i2cData[13];
gz = (int)gyroZ;

double dt = (double)(micros() - timer) / 1000000; // Calculate delta time
timer = micros();
tm = micros();
```

Figure 4.28. Recurring Variable Assignment on a Serial Interface

**Package Data in Binary Packets and wait for call command from the Arduino Due**

The data from serial is read and wrapped in binary data packets or buffers with a signature call byte as a header byte. All wrapped data packet would then be transmitted through the designated serial line on the slave IMU to the Master IMU on reception of the call command from the Arduino Due serial. Figure 4.29 shows how the IMU slave data was wrapped in a binary packet or buffer before being sent to the Master device.

```
pitchBuffer[0] = '$';    // assign a unique starter byte $ to the first position on the buffer
pitchBuffer[1] ='_';     // assign a unique starter byte _ to the second position on the buffer
pitchBuffer[2] ='$';     // assign a unique starter byte $ to the third position on the buffer

/*The first 32 bit integer variable 'tm' will be byte shifted in the right order before being casted
 *as 4 unsigned 8 bit integer variables in successive positions on the buffer, the same will be done
 *for the other 16 bit variables which will also be broken down and casted as 8 bit unsigned integers and stored in
 *positions on the declared buffer
 */
pitchBuffer[3] = (uint8_t)((tm >> 24) & 0xFF);
pitchBuffer[4] = (uint8_t)((tm >> 16) & 0xFF);
pitchBuffer[5] = (uint8_t)((tm >> 8) & 0xFF);
pitchBuffer[6] = (uint8_t)( tm & 0xFF);
pitchBuffer[7] = (uint8_t)(rollx >> 8);
pitchBuffer[8] = (uint8_t)(rollx & 0xFF);
pitchBuffer[9] = (uint8_t)(compAngleXx >> 8);
pitchBuffer[10] = (uint8_t)(compAngleXx & 0xFF);
pitchBuffer[11] = (uint8_t)(gyroXanglex >> 8);
pitchBuffer[12] = (uint8_t)(gyroXanglex & 0xFF);
pitchBuffer[13] = (uint8_t)(pitchx >> 8);
pitchBuffer[14] = (uint8_t)(pitchx & 0xFF);
```

Figure 4.29. Wrapping the IMU Slave Data in Binary Packets

From Figure 4.29, a buffer was created and each position in the buffer was assigned a variable which has been byte shifted and casted as an 8-bit unsigned integer, before been sent out to the Master device with the command line, *Serial.write(pitchBuffer, sizeof(pitchBuffer)).* The variables were casted as an 8-bit unsigned integer because the Arduino *Serial.Write* function only supports the transfer of bytes of binary data at the time of this project.

**Sending Call command across multiple serial interfaces**

On the Arduino Due interface, multiple virtual serial ports are initialized (see Figure 4.30) to allow multiple data channels for each of the IMU slave devices. Call commands from methods (see Figure 4.31) are then sent from the Arduino Due to each of these virtual serial ports (see Figure 4.32) on the slave IMUs to initiate synchronous data transmission to the Arduino Due.

```
void setup() {
Serial.begin(115200);   //set transmission baud rate on Serial to 115200
Serial1.begin(115200);  //set transmission baud rate on Serial1 to 115200
Serial2.begin(115200);  //set transmission baud rate on Serial2 to 115200
Serial3.begin(115200);  //set transmission baud rate on Serial3 to 115200
pinMode(5, OUTPUT);     //set pin 5 as an ouput pin
}
```

Figure 4.30. Initializing the Serial Ports

```
 void establishUno(){
  Serial1.write('k'); //Send a byte k to Serial1
}
 void establishDeux(){
Serial2.write('q');  //Send a byte q to Serial2
}
void establishTrois(){
Serial3.write('p');  //Send a byte p to Serial3
}
```

Figure 4.31. Call command methods

```
void humble(){
establishDeux();// Send command to request for data from the second slave device
while (Serial2.available()) {  //check if data is available on Serial2
    for (int i = 0; i < 39; i++){  // a for loop that incrementally reads data into a declared buffer from Serial2
   incomBuffer2[i] = Serial2.read();
  }
```

Figure 4.32. Send call command to request for data from Serial

**Call command received? (Handshaking)**

The Arduino Due will await data from each of the three slave IMU's before assigning the received data to a buffer (see Figure 4.33a and b). This essentially is known as handshaking. The Arduino Due would send a request to the slave device for the data and only on receipt of this request will the slave device then send the requested data as a response. This is essential to maintain data integrity during transmission. Figure 4.33a illustrates how one of the slave IMU's await a byte command from Serial before proceeding to package the data (see Figure 4.20) for sending. On Figure 4.33b the Master device would verify if the header bytes are available before reading in the data.

