1.1 - Synthesis Overall Design

A top-level view of system which I have created can be seen below:

*(Create and Insert Diagram here)*

As you can see in the diagram of the system, the microcontroller is attached to the skateboard, the position of microcontroller has stayed constant throughout the project so that data collection was consistent. Attached to the microcontroller is an SD card module which is used to store the data recorded by the microcontroller during a session, the way in which the SD card module has been attached to the microcontroller will be discussed in detail in section 1.1.1.

When the microcontroller is powered it waits for a connection to be made via Bluetooth, this connection is initiated by a mobile device with an application which can discover Bluetooth Low Energy (BLE) devices as these are not discoverable through either Android or iPhone standard Bluetooth discovery functions.

This may seem like an initial flaw to Bluetooth Low Energy but in fact it has been useful to the project. Firstly, BLE uses considerably less power than standard Bluetooth (insert quote with figure or percentage) which was great for the duty cycle of my microcontroller. I could only afford to attach a small power supply to the skateboard as not affecting the act of skating was something that I wanted to keep to minimum during the development of this project. Another perk to BLE is that once a connection has been made the device is no longer discoverable so it would stop any potential unwanted connection attempts if they were to occur.

When a connection is made between the microcontroller and the mobile then the microcontroller will start recording the data taken from the accelerometer and gyroscope capturing it and storing it onto the SD card.

I chose to store the data in this way as BLE is not as consistent as I would like for transmitting data. As mentioned in the Analysis section a study carried out by Siekkinen et al they found that when testing successful packet transmissions, they only had a success rate of 60%[1]. This would have been okay if I had the luxury of resending a piece of data if it came to light it wasn’t transmitted correctly but I need to store the data the instant that it is generate therefore using the SD card was the right decision. Standard Bluetooth would have probably overcome this problem but still wouldn’t have been perfect, as well as this it would consume way too much power for the microcontroller to have a long enough duty cycle.

The microcontroller will then stop recording data once the user has disconnected the phone from the device. Doing this meant that the microcontroller will not be constantly recording data and wasting space on the SD card.

As the data was no longer being transmitted over BLE the android application was no longer required as an freely available app to anyone with a smartphone called ‘nrfConnect’ [2] [3] could be used to establish a connection with Arduino board which start the capturing of the data. Because of this, the system as a whole is available to a wider range of people but also more robust.

