Department of Informatics
Computer Science

Report 2020

Robot Control Interface From Arduino to ARM Mbed

By Priyata Kaneria

Candidate:
168396

Supervisor Ron Grau

Statement of originality

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged. I hereby give permission for a copy of this report to be loaned out to students in future years.

Signed,

Priyata Kaneria

Acknowledgements

Special thanks to Supervisor Ron Grau

Contents

1. Introduction

Arduino is a prevalent platform in its use amongst hobbyists, but not so much amongst professional engineers. Its popularity with hobbyists is largely due to the lack of required knowledge to get started largely owing to its extensive official software libraries and example code, supported by further such code written by the online community. This is why it is a platform that is still widely used to drive plenty of devices and applications.

Arduino is an inexpensive, versatile platform seeing widespread application particularly in an educational context. Unfortunately, the Arduino boards do have certain limitations that make them not quite as preferred by professionals, as alternate controllers that are Arduino-compatible can be more powerful due to their increased RAM, Flash Memory and Clock Speed; widening the applications that can be supported. It is becoming increasingly popular for devices currently running under Arduino to be used within IoT applications, as exemplified by a hackster.io tutorial on using an Arduino MKR ENV Shield with the Arduino IOT Cloud and Amazon Alexa to interact with the board's sensors [1]. However, other boards have better IoT functionality from features including Bluetooth, sdCard and WiFi abilities that are present on platforms such as Mbed. The Mbed solution has more potential for growth in the long-term and is more suited for the future of IoT, providing an excellent framework to develop cloud-connected devices.

One can switch to higher performing Arduino boards such as the Arduino Mega 2650 with 8KB of RAM and 256KB of Flash Memory, but this is still not as high as certain Mbed boards and can have bigger and bulkier Printed Circuit Boards (PCBs), making it difficult to fit them inside certain machines.

As it appears moving to higher functioning boards will be useful for this increasingly IoT-based age, the problem one must consider is how to retire and replace previously used out-dated hardware, and porting the implementations and valuable software solutions that were running on them to a more advanced platform. The motivation of this project is to explore this process.

2. Objectives

The ZUMO robot for the Arduino platform is well-established. Although ports have been created for certain ARM Mbed platforms (e.g. the FRDM-KL25Z) there are currently none specifically for the FRDM-K64F. Arduino UNO R3 is a microcontroller board based on the ATmega328P microcontroller unit (MCU) [2], while the FRDM-K64F contains the power efficient Kinetis K64F MCU [3].

The purpose of this project is to remove the Arduino UNO R3 microcontroller board and translate its library codes (ZumoShield) [4] to create a C++ library that runs on the more powerful ARM Mbed FRDM-K64F microcontroller board. It can then be used to control the ZUMO robot and its components, sensors and actuators (LEDS, Pushbuttons, Motor Drivers, Buzzer, Infrared Reflectance Sensors, Accelerometer, Magnetometer and 3-axis Gyroscope). This report will evaluate the robot's performance and functionality on both the platforms and discuss the process and challenges faced during software porting.

3. Background and Requirements Analysis

3.1 Primary Tasks

3.1.1. Code translation

The Arduino UNO R3 and FRDM-K64F MCUs are both programmed in C++ so the specific translation of code is quite achievable due to the use of the same core language. The point at which the code for each board diverges is when the programmer decides to use certain functionality that is bespoke for either MCU. This occurrence is almost guaranteed due to the communication with the boards being entirely via the pins; these pins having entirely different names. Although the FRDM-K64F's 32 outer pins are compatible with the Arduino UNO R3's pin layout, the pin names are still non-identical, therefore the programmer will first and foremost have to modify the pin referred to in the Arduino code. The second change will most likely be due to the custom function [5] that is called in order to instantiate and initialise the signals with the pin, which again has a different name depending on the board. Subsequently, it's probable there will be further such custom functions that will need to be adjusted by name and translated to match a similar such custom function - instead referred to as APIs - in Mbed [6]. The only feature in this report referred to as an API in Arduino is Wire, a library that comes installed with the IDE, similar to the I2C API in Mbed.

Huang and Runberg explain: "The Arduino language has 60 or so *built-in* (or predefined) functions that make it easier for you to interact with hardware using simple, single-line instructions... Behind the scenes, the digitalWrite() function consists of more than 20 lines of code. Most of that code is complex, but the digitalWrite() function is easy to understand. Even when you understand a big sketch, typing 20 or more lines of code [when] you want to turn an LED on or off is tedious and error prone [7]." To exemplify to comparison, the corresponding custom function to digitalWrite() in Arduino is the API DigitalOut() in Mbed. A more in-depth breakdown of these custom function and API counterparts can be found in *Section 7.1.2.*

In summary, the ZumoShield library will be translated including its header files(.h) and source files(.cpp). The header files are interface-like while the source files provide an implementation of key features of the Zumo robot that are essential for its usage. *Table 1* shows the header and corresponding source files (if any) for the ZumoShield library files that must be ported for use on the FRDM-K64F.

Header files(.h)	**Source files(.cpp)**
L36.h	L3G.cpp
LSM303.h	LSM303.cpp
PololuBuzzer.h	PololuBuzzer.cpp
Pushbutton.h	Pushbutton.cpp
QTRSensors.h	QTRSensors.cpp
ZumoMotors.h	ZumoMotors.cpp
ZumoBuzzer.h	
ZumoReflectanceSensorArray.h	

ZumoShield.h	

Table 1 ZumoShield header and source files

Additionally, the example code within the library will also be translated as they contain control over features that will prove vital for comparing the performance and functionality of the two microcontroller boards. The example code files(.ino) are listed below.

- BorderDetect
- Compass
- LineFollower
- MazeSolver
- PushbuttonExample
- QTRAExample
- QTRARawValuesExample
- QTRRCExample
- QTRRCRawValuesExample
- RCControl
- SensorCalibration
- SumoCollisionDetect
- ZumoBuzzerExample
- ZumoMotorExample

3.1.2. Software Porting Investigation

The main focus on the programming end is to explore the concept of “software porting”; investigating the process of adapting software for use on an alternate platform to the one it was originally designed to execute on. I.e. altering software for it to be usable in different environments. Grover explains: “The method of porting Software comprises receiving a set of code operable in a first operating environment. The method further comprises converting the set of code into a class. The method further comprises providing the converted set of code operable in a second operating environment [8].” Notable solutions, findings and challenges during the process of software porting have been documented and concluded in *Section 7.1 & 8* of this report.

❖ *Language considerations*

Others have already evaluated the process of translating code from an Arduino MCU to an ARM Mbed MCU. Considering their discussions and findings is of significant use for being cognisant of important aspects of software porting before commencing the project. One such investigation includes the porting of software from an Arduino Mega 2560 R3 to Mbed NXP LCP1768 [9]. The author of this s-lab article states that the code is much cleaner working in Mbed, describing code on Arduino to be chunky as most programmers will situate all of their code in one file. The reason for this is due to library creation being notably arduous in Arduino, facilitating a simpler, more C-style of programming (C++’s predecessor) despite Arduino running machine code that has been compiled from the higher language of C++. C has an imperative (structural) paradigm, and this is the level programmers using Arduino are usually limited to, as they end up using more unnecessary computational steps. Ergo, C++ on Mbed is generally cleaner and more modular.

Despite this, the ZumoShield library does not seem affected by the difficult interface of the Arduino IDE. The library takes full advantage of C++’s Object-Oriented paradigm, making it better organised. This also makes it ideal for facilitating the additional functionality of the FRDM-K64F, which uses the same language on Mbed. C++ follows bottom-up programming, supporting a structure focused on hierarchies.