```
if (Serial.available()){    // Check if there are any bytes on Serial
  char command = Serial.read();  // If there are bytes , read in the first byte and store it as a char variable
    switch(command){             // This switch statement will check if the char variable correlates with the byte 'k'.
        case 'k':                // if the char variable is equal to k
        pitchBuffer[0] = '$';    // assign a unique starter byte $ to the first position on the buffer
        pitchBuffer[1] ='_';     // assign a unique starter byte _ to the second position on the buffer
        pitchBuffer[2] ='$';     // assign a unique starter byte $ to the third position on the buffer
```

Figure 4.33a. Handshaking between the Master/Slave IMU's

```
if(incomBuffer1[0] == '$'){  // if the first byte of data is equivalent to $ proceed to the next line
    if(incomBuffer1[1] == '_'){ // if the second byte of data is equivalent to _ proceed to the next line
      if(incomBuffer1[2] == '$'){ // if the third byte of data is equivalent to $ proceed to the next line
/*
 * As the header byte checks out, byte shift the next incoming data into the correct order and assign it to a variable
 * This is made possible by knowing the exact order in which they were sent from the Arduino Nano slave device.
 */
int32_t bytOne = (incomBuffer1[3] << 24)| (incomBuffer1[4] << 16) | (incomBuffer1[5] << 8) | incomBuffer1[6];
int16_t rollOne = incomBuffer1[7] << 8 | incomBuffer1[8];
int16_t pitchOne = incomBuffer1[9] << 8 | incomBuffer1[10];
int16_t yawOne = incomBuffer1[11] << 8 | incomBuffer1[12];
int16_t psiOne =incomBuffer1[13] << 8 | incomBuffer1[14];
int16_t thetaOne =incomBuffer1[15] << 8 | incomBuffer1[16];
int16_t phiOne =incomBuffer1[17] << 8 | incomBuffer1[18];
int16_t axOne = incomBuffer1[19] << 8 | incomBuffer1[20];
int16_t ayOne = incomBuffer1[21] << 8 | incomBuffer1[22];
int16_t azOne = incomBuffer1[23] << 8 | incomBuffer1[24];
int16_t gxOne =incomBuffer1[25] << 8 | incomBuffer1[26];
int16_t gyOne =incomBuffer1[27] << 8 | incomBuffer1[28];
int16_t gzOne =incomBuffer1[29] << 8 | incomBuffer1[30];
int16_t qwOne = incomBuffer1[31] << 8 | incomBuffer1[32];
int16_t qxOne =incomBuffer1[33] << 8 | incomBuffer1[34];
int16_t qyOne =incomBuffer1[35] << 8 | incomBuffer1[36];
int16_t qzOne =incomBuffer1[37] << 8 | incomBuffer1[38];
```

Figure 4.33b. Handshaking between the Master/Slave IMU's

**Wrap Data in a Binary Packet**

After all the data has being received from all three IMU's on the Arduino Due, it is then byte shifted into the appropriate byte format and wrapped with a header byte before being transmitted via cables to the base station.

```
void loop() {
dataBuffer[0] = '@';   //assign a unique starter byte @ to the first position on the buffer
dataBuffer[1] = '_';   //assign a unique starter byte _ to the second position on the buffer
dataBuffer[2] = '@';   //assign a unique starter byte @ to the third position on the buffer
grumble();             // execute the method grumble to read in data from Serial 1
humble();              // execute the method hrumble to read in data from Serial 2
bumble();              // execute the method brumble to read in data from Serial 3
Serial.write(dataBuffer, sizeof(dataBuffer));  // send the buffered data over serial to the computer
delay(1);              // wait 1milli second before beginning the loop again
}
```

Figure 4.34. Send the Wrapped Data to the Computer

## 4.6.2 The IntelliJ Software Architecture

The next flow chart would highlight the software architecture and flow of data on the Intelli J IDE.

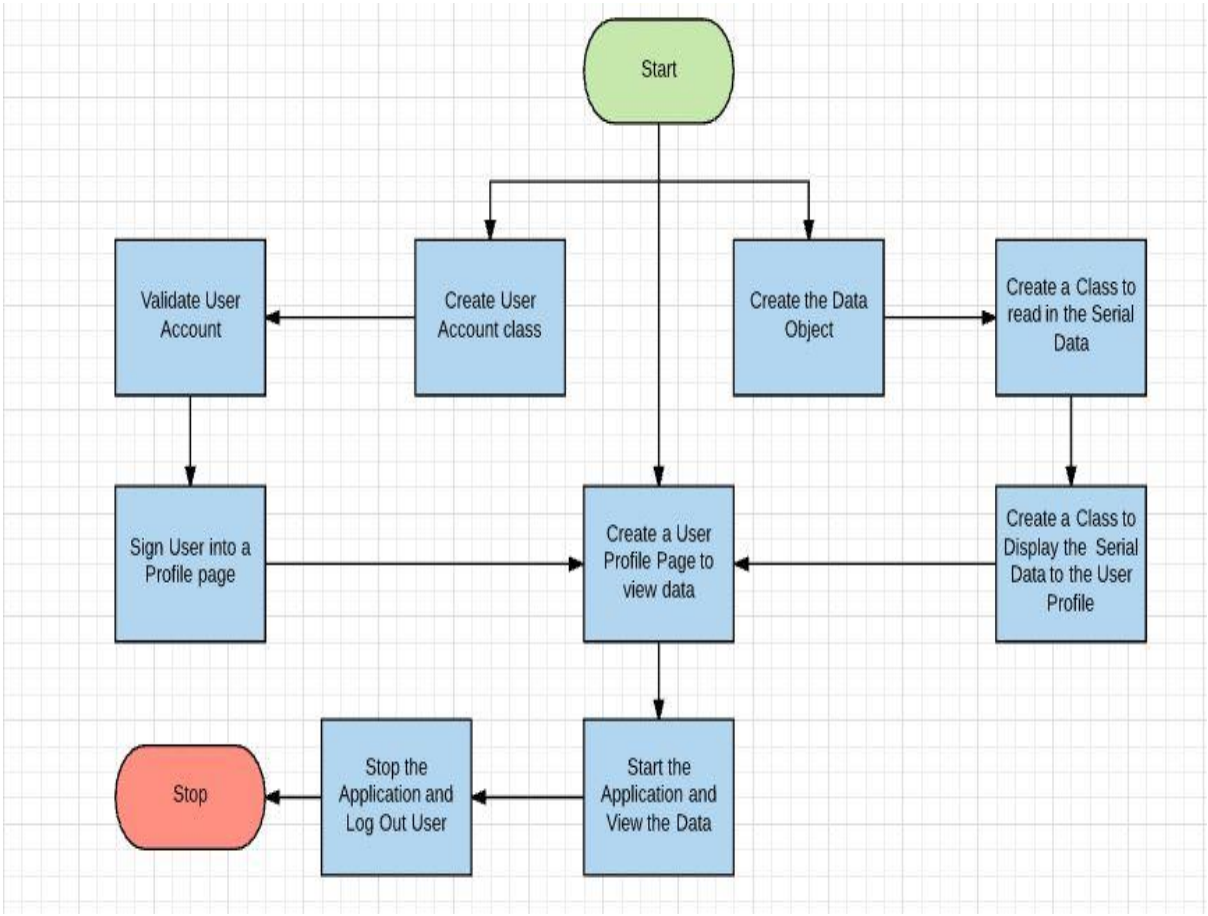


Figure 4.44. Flow Chart of the Software Architecture on Intelli J

Create User Account Class

The User Account Class is created with two unique variables set as the main parameters, a username and a password. Figure 4.45 is a screenshot image of the actual Account class written in Java. This essentially simplifies the requirements for the User to access the application