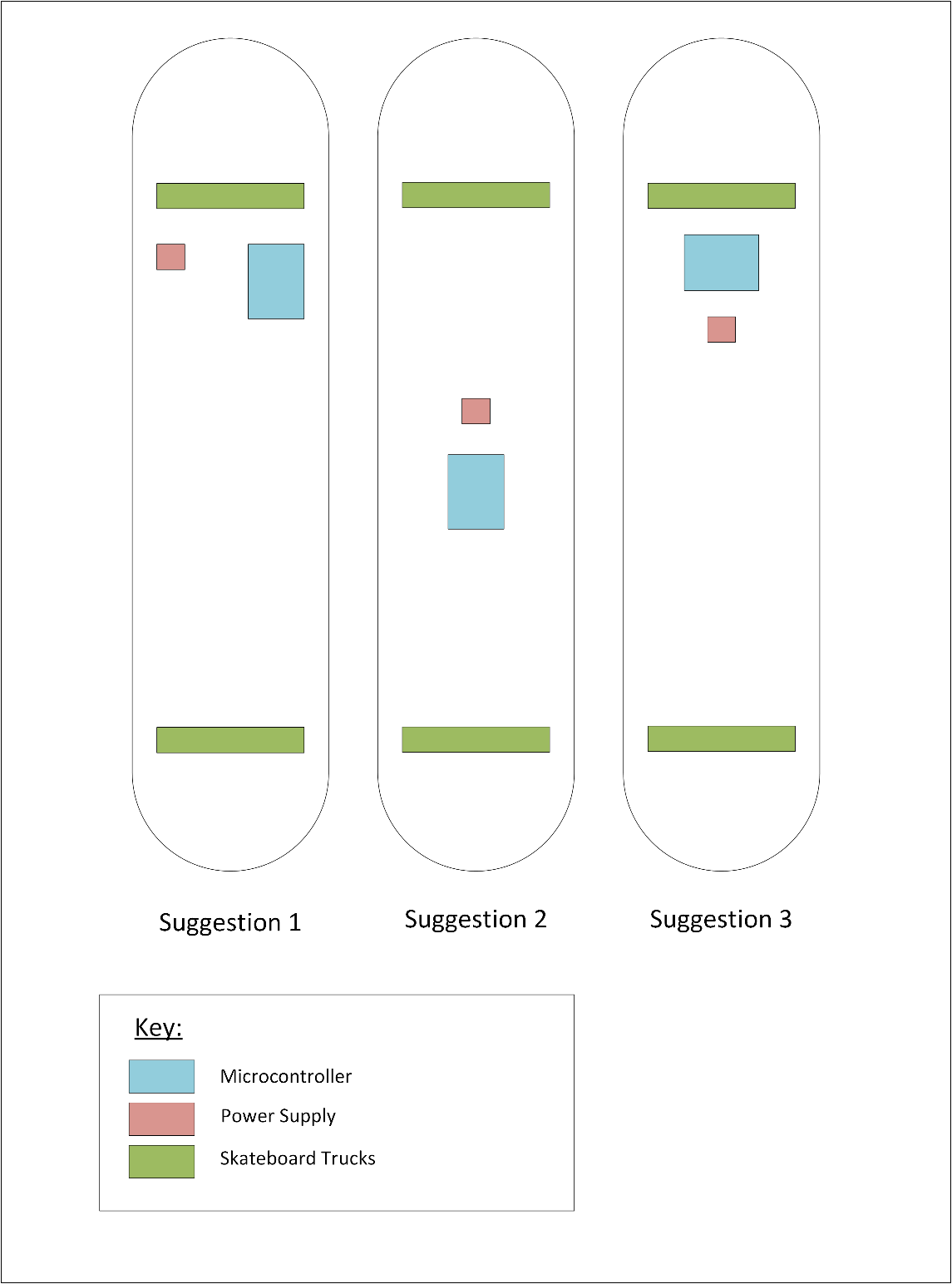
With the data recorded, you can then insert the SD card into a laptop and run it through the trick identification system. I created the trick identification based on graphs created from the initial data I collected when performing certain skateboard tricks. Using these graphs, I created rules that can be applied to the data set recorded on the SD card and if parts of the data meet certain rules then this means a certain trick has been performed. This will be discussed in more detail in section 2.3.

1.1.2 - Position of the Microcontroller on the Skateboard

When first taking on the project I did not think that the position of the position of the microcontroller would be something that would take much thinking about. However, in fact it did take some planning as I had to two take two key factors into account:

1. Minimalism the effect on the act of skating with the microcontroller and power supply present.
2. Reduce the chance of damage to the microcontroller and power supply.

Below are 3 different approaches that I considered to try and protect the microcontroller:



Suggestion 1 was my first proposal having the microcontroller towards the top on the side of skateboard just behind the front truck, trucks attach the skateboards wheels to its deck. However, after getting on the skateboard and riding it I noticed that when turning left and right the side of the skateboard is pushed towards the ground, a bit like how a motorcyclist’s knee does during a turn. This was causing the microcontroller to scrape along the floor while turning, this effected the riding of the skateboard as well putting the microcontroller at risk of being damaged. This could be prevented by putting the microcontroller into a metal case, which what eventually was done. However, the impact on the act of skating made this proposal of the position no good.

Suggestion 2 saw the microcontroller and power supply go right in the middle of the skateboard in terms of both height and width. Due to the observation taken during testing suggestion 1 putting the microcontroller at the centre of the skateboard meant that during turning the microcontroller was no longer coming into contact with the ground. This made Suggestion 2 seem like a perfect solution however I then realised certain tricks performed on a skateboard known as ‘Grind’ tricks often have the centre of the skateboard grind across a ledge of some kind. Therefore, the microcontroller would be no good placed here.