	Arduino Mega 2560 R3	Mbed NXP LPC1768
Price	€35	€46.96
Flash Memory	256KB	512KB
Clock Speed	16MHz	96MHz
RAM	8KB	32KB
EEPROM	4KB	Entire flash memory can be used
Pins	54	40

Table 2 Tech specs [10] (data updated by author to match with current information)

The s-lab article also draws attention to the advantages of the Arduino online community [9]. They highlight that there is an assortment of various implementations for the same endeavoured outcome of a program required on the Arduino Software (IDE) platform, a key advantage of most open source technologies. Unfortunately, they also claim that these code examples generally work but can frequently be poorly coded, contain copious bugs, or be rather unoptimised even if they are found directly on the Arduino website. Alternatively, the online Mbed community is significantly smaller but nevertheless consists of more advanced programmers. He advises that the (now deprecated) Mbed Handbook [12] and Cookbook [13] provided can be of valuable assistance, enabling one to import code directly into the IDE for testing, yet lacking in the sheer selection of code examples potentially required for one's project that are abundant in Arduino. This deficiency can still be counteracted due to the fact that C++ is one of the most utilised languages at this juncture, making it possible to discover one's preferred means of completing a task despite it requiring extra exertion in contrast to more relevant online solutions.

Yiu, author of 'The definitive guide to the ARM Cortex-M0' and staff engineer at ARM Ltd. explicates, "In 8-bit and 16-bit microcontroller programming, the peripherals control is usually handled by programming to registers directly. When using ARM microcontrollers, many microcontroller vendors provide device driver libraries to make use of the microcontroller easier. You can use these library functions to reduce software development time or write to the hardware registers directly if preferred. If you prefer to program the peripherals by accessing the registers directly, it is still beneficial to use the header files in the device driver library as these have all the peripheral registers defined and can save you time preparing and validating the code." [14]

❖ *Programming Environments*

Additionally, one must consider the key similarities and differences in the programming environment for the two microcontrollers as their structure is developed to support the unique functionality of both boards. These environments are provided by Arduino and ARM Mbed, namely the Arduino Software (IDE) supporting any Arduino board [15] and the Mbed OS doing the same for Mbed boards. "The Arduino Uno is programmed using the Arduino Software (IDE), our Integrated Development Environment common to all our boards and running both online and offline" [16]. Once the board is specified both IDE's include built-in support for them, Arduino using the "Boards Manager". This board manager "sets the parameters (e.g. CPU speed and baud rate) used when compiling and uploading sketches; and sets the file and fuse settings used by the burn bootloader command" [17]. The FRDM-K64F is "fully supported in the mbed platform, so it gets access to the free tools and SDK that provides experienced embedded developers with powerful and productive tools for building proof-of-concepts" [5].

The main similarity between the two IDEs, as mentioned earlier, is the almost mirrored available custom functions and APIs provided, with equally similar names that make the process of software porting smoother. A key aim is to be aware of the native capabilities of both platforms and their pre-defined functions in order to focus on the best

way to execute the desired functionality instead of focusing on the manner in which it was originally coded. This proves especially important for the author in terms of time saving, as explained in the upcoming *Section 8*. The author became too concerned with the minor differences in the pre-defined functions, focusing on how to exactly replicate them instead of looking at what was already provided in terms of functionality on the Mbed platform instead.

The Mbed platform contains unique features that enable it to be a basis for low-power systems, such as the RTOS library that allows one to run multiple independent threads making the code design much simpler. Rather than having one big main loop that handles all the implementation, the code for separate tasks can be split into threads. The Mbed OS also has Cloud capabilities that are already integrated, allowing one to seamlessly transfer data from the device to the cloud and aggregate it using one of ARM's cloud partners. It maintains a TCP/IP stack and Socket API on top of multiple transports (Ethernet, WiFi and 3G) [18] and a scheduler for ensuring tasks get their appointed run-time. It also includes a hypervisor, namely the PSA-compliant SPM, to separate different tasks, creating and managing independent secure partitions on Arm Cortex®-M microcontrollers so that user-tasks cannot compromise the security of the complete system. Mbed explains, "It increases resilience against malware", because if someone breaks into a complicated RF stack on one's system, they will find it difficult to gain access to the rest of the system. They continue, "…[it also] protects secrets from leaking between different modules in the same application [19]." This capability also opens up the possibility of running user generated or insecure code on the devices, as the SPM will ensure that it only takes granted resources and cannot access secure information such as cryptographic keys. These features are critical for IoT applications.

Arduino does not contain a debugger for checking scripts. On one hand this is suitable for inexperienced developers but on the other an experienced developer would much rather have a debugger; this missing feature is often an impediment to Arduino being taken seriously. Similarly, the Mbed OS online IDE also has this feature missing, but alternatives lie in the other desktop IDEs available such as the ARM Mbed CLI (a Python-based tool) and ARM Mbed Studio (a fairly new desktop IDE that was still in public beta mode at the start of 2019, taking feedback from alpha testers). For this reason, the author programmed this project using Mbed Studio, keeping in mind that there may be a few bugs due to its recent release.

The s-lab article [9] determines that, albeit the simplicity of the Arduino Software (IDE) on the basis of its handful of features, it is dependable and well-established. They instead liken the Mbed OS online IDE to a platform that is in a somewhat more beta stage of development rather than a finalised product. This is demonstrated through instances such as a high potential for crashes of the IDE to occur if the internet connection is lost, and goes on to propose that perhaps it is worth taking a look at alternatives to the online IDE to overcome this issue.

Nevertheless, they point out a crucial component of the Mbed OS online IDE which is the ease it provides in library creation and Object-Oriented practices. This is efficacious in enabling a team of programmers to work collectively on the same project, as each team member can work on their own module in their separate set of files. This is in contrast to Arduino which, as mentioned above, makes it rather challenging to create a library as it does not encourage exploiting convenient Object-Oriented programming practices. This is because the IDE does not let the programmer open any library files such as .h or .cpp files that are integral to C++ library programming, and only lets the user open example "sketches" in .ino format. Again, despite these limitations with the IDE, the ZumoShield library has multiple instances of Object-Oriented programming. This may have been achieved by the library having been written on another platform entirely and then transported back and forth between the Arduino IDE for testing. This confirms that the Arduino IDE does not accommodate projects that may be undertaken by more professional developers. The s-lab article concludes that the Arduino Software (IDE) is markedly more solid in regards to reliability, while the Mbed OS online IDE is imperative for large projects with prodigious amounts of code and multiple contributors.

Lim presents another perspective that shares a likeness to the one above, expounding that the Mbed online IDE lacks straight-forwardness and is laborious when attempting to determine “how the pins are defined for each board” [20]. Likewise he disapproves of the platform’s dependency on an internet connection, stating, “…I don't want to necessarily depend on it to compile my code.” He goes on to suggest PlatformIO as an offline option, stating, “It is also less cumbersome than the mbed drag-and-drop method.” The Mbed online IDE uses a custom boot loader and appears as a USB flash drive when plugged into the computer via the USB port. One must then compile their code using the online IDE, download the .hex file and copy it to the Mbed, resulting in time consumption. Both of the above viewpoints can be solved by using ARM Mbed CLI or ARM Mbed Studio.

Styger’s experiment is one closely related to the one carried out in this project. He programmed the FRDM- KL25Z to be usable with the Zumo robot instead of an Arduino board in the CodeWarrior IDE using Processor Expert [21]. NXP, creators of Freedom boards state, “Processor Expert® technology is a development system to create, configure, optimize, migrate, and deliver software components for our silicon. Processor Expert technology makes it much easier for [one] to deal with the low level intricacies of a hardware platform in an optimal manner... [One] design[s] custom peripheral drivers ideally suited to [their] needs, without having to know everything about the hardware” [22]. However, Processor Expert has been replaced by its successor, the MCUXpresso Configuration Tools, since Styger published his article in 2013. The NXP official Getting Started with the FRDM-K64F guide lists the MCUXpresso Software Development Kit (SDK) + IDE and Zephyr™ OS as recommended development paths, of course accompanied by ARM Mbed [23].

Finally, one gets no real experience of professional development tools with Arduino. If one starts their journey of micro-controllers with Arduino then it will becomes increasingly difficult to make complex intelligent circuitries in the future. Porting over to FRDM-K64F is a useful learning process for any engineer if they would like a deeper understanding of circuitry.