```
/**
 * The User Account Class
 */
public class Account {
    private String name;   //  Declare a String variable to represent the User's Name
    private String password; //  Declare a variable to represent the User's Password

    public Account(String name, String password) { //Create the Account constructor and pass two parameters
        this.name = name;                          // Initialize the variable name
        this.password = password;                  // Initialize the variable password
    }
```

Figure 4.45. A Screenshot of the User Account Class

**Validate User Account**

The User Account being created will have to be parsed as an XML document for ease of access. The Account parser class would enable the User Account to be created, read and stored on an XML file format. This would eliminate the need of having a database to hold the User information and reduce the system complexity. Validation of the User Account would occur when the User tries to sign in into the application and the Account Parser class has a method that does this, as shown in Figure 4.46. If a User does not have a profile on the XML account file he/she will not be able to log in and would be redirected unto a sign-up page. If the User has forgotten either his/her Username or password, the User would have

to sign up again as the application now does not support a username or password reminder function.

```
/*
 * This method is used to check if a username exist on file;
 */

public Element isUserExist(String userName) {
    //get the root elements in the document variable and pass them on into the Element object root
    Element root = document.getRootElement();
    Element newEle = null;  //initialize another Element object as null.
    for (Element ele : root.elements()) { //This for loop would iterate over the number of elements in root
        //If the username is equal to any variable named 'name' in the element object
        //assign the name to the new element object and return it.
        if (userName.equals(ele.attributeValue( s: "name"))) {
            newEle = ele;
            return newEle;
        }
    }
    return null;
}
```

Figure 4.46. User Account Validation

**Sign in User into a Profile page**

After the User Account has been validated, the User can now log in to the application and access his profile page. When the User inputs his/her username and password, a method in the Account Parser class would check if the details are in correspondence with what's on file and sign in the User unto the main window. Figure 4.47 is a screenshot of the method implemented.

```
//This boolean method is used to validate the Login account
public boolean signIn(String userName, String pwd) {  //a boolean method with the name and password parameters passed in
    Element ele = isUserExist(userName);  // This will check if the Username being passed exist on file
    if (ele != null) {   //provided there is a Username
        String pwdVal = ele.attributeValue( s: "password"); //initialize a string variable to hold a password on file
        if (pwd.equals(pwdVal)) {  // check to see if both passwords match
            currentAccount = new Account(userName, pwd);  //allow the user to log in
            return true;
        } else {  // if a username doesn't exist
            return false;
        }
    } else {
        return false;
    }
}
```

Figure 4.47a. Sign in User

**Create a User Profile Page/Frame**

After the User would have successfully signed in, he/she would be redirected unto the Main window frame. The MainWindow class will then handle the processing of the sign in action as well as the sign-up action and would redirect a user, if the sign in process is successful, to a profile page or container called the IMU container handled by the IMU container class as shown in Figure 4.48.