This lead me to suggestion 3 in the above diagram. I knew that the microcontroller could not be placed at either side of the board but also couldn’t go right in the centre of the skateboard. I then decided that placing it just underneath the front truck of the skateboard would be safest place for the microcontroller to go without affect the use of the skateboard. After trying this out I was satisfied I had made the right choice with the truck at the back of the skateboard often used as part ‘Grind’ tricks – the same reason suggestion 2 was not possible.

Methodology Approach

Arduino Software Development Approach

When considering the different software development methodology approaches it was apparent that the software for the Arduino 101 would require a different development approach to the trick identification system. The Arduino 101 software was developed using the waterfall development methodology:



This suited the development cycle of the Arduino software as the requirements for the software were very clear, the cycle of development would be relatively short and tested easily against its requirements. [2] This methodology approach would not be suited to the skate trick identification system as a lot of trial and error would be involved calculating the best rules and bounds to use to distinguish the tricks.

Skate Trick Identification Software Development Approach

Due to the trial and error basis, the skate trick identification was best suited to a prototyping approach. This approach allows an initial implementation to be tested against the requirements then modified in the areas it fell short. This cycle repeats until you end up with a final and complete solution.



This was the best-suited methodology approach for the trick identification program, as not only did it give a clear understanding of how the software works, it was also highly effective at finding subtle problems within the software due to the many testing phases that go in hand with taking on this methodology approach. [2]

1.1.3 - Connecting the Arduino to the SD Card Module

My initial plan at the start of the project was to send the data over Bluetooth and store it on an SD card via the mobile application. However, when it got around to the implementation I noticed that the recorded data was not as consistent as I would like and found that this is a problem with BLE as previously mentioned in section 1.1.

As a result, I had to purchase an SD card module that would be compatible with my Arduino 101 microcontroller. The process of wiring up the SD card module I purchased with the microcontroller was relatively simple, below is a schematic of the Arduino 101 and SD Card Module attached.

(SCHEMEATIC)

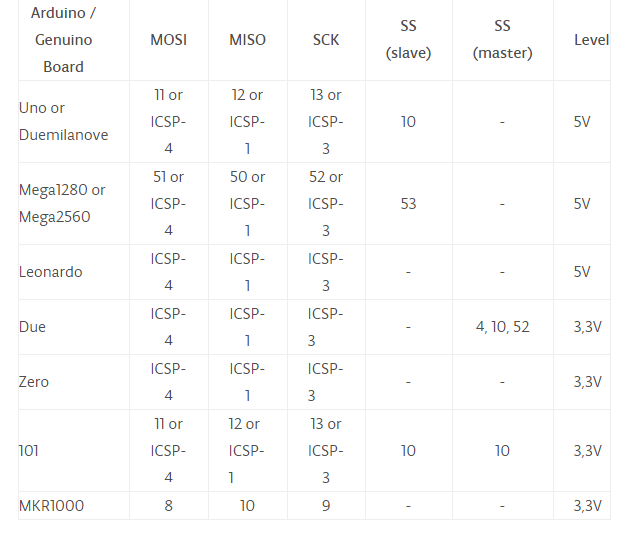
Data Lines

* MISO (Master In Slave Out) – The line used for the slave (the SD card module) to sender data back to the master (The Arduino 101), this is never used in my implementation as I do require the SD card module to send any data or commands to the Arduino 101.
* MOSI (Master Out Slave In) – This line is used by the Arduino 101 to send the data taken from the accelerometer and gyroscope to SD card module to be saved ready to be run through the trick identification system.
* SCK (Serial Clock) – Keeps both the components in correct time to keep data transmission in sync.
* SS (Slave Select) – This pin can be used by the Arduino 101 to enable or disable the SD card module.

Power Lines

* 3.3V and GNR - These lines give the SD card module 3.3V of power to be able to perform its required tasks.

Once I knew the data and power lines I needed to implement this protocol I had to check which pins on the Arduino 101 board these needed to be connected to. Fortunately, Arduino provide a table explaining which pins should be used for the lines required for the SPI protocol. This table can be seen below [2]:



As you can see for the data lines MOSI uses digital pin 11, MISO uses digital pin 12, SCK uses digital pin 13 and SS line uses digital pin 10. The power lines use the pins on the board with matching names 3.3V and GNR to provide the SD card with power.