❖ *Hardware*

It is important to consider the pin configurations which differ marginally between the microcontroller boards. The pinout diagrams show the layout of the pins (which make up ports); the Arduino UNO R3 containing 32 pins shown in *Figure 1* and FRDM-K64F containing 64 shown in *Figure 2*. The pins on each board are laid out in a similar manner particularly with the 32 outer pins of the FRDM-K64F, due to it containing, “Form-factor compatible with the Arduino® UNO Rev3 pin layout” [24]; which greatly assists in understanding the parallels between the pin signals in order to match and adapt functionality from one board’s pins to another: easing the process of software porting. The crucial distinction in regards to the hardware is the additional functionality provided by the 32 extra inner pins of the FRDM-K64F. That being said, this does not have any implications on the software porting itself, but more so on the optional requirements set out in this report such as testing the difference in the functionality of the Zumo robot between the two boards by writing further code that utilises the potential extra functionality from the additional pins. *Figure 3* illustrates the mapping between Arduino pins and ATmega328P ports [2]. To elucidate, these board pins correspond to the component leads extending from each of the four sides of the ATmega328P, depicted in the pinout diagram in *Figure 4*. On the other hand, “The freedom board headers enable up to 64-pins and give access to most of the Kinetis K64F signals. Outer row pins deliver right signals to meet Arduino R3 standard, the inner row is connected to up to 32 additional Kinetis K64F pins” [3]. *Figure 5* indicates the Kinetis K64F microcontroller signal connections with the board components (RGB LED, Motion Sensors) and extension sensors (SD Card, Bluetooth, WiFi); specifying the pin names required in order to communicate with them. The named pins in this diagram correspond to just a few pins of the 144 total pins in the Kinetis K64F MCU, presented in *Figure 6*.

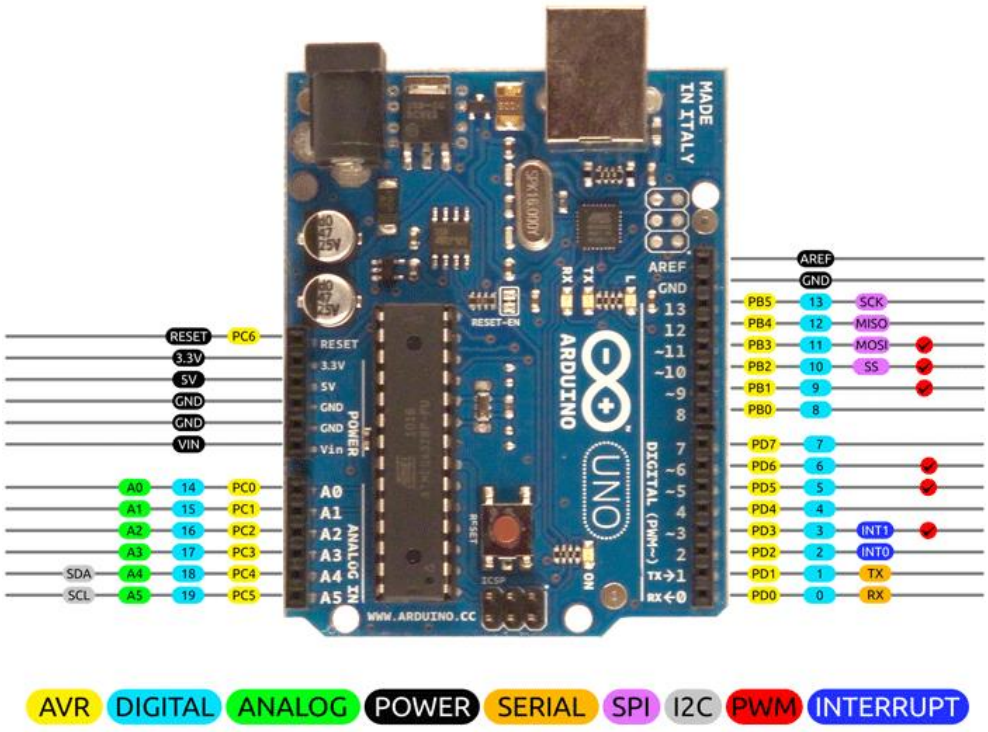

Figure 1 Arduino Uno Pin Diagram [25]

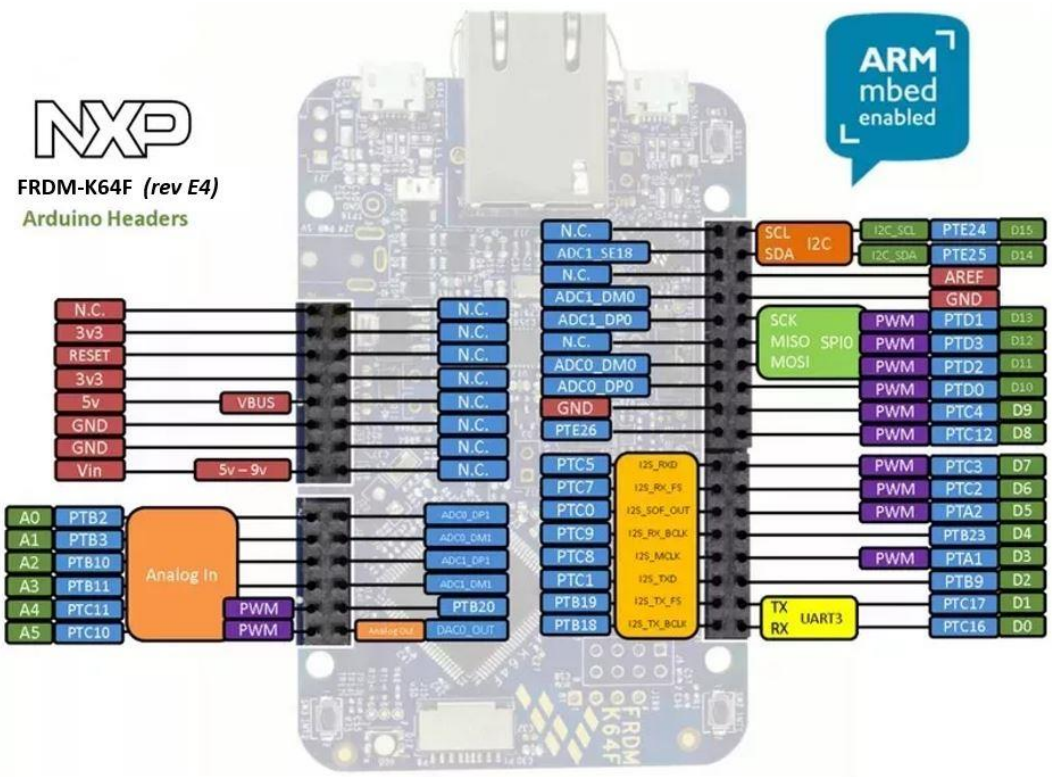

Figure 2 Arduino and NXP Header Pinout [3]