```
//process sign in action
if ("Sign In".equals(e.getActionCommand())) {
    if (accountParser.signIn(userName, new String(pwd))) {
        JOptionPane.showConfirmDialog(frame,  message: "Welcome " + userName ,  title: "You are now in the Sunken Place!", JOptionPar
        frame.setVisible(false);
        IMUContainer container=new IMUContainer();
        container.setVisible(true);

    } else {
        JOptionPane.showConfirmDialog(frame,  message: "Username and Password do not match, try again",  title: "Error", JOptionPane.
    }
} else if ("Sign Up".equals(e.getActionCommand())) {
    //process sign up
    if (accountParser.signUp(userName, new String(pwd))) {
        int res = JOptionPane.showConfirmDialog(frame,  message: " Your account was sucessfully created ,You may login ",  title: "Y

        //back to login window
        ecgIMUUserSignLabel.setText("LCIMU User Sign In");
        registerPanel.setVisible(true);
        signInButton.setText("Sign In");
        singTag = 0;
    } else {
        JOptionPane.showConfirmDialog(frame,  message: "Username '" + userName + "' exists,please choose another.",  title: "Error",
    }


}
```

Figure 4.48b. Sign in User

The IMU container class has a method used to create the appearance of the User profile interface. This includes a start button, stop button, a welcome panel, log out button and a table display as shown in Figure 4.49. On pressing the start/stop button the User would be able to begin the display of the data captured and view the data displayed on a table in his/her profile frame as executed by the main constructor in the IMUContainer class shown in Figure 4.50. Also

```
private void createUIComponents() {
    // Create the UI Components
    btnPanel = new JPanel();
    outPanel = new JPanel();
    Font buttonFont = new Font( name: "Microsoft YaHei", Font.PLAIN, size: 14);
    BEButtonUI beButtonUI = new BEButtonUI();  // declare a BEButtonUI object
    BEButtonUI startButtonUI = new BEButtonUI(); // declare a BEButtonUI object named for the startButton
    beButtonUI.setNormalColor(BEButtonUI.NormalColor.red);
    startButtonUI.setNormalColor(BEButtonUI.NormalColor.blue);
    startBtn = new JButton( text: "Start");  //Label the start button
    startBtn.setFont(buttonFont); //set the start button fonts
    startBtn.setMargin(new Insets( top: 10,  left: 10,  bottom: 10,  right: 10)); //set the start button margin
    startBtn.setUI(startButtonUI); // set the color of the start button
    stopButton = new JButton( text: "Stop");  //Label the stop button as 'stop'
    stopButton.setFont(buttonFont); //set the stop button font
    stopButton.setMargin(new Insets( top: 10,  left: 10,  bottom: 10,  right: 10));//set the stop button margin
    stopButton.setUI(beButtonUI); // set the color of the stop button
    outPanel.add(btnPanel);
    btnPanel.add(startBtn);
    canvasPanel = new JPanel();
    labelPanel = new JPanel();
    welcomeLabel = new JLabel();
    welcomeLabel.setSize(new Dimension( width: 400,  height: 48)); //set the size of the Welcome Label panel
    welcomeLabel.setFont(new Font( name: "Courier New",  style: Font.BOLD | Font.ITALIC,  size: 18)); // set the font
    welcomeLabel.setBackground(Color.orange);  // set the background color

    logOutButton = new JButton( text: "Log Out");  // the label of the log out button
    logOutButton.setFont(buttonFont);  // the log out button font
    logOutButton.setMargin(new Insets( top: 10,  left: 10,  bottom: 10,  right: 10)); // set the logout button margins


    initDataTable();
```

Figure 4.49. The User Profile Interface structure

```
public IMUContainer() {
    container = this;
    setTitle("The LCIMU Application User Interface");  //Set the title of the Jframe
    setContentPane(outPanel);
//      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(new Dimension( width: 600,  height: 320));  //set the size of the frame
    setLocationRelativeTo(null);    // set the location of the frame relative to no component

    //start the CanvasGraph
    startBtn.addMouseListener((MouseAdapter) mouseClicked(e) → { // When the start button is clicked listen for a res
            initCanvasGraph();  //execute the method initCanvasGraph
            super.mouseClicked(e); //to ensure that the mouseClicked event is not overridden
    });

    setExtendedState(JFrame.MAXIMIZED_BOTH); //maximize the Jframe

    //stop the CanvasGraph
    stopButton.addMouseListener((MouseAdapter) mouseClicked(e) → {// When the stop button is clicked listen for a res
            destroyCanvasGraph(); // the response awaited
            super.mouseClicked(e);//to ensure that the mouseClicked event is not overridden
    });

    //Log out the user
    logOutButton.addMouseListener((MouseAdapter) mouseClicked(e) → {// When the logout button is clicked listen for a
            super.mouseClicked(e); //to ensure that the mouseClicked event is not overridden
            container.dispose();  // dispose of the IMUContainer
            new MainWindow().main(new String[]{});
    });
}
```

## Figure 4.50. The Main User Profile Interface Class

**Create the Data Item Object**

The Data Item Object would consist of all the variables being read in from the serial port. The main variables are Time, Roll, Pitch, Yaw, x-axis angular acceleration, y-axis angular acceleration, z-axis angular acceleration, x-axis acceleration, y-axis acceleration, z-axis acceleration, psi, theta, phi, w-axis quaternion, x-axis quaternion, y-axis quaternion and z-axis quaternion. These variables are all tripled to represent the values of the three IMU slave devices. Figure 4.51 is a screenshot of the DataItem class and its variables being initialized.