Through implementing this protocol, I was able to create a plain text file on the SD card module, which is exactly what I need to store the data taken from the accelerometer and gyroscope of the Arduino 101 board. Now that this was done ready to design the rest of the software.

1.2 - Arduino Software Design

When it came down to creating the software required for the Arduino I would be essentially writing my code in C/ C++ as “the Arduino language is merely a set of C/C++ functions that can be called from your code” [3]. The Arduino IDE used to write the code in has a build process which takes care of things such as creating function prototypes.

My first software design decision was whether I was going to use the accelerometer readings on their own, the gyroscope readings on their own or a combination of both. To figure out the answer to this question I started looking at the libraries for the accelerometer and gyroscope to see how you extract the data from them. When doing so I discovered that there was library called Curie IMU which allowed me to compare both accelerometer and gyroscope readings together to a provide a single a combined reading using the *readMotionSensor* function of the Curie IMU library [4].

It was possible to read both the gyroscope and accelerometer separately but decided to use the combined reading instead as it will give a better overall orientation of the board and the orientation of the board is essential to provide accurate data for the trick identification system.

(Talk about range values for accelerometer and gyroscope and values will have to be scaled based on this range)

Another thing that I had to consider was my sample rate, how many times a second would I be taking readings form the accelerometer or gyroscope. Often the act of performing a skate trick is over in less than two seconds, this meant my sample rate was going to be rather high, as I need enough samples to be able to see a pattern in the readings that distinguish the trick.

As I am using the Curie IMU library when looking how to set the sample rate for the accelerometer and the gyroscope I discovered that the sample rates for both components have to be set to specific values. The values that where both realistic and available to me where 25, 50 and 100 [5] [6] samples per second. I decided to choose a sample rate of 50 as I knew that 25 samples would not provide enough data readings to show a good enough pattern in the data based on how long a skate trick takes to be performed.

On the other end of the spectrum, I thought that a sample rate of 100 would leave me with unnecessarily large data files and would cause equally unnecessary strain on the microcontroller and SD card module that was attached. After collecting some data, using a sample rate of 50 I produced graphs that had very clear trends consistently.

Below is a series of graphs to show the pattern the data creates when I moved the board in the way it would during a “Kickflip” below are the graphs of 3 different “Kickflips” I performed by hand:

(Insert consistent graphs of 3 different kickflips from 50 samples a second)

Having such obvious patterns in the data was very pleasing as this meant that the trick identification system was going to be possible as I could establish rules based on these graphs that the system could recognised in data produced by the board, I was also satisfied with my sample rate of 50 based on these graphs.

(Should I talk about BLE Characteristics, as I am not using them anymore?)

Once I was happy with the sample rate my attention was turned to how I would control when data was recorded using Bluetooth Low Energy. Once again, there was a library provided by Arduino for me to use called Curie BLE [7] that has all the relevant methods to establish a Bluetooth connection.

Within the Curie BLE library is a function called *connected* [8].Calling this function checks to see whether there is a connection between the Arduino 101 board and another device. This lead to me to the design decision that a while loop would control data capture. In practice, this would look something like the follow pseudo code:

*While (device connected) {*

*Code for capturing and storing data*

*}*

When trying to investigate possible alternatives it was obvious this was the only logical way I could implement this with the functions that I had available to me.

1.2.1 - Class Diagram

With the nature of the way Arduino software is created, I ended up with only the one class to contain the software required for the board to be functional. Below is the class diagram for my software:

Update class diagram



1.2.1.1 - Class Variables

The filter variable you see at the top of the class diagram is the means of which I will determine the orientation of the board. As filter is of type Madgwick, I can call a function from its library called “*updateIMU*” which will update the filter based on the latest readings from the accelerometer and gyroscope. Once the filter updates, I can then call functions *getRoll*, *getPitch* and *getYaw* (part of the Madgwick library) to extract the values I need.