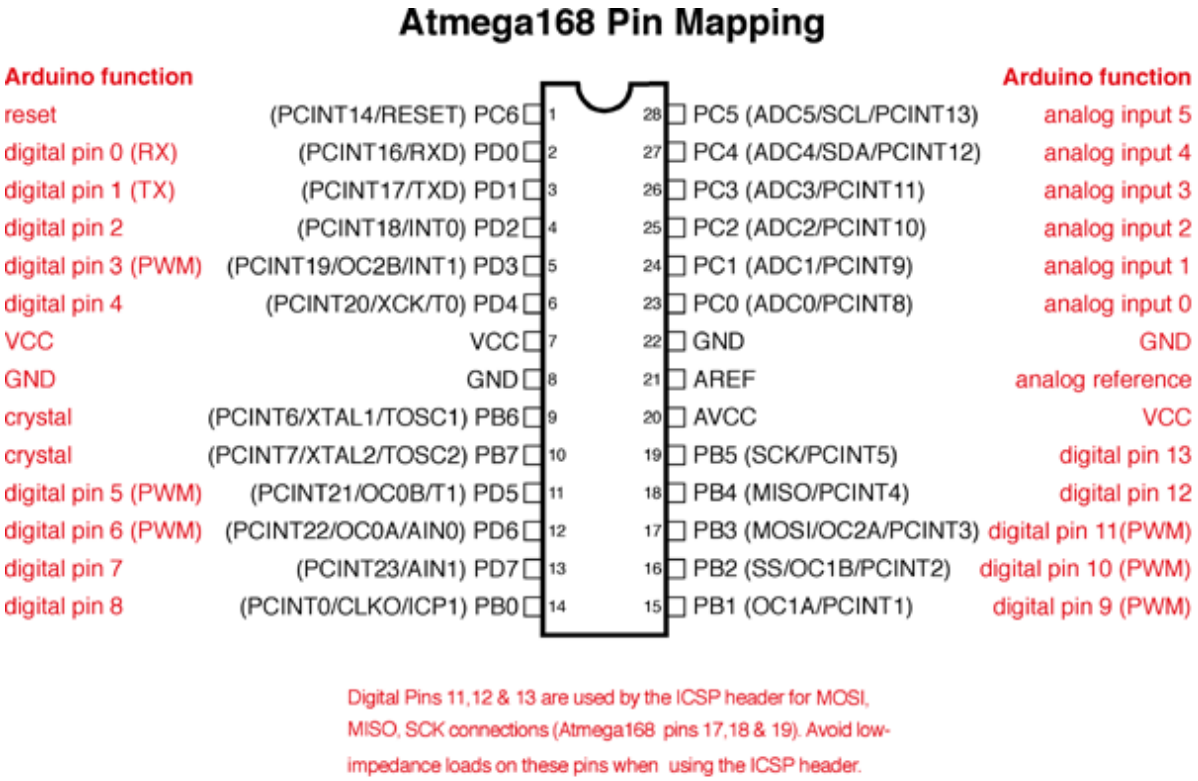

Figure 3 ATmega328P - Arduino Pin Mapping [26]

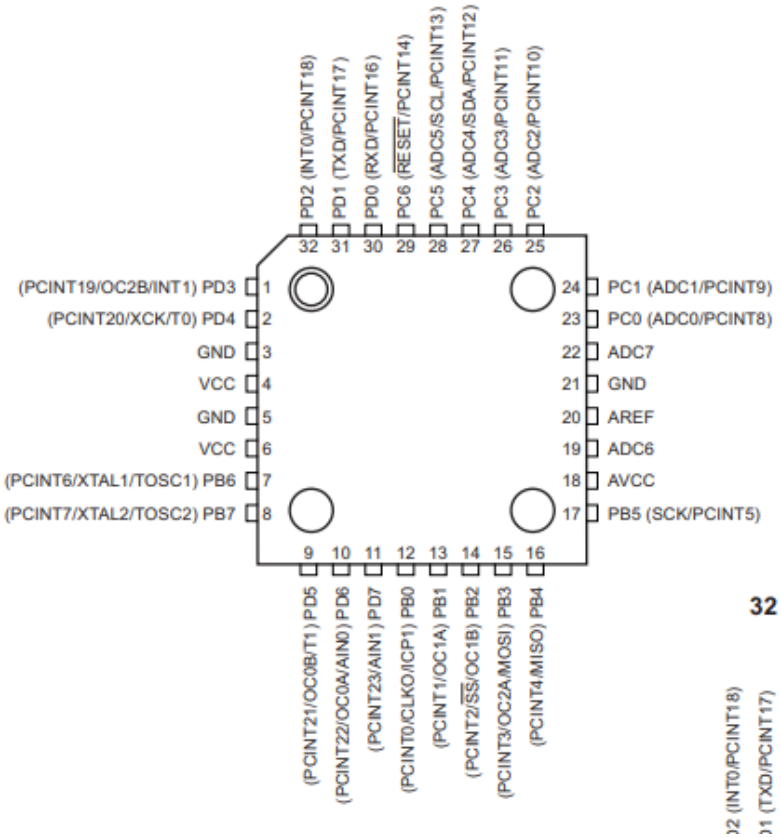

Figure 4 ATmega328P Pinout (Figure1-1 [27])

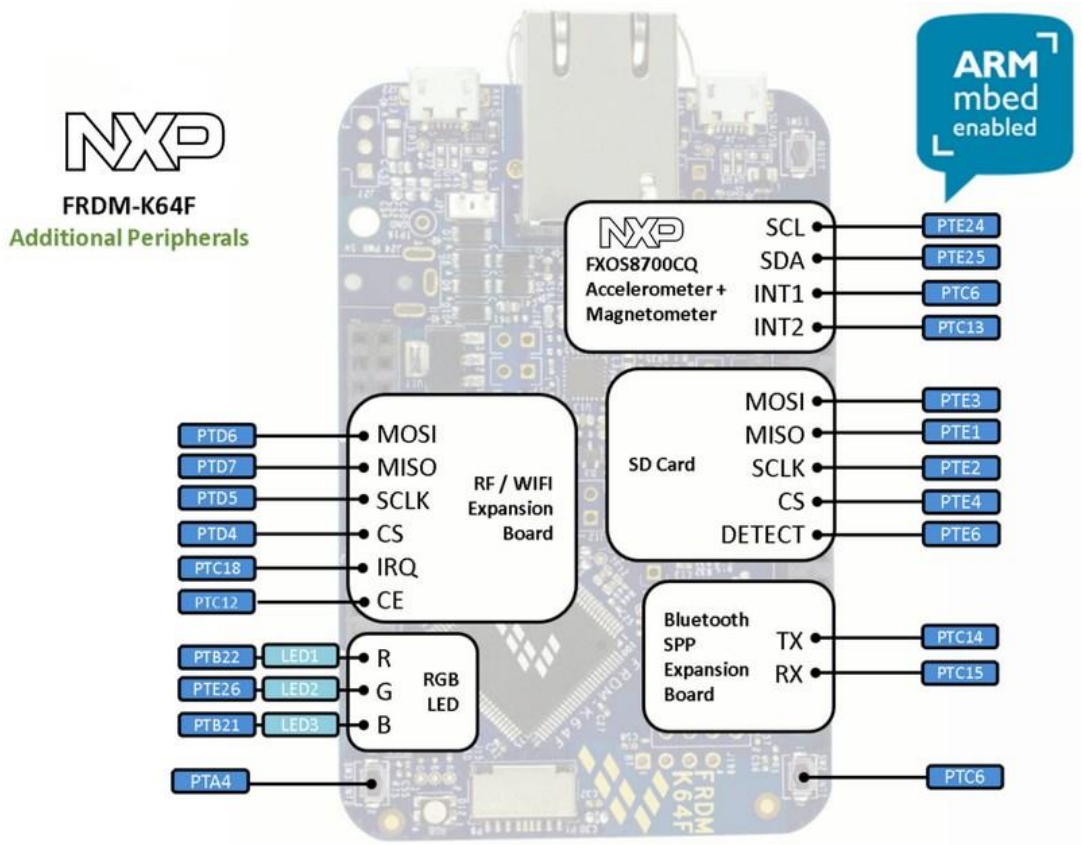

Figure 5 FRDM-K64F Additional Peripherals [3]