```
/**
 * The DataItem class will be used to instantiate
 * all the incoming serial sensor data variables
 */
public class DataItem {

    private int time_val=0;  // Initialize the sensor time variable
    private float rollone_val=0; // Initialize the roll variable of the first sensor
    private float pitchone_val=0;// Initialize the pitch variable of the first sensor
    private float yawone_val=0;// Initialize the yaw variable of the first sensor
    private float psione_val=0;// Initialize the psi variable of the first sensor
    private float thetaone_val=0;// Initialize the theta variable of the first sensor
    private float phione_val=0;// Initialize the phi variable of the first sensor
    private float axone_val=0;// Initialize the x-axis acceleration variable of the first sensor
    private float ayone_val=0;// Initialize the y-axis acceleration variable of the first sensor
    private float azone_val=0;// Initialize the z-axis acceleration variable of the first sensor
    private int gxone_val=0;// Initialize the x-axis angular acceleration variable of the first sensor
    private int gyone_val=0;// Initialize the y-axis angular acceleration variable of the first sensor
    private int gzone_val=0;// Initialize the z-axis angular acceleration variable of the first sensor
    private int qwone_val=0;// Initialize the w-axis quaternion variable of the first sensor
    private int qxone_val=0;// Initialize the x-axis quaternion variable of the first sensor
    private int qyone_val=0;// Initialize the y-axis quaternion variable of the first sensor
    private int qzone_val=0;// Initialize the z-axis quaternion variable of the first sensor
```

Figure 4.51. The Data Item Class variables initialized

**Create a Class to Read in Serial Data**

The SerialLogger class was created to read in Serial Data from the serial port on the local system. The SerialLogger constructor was responsible for implementing the RXTX library and checking to retrieve the available ports on the local system and displaying these ports as shown in Figure 4.52. The SerialEvent class on the other hand would initiate communication with a port with available data, configure the port and begin transmission of the data as shown in Figure 4.53.

```
public SerialLogger(ConcurrentLinkedQueue<DataItem> dataQ, String serial) {

    this.dataQ = dataQ;  //instantiate the concurrent array variable as dataQ

    serialPort = null; //initialize the serialPort variable as null
    portList = CommPortIdentifier.getPortIdentifiers(); // retrieve the list of ports

    //Check if any other elements are available on the port list
    while (portList.hasMoreElements()) { //while there are more elements available
        portId = (CommPortIdentifier) portList.nextElement(); //assign an id to the element
        if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL) { // assign a type to the port Id
            if (portId.getName().equals(serial)) {  //if the portId is equal to the string variable passed
                try {
                    serialPort = (SerialPort) portId.open( s: "LCIMU Application",  i: 2000); //open the port
                } catch (PortInUseException e) { // throw an exception error if the port cannot be opened
                    System.out.println(e);  //print out the error exception
                }
            }
        }
    }

    if (serialPort == null) {  // if there are no serial ports found

        System.out.println("Serial Port not found...Exiting."); // send out a notification
        System.exit( status: 1); //exit after one milli seconds

    } else {
```

Figure 4.52. Checking for available ports

```
public void serialEvent(SerialPortEvent serialPortEvent) {
    switch (serialPortEvent.getEventType()) {

        case SerialPortEvent.BI:  //Break interrupt.
        case SerialPortEvent.OE:  //Overrun Error
        case SerialPortEvent.FE:// Framing error.
        case SerialPortEvent.PE://Parity error.
        case SerialPortEvent.CD://Carrier detect
        case SerialPortEvent.CTS://Clear to send.
        case SerialPortEvent.DSR://Data set ready.
        case SerialPortEvent.RI: //Ring indicator.
        case SerialPortEvent.OUTPUT_BUFFER_EMPTY: //Output buffer is empty.
            break;
        case SerialPortEvent.DATA_AVAILABLE: //Data available at the serial port.
            //byte[] buff = new byte[111];
            try {
                byte the_byte = 0;   //initialize a byte to null
                while (inputStream.available() > 0) { //if there is data available
                    synchronized (syncObj) {
                        if (tag != 0) {
                            return;
                        }
                    }
                    the_byte = (byte)inputStream.read();  //read in the first byte and cast it as a byte
                    if(the_byte == '@'){  //if the byte variable is equal to '@' proceed
                        the_byte = (byte)inputStream.read();  //read in the second byte and cast it as a byte
                        if(the_byte == '_') {  //if the byte variable is equal to '_' proceed
                            the_byte = (byte) inputStream.read();//read in the third byte and cast it as a byte
                            if (the_byte == '@') {//if the byte variable is equal to '@' proceed
                                int  dataB_1 =  Byte.toUnsignedInt((byte) inputStream.read()); // the next byte would be caste
                                int dataB_2 =  Byte.toUnsignedInt((byte) inputStream.read());//the same will be done for this
```

Figure 4.53. Connecting to a Serial Port

**Display the Captured Data on the User Profile**

As shown in Figure 4.50. The IMUContainer class is responsible for executing the data display by a call to the method initCanvasGraph. This method triggers the CanvasGraph class instance and displays the data captured on the User Profile. Figure 4.54a is a screenshot of this method.