By calling these methods, I can assign them to the roll, pitch and heading variables of my class, which are of type float as the *getRoll*, *getPitch* and *getYaw* return values of this type. The *roll*, *pitch* and *heading* variables represent the following in terms of the board’s orientation:

* Roll – This represents a part or whole of the board moving up and down vertically. One skate trick, known as manual, is something I would expect to effect the way in which the roll value changes. Consider this description of the trick – “*A trick similar to a bicycle wheelie where the rider balances with the front or back wheels off and without the tail or nose on the ground.”*  [9]
* Pitch – This represents any rotation of the board.
* Heading – as its name suggests this value represents in which way the board is heading.

The *motionReading* variable contains converted readings of the roll, pitch and heading values to be saved as one line of text delimited by commas to a file on the attached SD card. This means that the data is formatted and ready to input to the trick identification system without any further manipulation after the data capture. I believe this to be a solid design choice as it eliminated any time wasted on getting the data ready for the trick identification system or implementing a more convoluted way of getting the data into the trick identification system as I could use a simple string splitting function to extract the relevant readings.

The *blePeripheral* variable in essence will enable the boards Bluetooth Low Energy capabilities. Using functions of the BLE Peripheral class [10] I can give the Arduino 101 a name readable to humans as well as a UUID, a 128-bit number to uniquely identify the Arduino 101 board. This UUID is will be generated using the *bleService* variable seen in the class diagram.

1.2.1.2 - Class Methods

*Setup (void)* - This method will be used to initialise anything that needs it. In the case of my software, this method will initialise the filter and set both the rates and the range for the accelerometer and gyroscope to their desired values (As laid out in section 1.2). It will initialise everything required to make the Arduino 101 board discoverable by another Bluetooth enabled device. This method will also create a text file on the SD card attached for the collected data to be written to.

*Loop (void)* – This is the core method to my board’s software. This method will be responsible for pending on a Bluetooth connection then upon receiving a connection, poll the filter to extract the accelerometer and gyroscope readings and store them onto the SD card until a connection is no longer active.

*ConvertRawAcceleration (int aRaw)* – This method is called from the *loop* method to convert the raw accelerometer data into a format that is easier to understand, this will make the data easier to analyse when it comes down to identifying tricks and the patterns they create within the data.

*ConvertRawGyroscope (int aRaw)* – This method will be almost identical to the *convertRawAccerlation* method and is called from the loop method; however the way in which the data is scaled will be different the range value for the gyroscope is different to that of the accelerometer.

1.4 – The Trick Identification System

The trick identification system for the project was coded using JAVA, the software needed to make use of the data collected by the Arduino 101. Using this data to create graphs representing tricks, I would be able define rules for the system to follow this will be discussed later in this section.

The system is able to detect 7 different based on the data collected, the tricks it can detect are as follows:

* Manual
* Nose Manual
* Heelflip
* Kickflip
* Ollie
* Ollie 180
* Pop Shuv – It

1.4.1 – Trick Identification Class Diagram

Below is the class diagram for the trick identification system; this again was a one-class program:



The first two variables in the class diagram assign the size parameters of the multi-dimensional arrays that are used. The variable *cols* is set to 3 as there will always be 3 columns in the arrays to accommodate for the roll, pitch and headings values extracted from the data file.

The *rowCount* variable is assigned a value by the *getFileRowNumbers* function, which will scan through the data file and determine how many rows of data there is. This will ensure the array is large enough to handle all the data values within the file. The *col* and *rowCount* values will be used to initialise all 4 of the multi-dimensional arrays in the class: *data*, *tempA*, *tempB* and *tempC*.

The data file created by the Arduino 101 is scanned using a FileInputStream so the data can be stored in the arrays for the system to analyse. The process of reading this file (the *dataFile* variable in the class diagram) is sped up using the BufferedInputStream (*bis*) and the DataInputStream(*dis*) which gives the system better overall performance. The data file is read using the *loadDataFile* function seen in the class diagram.

All the count variables (*manualCount* through to *popShuvitCount*) that you see in class diagram is how the system keeps track of the tricks performed. These are incremented by the check methods (*checkForManual* through to *checkForPopshuvit*) if the rules for the tricks are satisfied within a given number of readings. These counts are then outputted to the screen to show the user what tricks they performed.