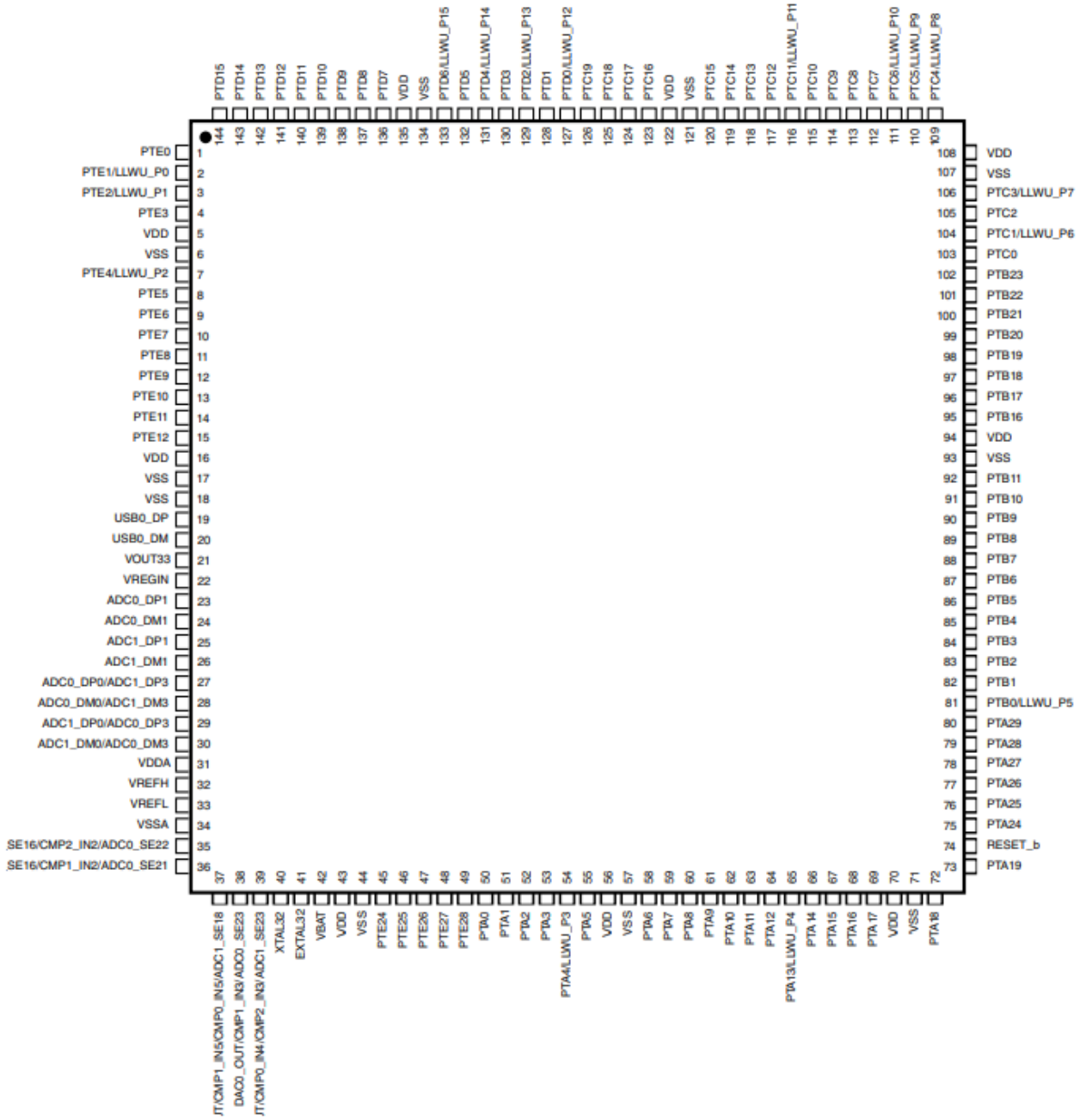

Figure 6 144 LQFP K64 Pinout Diagram (Figure 37 [28])

During Styger's process, he took note of strategic tactics he could implement to make the debugging process more convenient, advising, "Generate bread crumbs for debugging: used the command line shell built-in, transmitted messages over Bluetooth, or using an extended USB cable". For this maze experiment he also warns to, "Never underestimate problems with sensors. Depending on the laser printer and paper used, the reflections were different…", causing misperception on the cause of the occasional sensor unreliability [29]. This was also due to the use of laser printing to create the black lines for the Zumo robot to follow instead of black tape which could hypothetically cause less reflection.

The author will note the general, Arduino or Mbed specific, and Arduino UNO R3 and FRDM-K64F microcontroller board specific challenges and significant phases of the process of software porting in *Section 8*.

3.1.3. Microcontroller board performance comparison

Table 3 indicates the technological specifications for the two boards, the data indicating that the FRDM-K64F would yield faster results and be generally more powerful with potentially superior functionality. The assumption is that the increased RAM, Flash Memory, and Clock Speed will cause the code to run without crashing, giving better accuracy of data and more control of the robot. The final requirement of the project will be to investigate and test the boards to see if these assumptions hold true.

	Arduino UNO R3	FRDM-K64F
Price	€20	€31.64
Flash Memory	32KB	1024KB
Clock Speed	16MHz	120MHz
RAM	2KB	256KB
EEPROM	1KB	Entire flash memory can be used
Pins	28	32 (64 including inner and outer)

Table 3 Tech Specs for Arduino Uno R3 & FRDM-K64F

A list is presented below depicting examples of types of experiments that will be carried out in order to fulfil the investigation.

Concepts evaluated when comparing boards:

- Timing
  - ⇒ Compile time - indicates which IDE or compiler is more efficient for executing software and therefore which board might be faster due to its IDE
  - ⇒ Time Complexity of programs on either board – computational steps
  - ⇒ Robot response time
- Functionality – the variety of useful functions that can be implemented and directly compared for each board
  - ⇒ FRDM-K64F's 6-axis combo Sensor Accelerometer and Magnetometer [30] vs. Arduino UNO R3 with just these sensors provided on Zumo
    - ➢ This will be achieved by directly comparing the SumoCollisionDetect example code on each board. The author bought a second Zumo robot in order to accurately compare the difference in their performance.
  - ⇒ Whether the additional inner 32 pins on the FRDM-K64F board can enhance functions performed by the Zumo robot in contrast to the Arduino board
- Comparing the quality of execution of the sensors that come with ZUMO incl. [31]
  - ⇒ Reflectance sensor array (infrared QTR sensors) e.g:
    - ➢ to test how accurately it can follow a line [32]
    - ➢ to test which board performed better at border detection
    - ➢ using the compass example to compare the performance of the robot coordinating 90 degree turns and driving in squares

3.2 Secondary Tasks

Objectives that would appear further than the time limit include:

- Creating original functions to portray functionality that is unique to the FRDM-K64F board in order to document what abilities the Zumo robot is capable of on this board in contrast to Arduino UNO R3
    - ⇒ This includes the use of FRDM-K64F's RGB LED, SD Card, Bluetooth, Wifi that is not available on the Ardunio UNO R3
    - ⇒ The additional 32 inner pins on the FRDM-K64F have connections to these components. Also valuable to test what other signals and capabilities these pins uniquely provide to the Zumo robot
- Connecting an RC servo and receiver to both boards and turning the Zumo into a radio-controlled vehicle, again comparing the performance

3.3 Software Requirements Summary

A summary of the mandatory and optional requirements of the software implementation will now be presented below as a result of the above requirements analysis.