```
/*
 * This method allows the User to
 * click the 'start' button to start the application
 * and Display the Data captured
 */
public void initCanvasGraph() {
    System.out.println("Display the Data");
    datas = new Vector<DataItem>();
    SwingUtilities.invokeLater(new Runnable() {// wait until the call to run as being executed before updating the thread
        @Override
        public void run() { CanvasGraph.getInstance().initAndShowUI(); }
    });
}
```

Figure 4.54a. The Data Display Method on the User Profile Interface

**Start the Application**

When the User clicks on the Start button. The initCanvasGraph method is executed and the data is displayed as illustrated by the methods shown in Figure 4.50 and 4.54 respectively. The CanvasGraph class has a main method that returns an instance of the class method initAndShowUI, which was illustrated in Figure 4.54. Figure 4.54b is a screenshot of the method instantiated as it is on the CanvasGraph class.

```
/*This method would initialize and display the UI
 * when called by default
 */

public void initAndShowUI() {
    setVisible(true); //Make the UI visible

    //Update this GUI component from an outside thread
    Platform.runLater(new Runnable() {
        @Override
        public void run() {
            try {
                initFX(jfxPanel); //This method will update the thread from outside
                start();
            } catch (Exception e) {//Throw Exception
                e.printStackTrace();
            }
        }
    });
}
```

Figure 4.54b. The Data Display Method on the User Profile Interface

**Stop the Application and Log out user**

As shown in Figure 4.50, the User can stop the data capture process and remain on his/her profile unless he/she chooses the log out of his profile by which the IMUContainer class will be disposed of and the User redirected to the sign in frame.

## 4.6.2 The MATLAB Software Architecture

The software structure on the MATLAB IDE is geared towards a simple analysis of the data captured from the LCIMU system to derive the power expended when a User undergoes an activity.

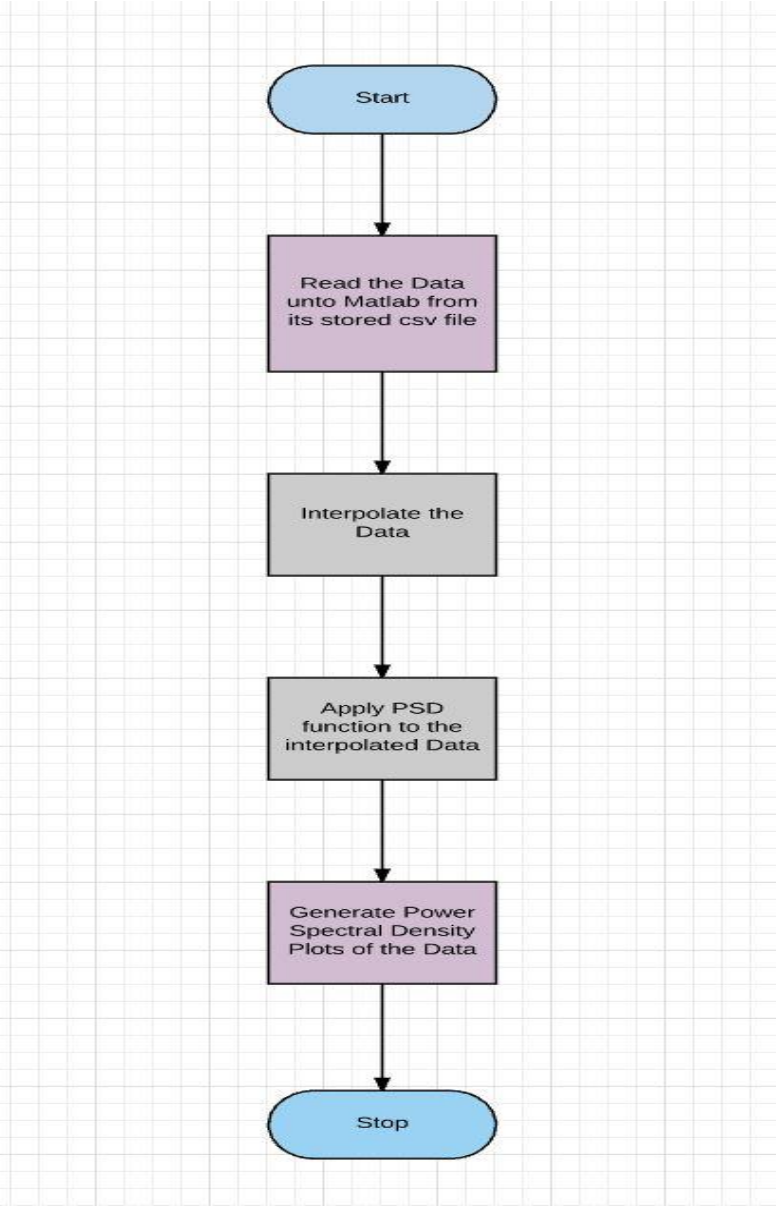


Figure 4.55. A Flow Chart of the program flow on MATLAB

Read the data unto MATLAB

On MATLAB, the data is read in from the file in which it was stored on the base station or computer. The data is now then stored in multiple declared variables on the IDE. Figure 4.56 is a screenshot of a section of the MATLAB code that reads in the data from the csv file. It is important to point out here that the sensor data was not converted on IntelliJ to improve the speed of execution of the programs. The conversion was carried out in MATLAB as can be seen below.