1.4.2 – Looking at the graphs

With the data produced using the Arduino 101 represented as graphs, patterns in the data can be established from which rules can be created to distinguish when tricks are performed in the data. This is the pitch reading (Rotation of the board) for one of the tricks the system should be able to detect called a kickflip:

There is an obvious pattern in this data, the data values rise to above 60 then fall to below -60 before levelling out as the trick finishes. This pattern is what the *checkForKickflip* function follows to identify a kickflip. This analysis proved that trick identification was possible and the same analysis was performed for all seven tricks that the system should be able to identify which will be discussed as part of the implementation in section xxx.

[1] - <https://www.eecs.umich.edu/courses/eecs589/papers/06215496.pdf>

[2] - <https://itunes.apple.com/gb/app/nrf-connect/id1054362403?mt=8>

[3] - <https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp&hl=en_GB>

[2] - <http://www.tatvasoft.com/blog/top-12-software-development-methodologies-and-its-advantages-disadvantages/>

[2] - <https://www.arduino.cc/en/Reference/SPI>

[3] - <https://www.arduino.cc/en/Main/FAQ>

[4] - <https://www.arduino.cc/en/Reference/CurieIMU>

[5] - <https://www.arduino.cc/en/Reference/CurieIMUsetAccelerometerRate>

[6] - <https://www.arduino.cc/en/Reference/CurieIMUsetGyroRate>

[7] - <https://www.arduino.cc/en/Reference/CurieBLE>

[8] - <https://www.arduino.cc/en/Reference/BLEPeripheralConnected>

[9] - <https://en.wikipedia.org/wiki/Freestyle_skateboarding_tricks#Manual>

[10] - <https://www.arduino.cc/en/Reference/BLEPeripheralConstructor>

System Implementation

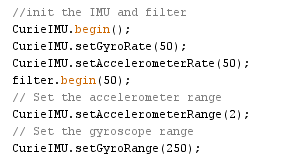
This section will discuss in detail the implementation process for the project. This will cover the implementation for the Arduino 101 software, how the mobile application nrfConnect plays its role in the system and the implementation of the skate trick identification software.

2.1.1- Arduino 101 Software Implementation

The full code listing for the Arduino software can be seen at Appendix D.1.

After declaring the variables as shown in the class diagram for the Arduino software (Section xx), the first part of the implementation was to tackle the setup function, responsible for initialising everything within the software.

The first part of the setup function sets up the motion sensor and allocates the both the accelerometer and gyroscope’s sample rate and range, shown in the code below:



Using the *begin* function starts the IMU (Inertial Measurement Unit). Following on from this the rates for both the gyroscope and accelerometer are set according to rates specified during the design phase (50 samples per second). The filter then starts for later use in the program to update the IMU. The range for the accelerometer and gyroscope is then set according to the values specified during the design phase (2g for the accelerometer and 250®s for the gyroscope).

The next part of the setup function assigns values to the *readingInterval* and *previous* variables used to keep the updating of the filter in sync with the rate in which accelerometer and gyroscope are updating.



The *micros* function returns the amount of time elapsed since the software started running. [1] Assigning the variable *previous* using the *micros* function during the setup function with give it the value of zero as the system has not yet ran. The *previous* variable is then referenced within the loop function to check if it is time to update the filter. This is discussed in more detail later in the section.

The next part of the setup function handles the creation of the file on the SD Card:



The *begin* function [2] is used to specify which pin the SS (Slave Select) line runs from. As shown in the schematic of the Arduino 101 and SD module (fig xx) the SS line runs from pin 10 of the Arduino board, hence the parameter passed to function being 10.

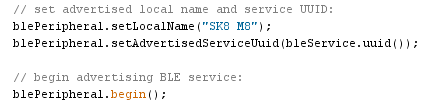
The SD.open [3] function then creates the text file and assigns it to the dataFile variable, written to during the programs operation. The first parameter passed to function gives the file its name and type, the second parameter tells the board we are writing to this file.