Mandatory Requirements	**Justification**
Translate L36.h and L36.cpp from Arduino for use on FRDM-K64F	Interfaces with and enables reading of raw data from 3-axis gyros
Translate LSM303.h and LSM303.cpp from Arduino for use on FRDM-K64F	Interfaces with and implements compass and accelerometer
Translate PololuBuzzer.h and PololuBuzzer.cpp from Arduino for use on FRDM-K64F	Enables the buzzer
Translate Pushbutton.h and Pushbutton.cpp from Arduino for use on FRDM-K64F	Enables the Pushbutton
Translate QTRSensors.h and QTRSensors.cpp from Arduino for use on FRDM-K64F	Implements the reflectance sensors
Translate ZumoMotors.h and ZumoMotors.cpp from Arduino for use on FRDM-K64F	Implements motor control
Translate ZumoBuzzer.h from Arduino for use on FRDM-K64F	A trivial subclass of PololuBuzzer
Translate ZumoReflectanceSensorArray.h from Arduino for use on FRDM-K64F	Provides an interface for using a Zumo Reflectance Sensor Array and provides access to the raw sensor values as well as to high level functions including calibration and line-tracking
Translate ZumoShield.h from Arduino for use on FRDM-K64F	A subclass of many of the above
Translate example code BorderDetect from Arduino for use on FRDM-K64F	Example code for detecting a white border on a black surface
Translate example code Compass from Arduino for use on FRDM-K64F	Example code for making use of the compass features
Translate example code LineFollower from Arduino for use on FRDM-K64F	Example program that demonstrates how a Zumo can be programmed to follow lines with a reflectance sensor array and run a line-following course.
Translate example code MazeSolver from Arduino for use on FRDM-K64F	Example code for implementing the solving of a maze
Translate example code PushbuttonExample from Arduino for use on FRDM-K64F	Example code for implementing the use of the pushbutton
Translate example code SensorCalibration from Arduino for use on FRDM-K64F	This example is in order to calibrate the Reflectance Sensor Array with its surroundings
Translate example code SumoCollisionDetect from Arduino for use on FRDM-K64F	Example code to test the robot's collision detection
Translate example code ZumoBuzzerExample, ZumoBuzzerExample2 and ZumoBuzzerExample3 from Arduino for use	Example code in order to test the buzzer

on FRDM-K64F	
Translate example code ZumoMotorExample from Arduino for use on FRDM-K64F	Example code in order to test the motor and drive the robot

Table 4 Requirements summary of primary objectives

Optional Requirements
Translate example code QTRAExample, QTRARawValuesExample, QTRRCExample, and QTRRCRawValuesExample from Arduino for use on FRDM-K64F
Creating original examples that only support FRDM-K64F specific functionality e.g. RGB LED, SD Cards, Bluetooth, WiFi, and other FRDM-K64F pin signals
Translate example code RCControl from Arduino for use on FRDM-K64F while connecting an RC servo receiver to Arduino UNO R3 and FRDM-K64F

Table 5 Requirements summary of extension objectives

4. Professional Considerations and Ethics

This project meets all the specified ethical requirements outlined in the Ethical Compliance Form attached in the appendix, as above all it will not involve any other members of the public save for this report's author and supervisor Ron Grau. The health, privacy, security and wellbeing of the two of us and the environment will not be impacted by the investigations on the Zumo robot as it has a negligible and inconsequential effect on its surroundings. This is due to the fact that it does not have access to or collect personal data from either of us, nor does it have sharp enough edges or a substantial power supply to cause physical harm or injury as "the Arduino's regulated 5V and 3.3V voltages supply power to the motor driver logic, buzzer circuit, and compass module on the Zumo Shield" [31]. The experiments will be conducted in a contained environment comprising of a small black surface such as a Dohyo (a wooden battle arena for Robot Sumo) portrayed in *Figure 7* below or white sheets of paper as shown in *Figure 8*. As a result neither discrimination nor malpractice against Third Parties can occur, participants will not be exposed to any risks greater than those encountered in their normal working life and all relevant legislative and regulatory requirements will be met.

Figure 7 A Zumo Robot in a test arena for evaluating movement and IR sensors [31]

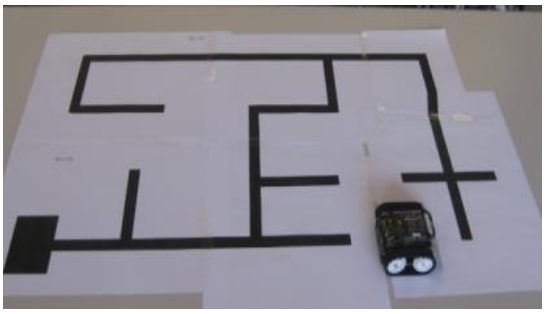

Figure 8 Maze for Zumo Robot [29]

The study materials comprise of software running on standard hardware, as indicated via the use of the the well-established ZumoShield robot, Arduino UNO R3 and FRDM-K64F boards. The work undertaken in this project is within the professional competence of the author as it utilises an array of expertise provided by the author's Computer Science undergraduate degree, including the training of C++ during their foundation year. The author's professional knowledge will continue to be developed for the duration of the project due to the length of the degree course. The author will also make it a priority to independently find answers to fill gaps in their knowledge and will ensure any further research and references provided in the final report are up to date. Furthermore, the value of the extensive insight possessed by the author's supervisor will also be an aid in maintaining awareness of any technological updates, developments or standards that may be relevant and important for the investigation. The author will also have the opportunity to obtain alternate viewpoints and criticism from the supervisor which will be respected and always taken into account. Conclusively having discussed this in length with given supervisor, it has been agreed that no ethical review or ethical compliance form will be required.

5. Project Management

5.1 Trello

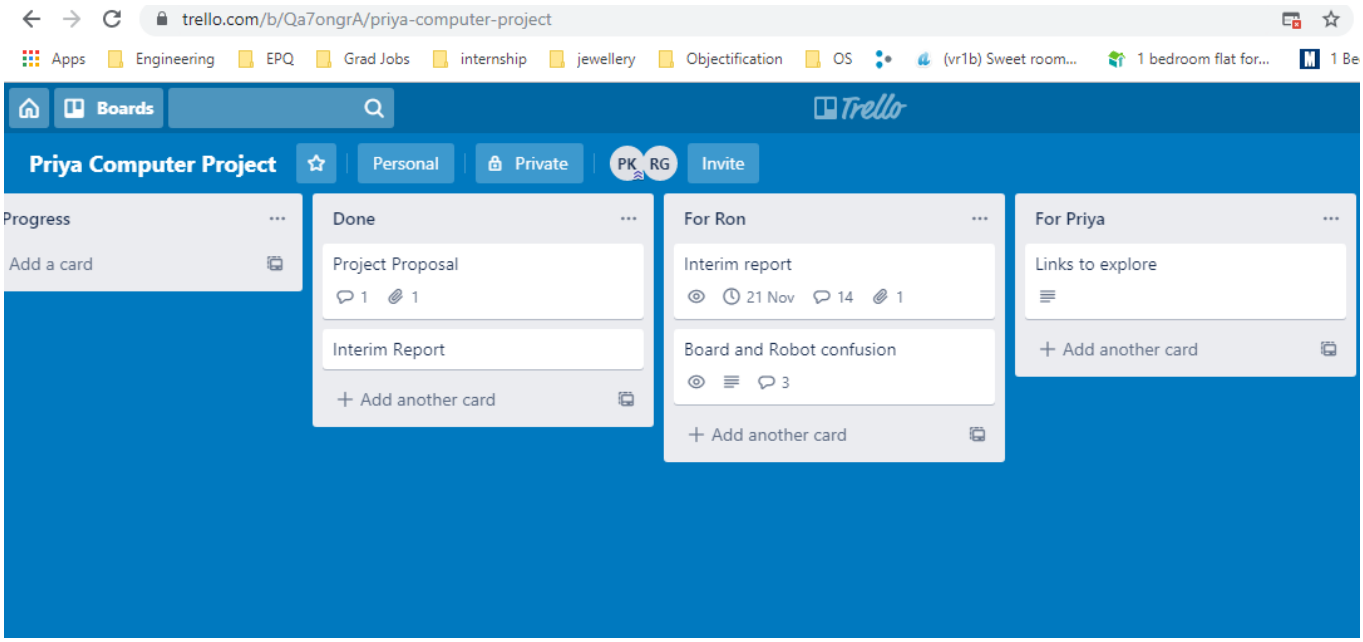

Figure 9 Trello board, the online project organiser

The author has been communicating and planning the project with their supervisor on the Trello platform and will continue to do so for the entirety of the project. It is a useful platform which consists of a board, as depicted in *Figure 9*, in order to organise all upcoming tasks in laid out sections and share completed work along while being able to present any questions for the supervisor or clarifications required from the supervisor. Reminders are also a feature available on this platform which aids in keeping up with deadlines.