```
data = csvread('2017_04_29_08_15_46.csv',2);
lili = data(:,1);      % This would hold the time variable
Roll1 = data(:,2)*(180/pi); % This would hold the first Roll variable
Pitch1 = data(:,3)*(180/pi); % This would hold the first Pitch variable
Yaw1 = data(:,4)*(180/pi);% This would hold the first Yaw variable
Acc1x = data(:,5) * 0.061; % This would hold the the first sensor x-axis acceleration variable
Acc1y = data(:,6)* 0.061; % This would hold the first sensor y-axis acceleration variable
Acc1z = data(:,7)* 0.061; % This would hold the first sensor z-axis acceleration variable
Gyro1x = data(:,8)/100;% This would hold the first sensor x-axis angular acceleration variable
Gyro1y = data(:,9)/100;% This would hold the first sensor y-axis angular acceleration variable
Gyro1z = data(:,10)/100;% This would hold the first sensor z-axis angular acceleration variable
Q1w = data(:,11)*(1.80/pi);% This would hold the first sensor w-axis quarternion variable
Q1x = data(:,12)*(1.80/pi);%This would hold the first sensor x-axis quarternion variable
Q1y = data(:,13)*(1.80/pi);%This would hold the first sensor y-axis quarternion variable
Q1z = data(:,14)*(1.80/pi);%This would hold the first sensor z-axis quarternion variable
```

Figure 4.56. Reading in the Data on MATLAB

**Interpolate the data**

The data is then interpolated to maintain sampling integrity and enable better data analysis. Figure 4.47 gives an illustration of this process being implemented.

```
Time_Length1 = (TimeSensor(length(TimeSensor))  - TimeSensor(1)); % the length of the time data
Period_1 = (Time_Length1/length(TimeSensor));  % The period
Sample_rate1 = (1/Period_1) * 1000000; %The sample rate
%% Interpolate the Sensor Data to a fixed sample rate using interp1 function
xq1 = TimeSensor(1):Period_1*1.33:TimeSensor(length(TimeSensor)); % The adjusted sample  frequency
Time_Sensorint = interp1(TimeSensor,TimeSensor,xq1,'linear'); % The interpolated Time variable for the first
Roll1_int = interp1(lili,Roll1,xq1,'cubic');
Pitch1_int = interp1(lili,Pitch1,xq1,'cubic');
Yaw1_int = interp1(lili,Yaw1,xq1,'cubic');
Acc1x_int = interp1(lili,Acc1x,xq1,'cubic');
Acc1y_int = interp1(lili,Acc1y,xq1,'cubic');
Acc1z_int = interp1(lili,Acc1z,xq1,'cubic');
Gyro1x_int = interp1(lili,Gyro1x,xq1,'cubic');
Gyro1y_int = interp1(lili,Gyro1y,xq1,'cubic');
Gyro1z_int = interp1(lili,Gyro1z,xq1,'cubic');
Q1w_int = interp1(lili,Q1w,xq1,'cubic');
Q1x_int = interp1(lili,Q1x,xq1,'cubic');
Q1y_int = interp1(lili,Q1y,xq1,'cubic');
Q1z_int = interp1(lili,Q1z,xq1,'cubic');
```

Figure 4.57. Interpolating the Data

**Apply Power Spectral Analysis function to the Interpolated Data**

On MATLAB, a unique function being called (*pwelch*) allows the power spectral analysis to be carried out on the interpolated data. The power spectral density of the variables will then be derived. Figure 4.58 is a screenshot of this illustration.

```
%% The power spectral density for each components will be calculated below
[pax, fraq] = pwelch(modData1,[],[],[],67);
[pex, freq] = pwelch(modData2,[],[],[],67);
[pix, friq] = pwelch(modData3,[],[],[],67);
[pox, froq] = pwelch(modData4,[],[],[],67);
[pux, fruq] = pwelch(modData5,[],[],[],67);
[puxy, fruqy] = pwelch(modData6,[],[],[],67);

[pack, lacs] = findpeaks(pax);
[peck, lecs] = findpeaks(pex);
[pick, lics] = findpeaks(pix);
[pock, locs] = findpeaks(pox);
[puck, lucs] = findpeaks(pux);
[pucky, lucsy] = findpeaks(puxy);
```

Figure 4.58. Analysing the Data with PSD

**Generate Power Spectral Density Diagrams**

Plots representing the power spectral density of the aggregated data variables being analysed can then be generated for analysis.