The final part of the setup function initialises the *blePeripheral* to allow a Bluetooth connection to be established while the software is running. Below is the code responsible for this:

*Variable Declaration of bleService:*



*Code from setup function:*



This code gives this board a readable name (SK8 M8) as well as a UUID, used when searching for the device using the nrfConnect application so that the user knowns the device to connect to.

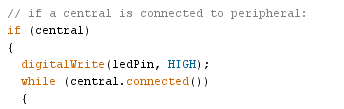
Once the setup function has ran the loop function will start running continuously. Firstly, some local variables are declared to temporarily store the raw accelerometer and gyroscope readings, along with variables to store the converted values into so that the filter can update the IMU.

Also declared a variable called *timeNow*, used in the function to check if it is time to update the IMU, which as its name suggests is assigned the current time on the system when called.

The software then waits for a Bluetooth connection to be established.

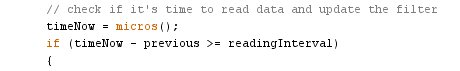


The board will sit idly until a connection is established, when a connection is established a while loop is entered to record data from the motion sensor until the device is disconnected.



Once a connection is established, the on-board LED will turn on showing that the board has accepted the connection. Using *(central.connected())* as the condition for the while loop ensuring that data is only recorded when a Bluetooth connection was active.

The first action in the while loop is to perform a check on whether it is time to record the motion sensor readings. This is done using global variables *readingInterval* and *previous* in conjunction with the local variable *timeNow* (The current system time). These variables create the condition for the following if statement:

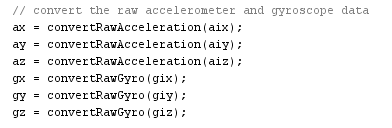


The *timeNow* variable is assigned the value returned by the micros function. As a result, subtracting the value of the *previous* variable (0) and subtracting it from the *readingInterval* variable will determine whether enough time has based in the system for another reading to be taken.

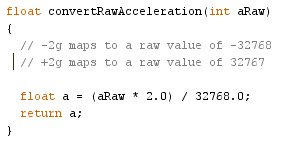
The filter will not update if the result of the variable previous subtracted from timeNow is less than readingInterval. However if the condition is satisfied then the code to record the data from the motion sensor will run.



This line of code reads the motion sensor which produces x, y and z values for both the accelerometer and gyroscope. These are then stored into the local variables of the loop function created to store the raw values from both these sensors. Once the raw data has been collected and stored into the relevant variables they go through the process of being converted into a format that the filter can understand to compute the orientation of the board.

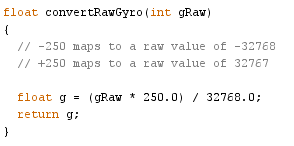


The raw data is assigned to variables local to the loop function, created to store the converted motion sensor data. The converted data is produced using the *covertRawAcceleration* and *covertRawGyro* functions. The function to convert the raw accelerometer data is as follows:

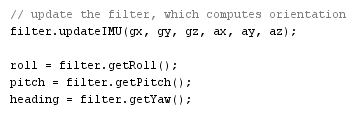


This function will convert the data based on the range set for the accelerometer. As previously mentioned, the range declared in the setup function was 2g. The data passed into the function as shown in figure xxx is multiplied by 2 (as this is the range set for the accelerometer) then divided by 32768 as the range value 2g maps to this numerical value. With the data converted, it is returned as a float value for use when computing the orientation of the board.

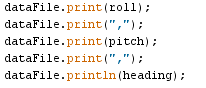
The function to convert the gyroscope manipulates the raw data in exactly the same way multiplying the data by the value of the range (250 for the gyroscope) and dividing this by the mapped numerical value of the range. A range value of 250 for the gyroscope maps to the same value as that of a range of 2g for the accelerometer (32768).



With the data converted, the Madgwick filter can be updated. Once updated the roll, pitch and heading values are extracted ready to be stored onto the attached SD card.



The updateIMU function is called with the converted data values passed into it as parameters using the getRoll, getPitch and getYaw functions to extract the values needed for trick identification. This data is then written to the SD card attached the Arduino board.