6. System Design and Development

6.1. Initial Designs

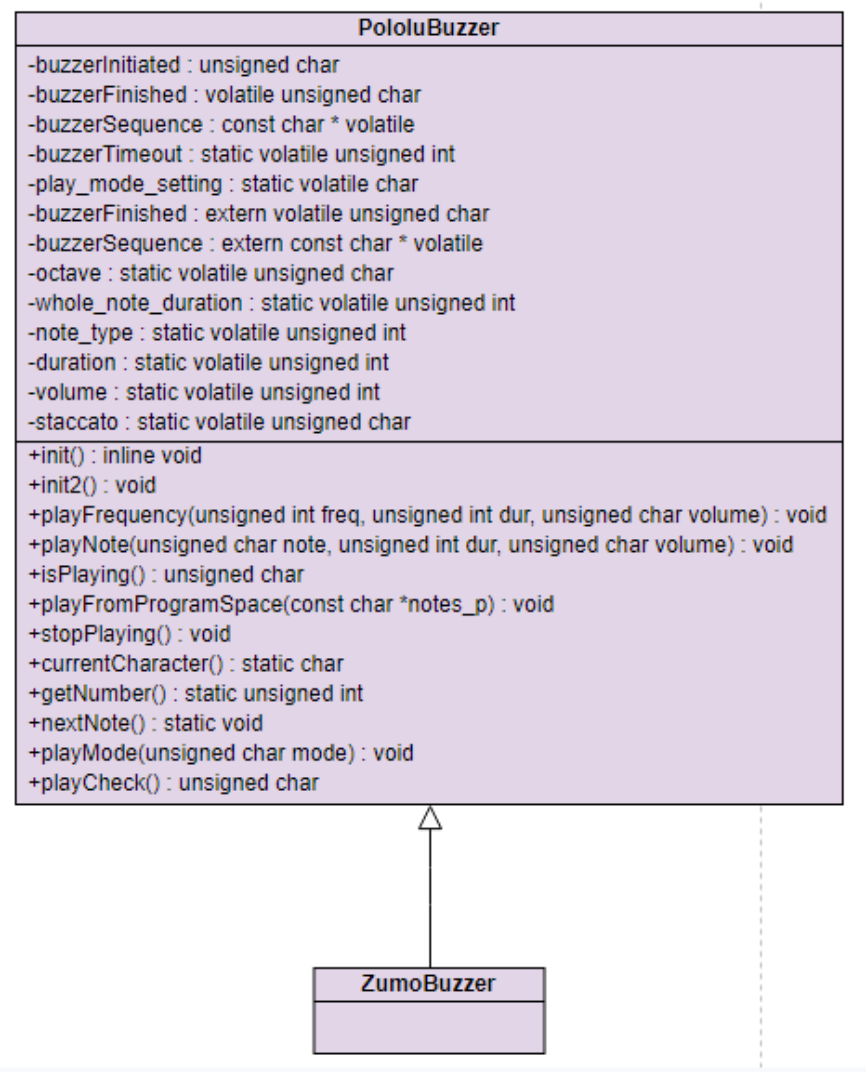

Figure 10 UML diagram showing ZumoBuzzer as the subclass of PololuBuzzer

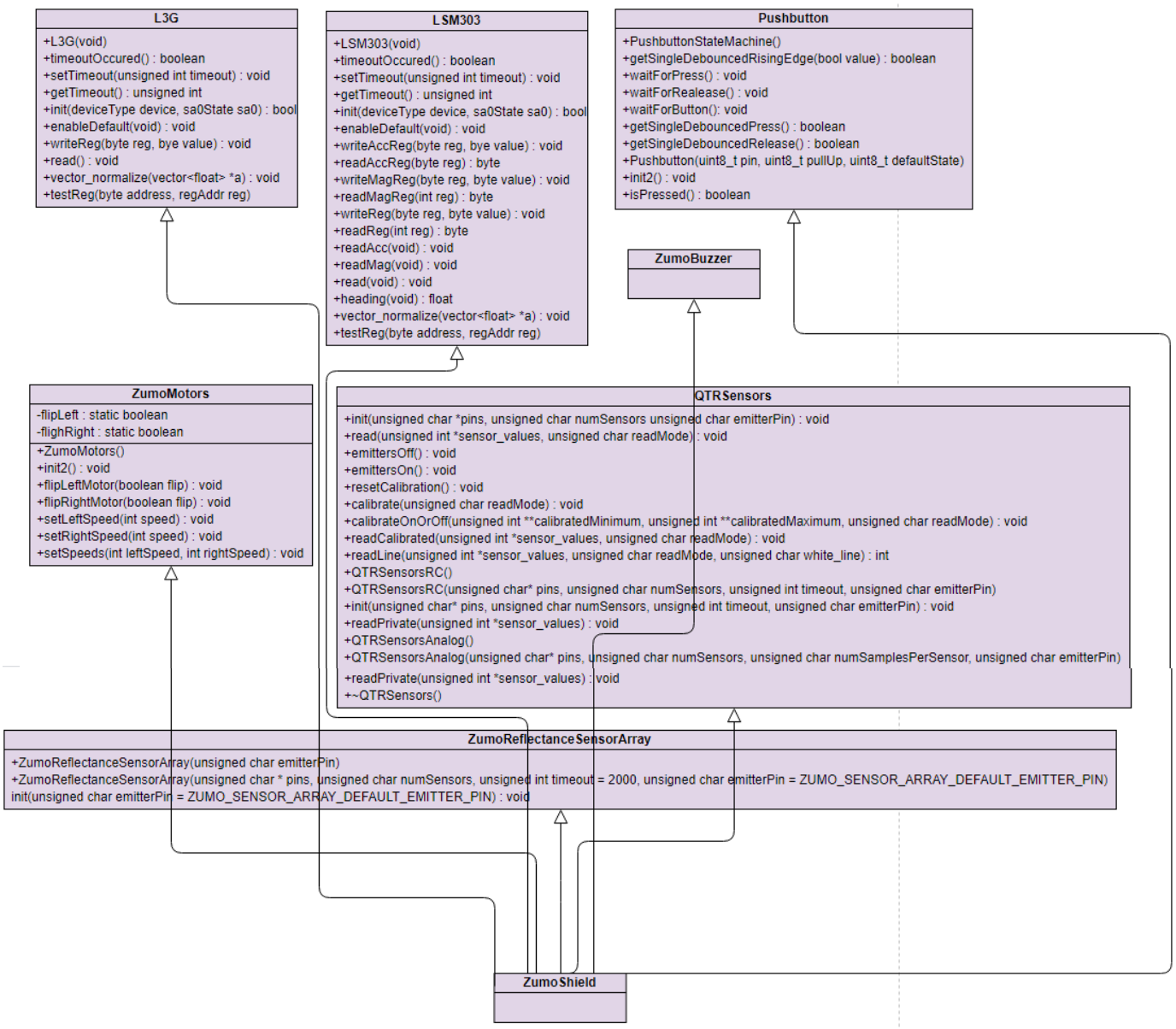

Figure 11 UML diagram showing the inheritance of ZumoShield

7. System Building

The use of class instantiations within each example goes as follows:

- PushbuttonExample: i) Pushbutton
- ZumoBuzzerExample: i) Pushbutton
  ii) ZumoBuzzer
- ZumoMotorExample: iii) ZumoMotors
- BorderDetect: i) Pushbutton
  ii) ZumoBuzzer
  iii) ZumoMotors
  iv) QTRSensors
  v) ZumoReflectanceSensorArray
- LineFollower: i) Pushbutton
  ii) ZumoBuzzer
  iii) ZumoMotors
  iv) QTRSensors
  v) ZumoReflectanceSensorArray
- SumoCollisionDetect: i) Pushbutton
  ii) ZumoBuzzer
  iii) ZumoMotors
  iv) QTRSensors
  v) ZumoReflectanceSensorArray
  vi) LM303

This therefore encouraged the system requirements to be built in this order:

1. ZumoShield.h
2. ZumoBuzzer.h
3. Pushbutton.h
4. millis.h
5. millis.cpp
6. Pushbutton.cpp
7. PushbuttonExample.cpp
8. PololuBuzzer.h
9. PololuBuzzer.cpp
10. ZumoBuzzerExample.cpp
11. ZumoMotors.h
12. ZumoMotors.cpp
13. ZumoMotorExample.cpp
14. QTRSensors.h
15. QTRSensors.cpp
16. ZumoReflectanceSensorArray.
17. BorderDetect.cpp
18. LineFollower.cpp
19. MazeSolver.cpp
20. LM303.h
21. LM303.cpp
22. L3G.h
23. L3G.cpp
24. SumoCollisionDetect.cpp
25. SensorCalibration.cpp
26. compass.cpp

This aided in building the system step-by-step, coding it in a way that ensured that when building a new piece of example code, the other included classes required to run this code correctly were already developed beforehand. The L3G gyroscope class was not included in any example code of the ZumoShield library, so the ordering for this requirement was arbitrary.