```
%% The plots of the different components
plot(fraq(lacs),db(pack),'DisplayName','Chest Sensor');
hold on
plot(freq(lecs),db(peck),'g--','DisplayName','Left-Hand Sensor');
hold on
plot(friq(lics),db(pick),'r--','DisplayName','Right-Hand Sensor');
legend('show')
t = xlabel({'Frequency','(Hz)'});
t.Color = 'blue';
ball = ylabel({'Power','(db)'});
ball.Color = 'red';
title('PSD of the mean acceleration of an individual Cycling Bar (Forearm Upper-Limb Placement) ');
```

Figure 4.59. Plotting the PSD of the data components

## 4.7 Performance

The performance of the LCIMU system in terms of its rate of sampling was carried out with the aid of a logic level analyser. A logic level analyser is used to adjust the timing and sampling rate of the master IMU. A digital pin ( pin 5, see Figure 4.30) from the master IMU is set high and was connected to the logic analyser(Saleae 2016). The sampling rate of the IMU fell within the range of 50 – 150Hz. The master IMU was then adjusted until a steady sampling rate of 67Hz over an extended period was obtained. Figure 4.60 is an image of a Saleae logic analyser and the Saleae platform.

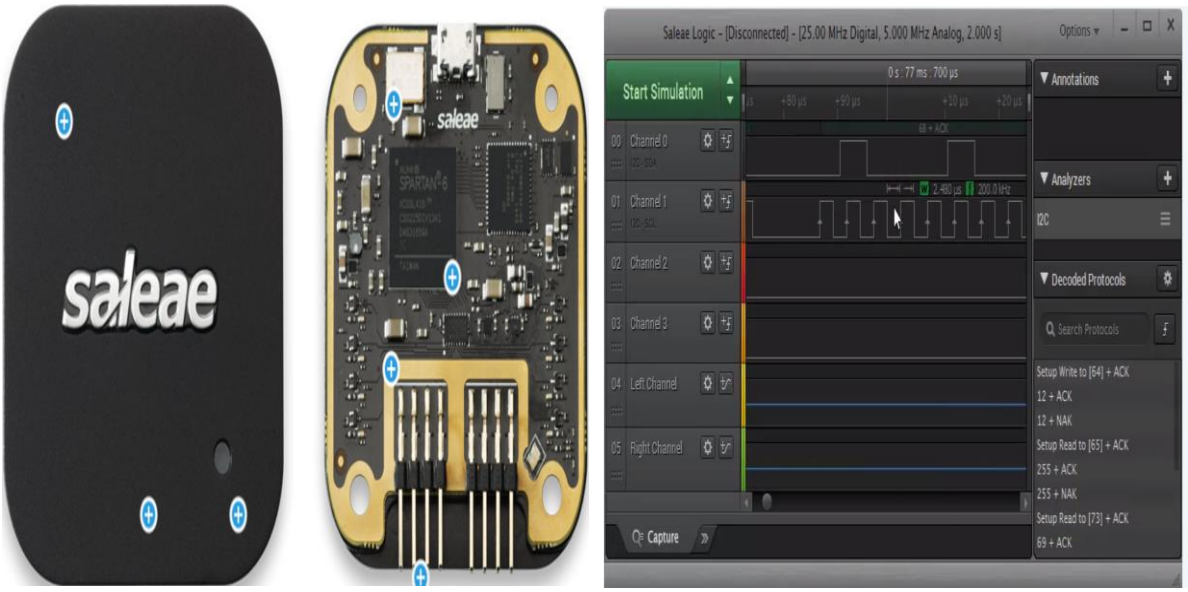


Figure 4.60. A Saleae Logic Analyser

## 5.0 Product Design Specification Approval

The undersigned acknowledge they have reviewed the LCIMU **Product Design Specification** document and agree with the approach it presents. Any changes to this Requirements Definition will be coordinated with and approved by the undersigned or their designated representatives.

Signature: _____ Date: _____

Print Name: _____

Title: _____

Role: _____

Signature: _____ Date: _____

Print Name: _____

Title: _____

Role: _____

Signature: _____ Date: _____

Print Name: _____

Title: _____

Role: _____

# Appendix A

Adafruit (2014). "Through-Hole Resistors ". from https://www.adafruit.com/product/2784.

Arduino (2016). "Arduino Due."

Arduino (2016). "Ardunio Nano." from https://www.arduino.cc/en/Main/ArduinoBoardNano.

DFROBOT (2016). "ProtoBoard - Rectangle 2" Double Sided." from https://www.dfrobot.com/index.php?route=product/product&path=66_67&product_id=660.

Electronics, H. (2016). "RJ45 8-Pin Connector." from http://www.hobbytronics.co.uk/rj45-socket.

Electronics, S. (2014). "SparkFun Logic Level Converter - Bi-Directional." from https://www.sparkfun.com/products/12009.

Goutorbe, R. (2015). "Arduino Radio Control." from http://www.reseau.org/arduinorc/index.php?n=Main.Connections.

InvenSense (2011). "IDG-2020 & IXZ-2020 Product Specification Revision 1.0." from http://www.invensense.com/mems/gyro/documents/PS-Ixx-2020.pdf.

InvenSense (2013). MPU-6000 and MPU-6050 Product Specification Revision 3.4.

Invensense (2015). "MPU-6050 Six-Axis (Gyro + Accelerometer) MEMS MotionTracking™ Devices."

Jiang, J. (2011). BeautyEye : A Swing look and feel.

Paradigm, B. (2014). "Introduction to I²C and SPI protocols." from http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/

Rouse, M. (2011). "UART(Universal Asynchronous Receiver/Transmitter)."

Rowberg, J. (2015). i2cdevlib.

Saleae (2016). "Saleae Logic : The Logic Analyzer."

Sparkfun (2014). "Serial Communication." from https://learn.sparkfun.com/tutorials/serial-communication.

Sparkfun (2014). "Serial Peripheral Interface (SPI) ". from https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

SparkFun (2016). "Working with wire." from https://learn.sparkfun.com/tutorials/working-with-wire.

# Appendix B: Key Terms

| | |
|---|---|
| IMU | Inertial Measurement Unit |
| LCIMU | Low Cost Inertial Measurement Unit |
| Base Station | The personal computer or Lap top |
| LLC | Logic Level Converter |
| TX | Transmitter |
| RX | Receiver |