This process of collecting the data, converting it and storing it to the SD card module is repeated 50 times per second while a Bluetooth connection is established. Once the mobile device disconnects from the Arduino board the following code is called.



As the data capture process is all contained within the while loop satisfied by a Bluetooth connection the only action required upon disconnection is to close the data file and turn off the LED indicating a device is connected.

2.1.2- The mobile app

The development of the mobile application brought to light a major problem in terms of receiving data over Bluetooth Low Energy from the Arduino 101 board.

2.1.3- Trick Identification Software Implementation

2.1.3.1 – Loading the data file

2.1.3.2 – Trick Identification

2.1.3.2.1 – Manual’s & Nose Manual’s

2.1.3.2.2 – Heelflip’s & Kickflip’s

2.1.3.2.3 – Ollie’s & Ollie 180’s

[1] - <https://www.arduino.cc/en/reference/micros>

[2] - <https://www.arduino.cc/en/Reference/SDbegin>

[3] - <https://www.arduino.cc/en/Reference/SDopen>

3.1 – Testing

When it came down to testing the system and the nature of my methodology being prototype based there was several testing phases that where completed in order to ensure the product was fit for purpose.

Unit testing/ test driven development

3.1.1 - Testing Phase 1

3.1.1.1 - Arduino 101 Software Testing

(collect test data from Arduino board – show how when no device is connected it writes to the file “No device Connected” but when a connection is established )

3.1.1.2 - Trick identification system testing

The first test of the trick identication system occurred a point where the system should be able to detect 4 different tricks:

1. Kickflip
2. Heelflip
3. Pop Shuv – It
4. Manual

Below are the test results of this implementation.

TESTING TABLE FROM TEST LOG FILE

I am happy with the process of detecting both the Manual and Pop Shuv-It Tricks, Pop Shuv-It was always going to be easiest to detect as it is one of the only tricks that a significant change in heading is an indication of when the trick has been performed.

I have now changed the method for identifying both kickflips and heelflips to follow this procedure:

1. Check for when first phase bounds are satisfied (Low Roll & high pitch)
2. If this is satisfied 7 times in a row, then this will change phase 1 to true.
3. If phase 1 is set as true, a variable count will be incremented for every reading after it has been set too true. If the count goes above 40 every is reset as if a kickflip of heelflip had been performed, then it should have go into phase 2 by now.
4. If phase 1 is set to true and the conditions for phase 2 are met by a reading (High Roll & low pitch) and then satisfied 7 times phase 1 is set back to false and phase 2 is set to true if this also happened within 40 readings.
5. Having got the point where phase 2 has now been set to true we can be happy that a kickflip or heelflip has been performed and increment the count for the relevant trick.

3.1.2 – Testing Phase 2

Intro to testing phase 2

With a much better method for detecting the heelflip and kickflip tricks, I am going to run a small sequence of tests to make sure I have a 100% accurate method for detecting them as well making sure the other tricks are still detected as expected. Same 4 tricks being used.

Below are the test results of this implementation.

TESTING TABLE FROM TEST LOG FILE

Happy with how both the heelflips and the kickflips are detected I will now try to rectify the issue where Pop Shuv -It’s not detected when a manual occurs before a Pop Shuv It.

3.1.3 – Testing Phase 3

It was brought to light when investigating the cause of the problem that Pop Shuv-It’s where not being detected when a manual occurred before it was not the case.

It occurred when a manual was at the start of the data set as the heading value of the initial reading was much lower than all the other tricks (All other tricks are around the same) so increasing all the values for the manual heading so they were like all the other files fixed this.

After some slight tweaks to the manual detection function I was satisfied everything was going to work as expected based on previous tests.

With the manual data also manipulated to correlate with the rest of the data I set about performing some more rigours tests.

TESTING TABLE FROM TEST LOG

Satisfied with now the system is currently performing I am going to collect another set of test data that wasn’t used to create the rules for the system and see how well it performs. If it is required (which I am feeling is very likely) then I will have to come up with a new way of iterating through the data as the rules I am 100% satisfied with at this moment in time.