The author had originally planned on building the system in the Mbed OS online IDE. However, the Mbed Studio proved to be of better use in regards to its offline coding capabilities. Moreover, the compiled file does not need to be uploaded using drag and drop but instead through the IDE itself, proving to be more efficient as suggested in the background research. The author also preferred the layout and graphics in terms of managing the system, as the Mbed Studio has a more professional appearance. In addition,

this pairs with the desktop IDE selected for Arduino, the Arduino Software (IDE), which could also have been its online counterpart, Arduino Web Editor.

7.1 *Code translation*

The process of building the Mbed system started off by opening the Arduino library files inside the Mbed studio, the primary action being to add an include for the “mbed.h” file. The next procedure was to create an int main() function and place a call to the Arduino example’s setup() and loop() functions, situating the latter in an infinite while loop. The Arduino compiler holds a call to these two functions behind the scenes but in Mbed these calls must be stated. Accordingly, it was clear what the distinctions between the two compilers were simply due to the visible errors, leaving the rest of the code reusable.

7.1.1 Pin Mapping

In order to alleviate the struggles faced due to the complications caused by porting software between two microcontroller platforms, the primary task is to identify and understand the corresponding pin signals between the two boards. Modifying the pin names referred to would correct one of the first errors evident in any example. In regards to the corresponding signals between the Arduino Uno R3 and the FRDM-K64F microcontrollers, a clear mapping has hitherto not been available online. As a result, the author formed *Figure 12* below to make this abundantly clear. The diagram was based on the official FRDM-K64F Rev 3 pin multiplexing spreadsheet [33] provided on the Mbed website [3], plotting the mirrored signals in relation to their equivalent locations. This illustration is a vital stage in the process of software porting, proving to be a valuable aid in building the system.

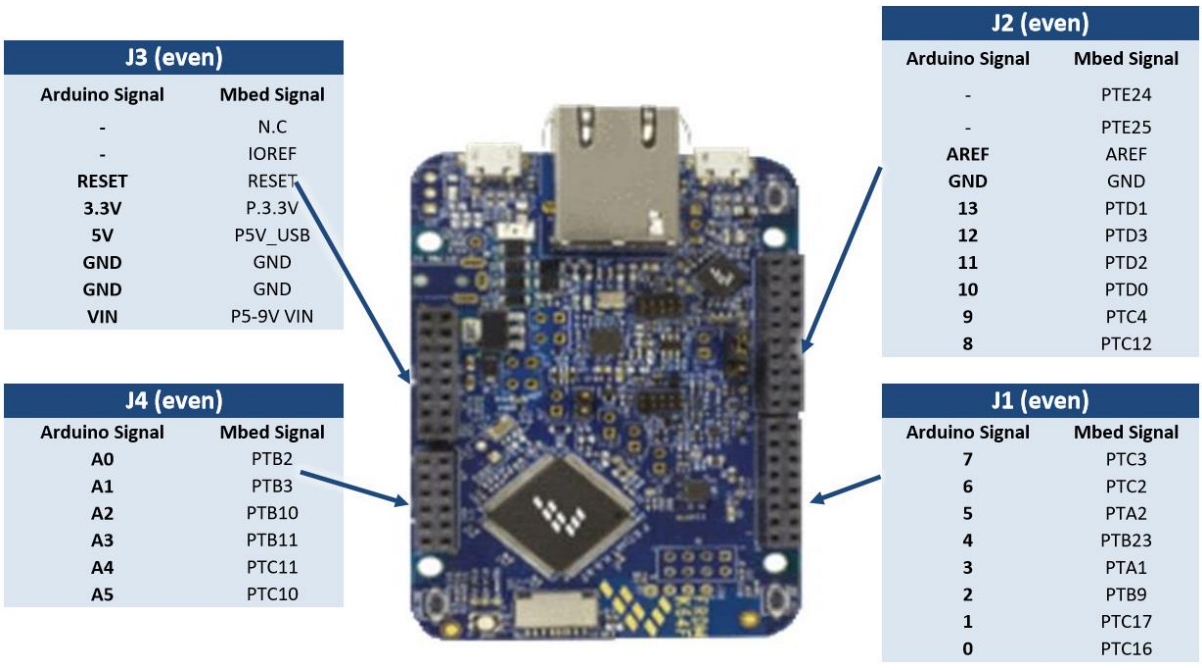

Figure 12 Arduino R3 to FRDM-K64F pin mapping

7.1.2 Name Counterparts

Many of the remaining errors could be corrected by replacing pre-defined Arduino functions with, as explained in previous chapters, an equivalent Mbed counterpart – a pre-defined API. A major technical difficulty was fine-tuning the code to exactly replicate the Arduino functionality, as even the almost identical Mbed counterparts do not necessarily have the exact same capabilities or resulting values. The online Mbed forums were of great assistance when attempting to amend this, facilitating in mirroring the code accurately. Consequently, following the completion of each example code, a great deal of testing and adjusting of code was implemented. A struggle for the author was to grasp when this was significant in terms of reaching the desired outcome or whether the resultant behaviour would work effectively regardless, as an exact imitation was not always necessary. For example, the Arduino function millis(), "returns the number of milliseconds passed since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days [34]." In order to mirror this behaviour in Mbed one must utilise the Timer API, as depicted below in *Table 6*. To replicate the time being recorded since the board started running the current program, this is best achieved by placing a timer.start() as the first instruction in your Mbed int main() function. However, the author misplaced their focus on duplicating the timer overflowing after 50 days. This was a mistaken priority as it was uncovered that millis() would simply be used to record short periods of time in the scope of the ZumoShield library, thus resetting the value after 50 days would not be required.

A clear list of discrepancies between the two IDEs that one should be aware of in order to ease the process of going from Arduino to Mbed is listed below in *Table 6 and 7*.

Arduino	**Mbed**	**Functionality**
pinMode(pin, INPUT);	DigitalIn myled(pin, mode)	Configures the specified pin to behave as a digital input.
analogRead(pin)	analogIn(pin);	Reads the external voltage from the specified analog pin.
digitalRead(pin)	myled.read();	Reads the value of a specified digital input pin
pinMode(pin, OUTPUT);	DigitalOut myled(pin, value)	Configures the specified pin to behave as a digital output.
analogWrite(pin)	PwmOut(pin);	Writes an analog value (PWM signal) to a pin
DigitalWrite(pin, value);	myled.write(value)	Writing a value to a digital output pin, setting its state
#include <Wire.h>	I2C i2c(I2C_SDA, I2C_SCL)	Provides I2C functionality and is used to communicate with I2C (Inter-Integrated Circuit) devices such as serial memories, sensors and other modules or integrated circuits.
Wire.begin(address);	i2c.start();	Initiates and creates a start condition on the I2C bus

Wire.requestFrom(address, quantity, stop);
While(Wire.available())
{
Wire.read();
}
	i2c.read(address, *data, length, repeated=false);	Reads from an I2C slave. Used by the master to request bytes from a slave device. Performs a complete read transaction.

Wire.beginTransmission(address);	i2c.write();	Begins a transmission to

Wire.write(value);		the I2C slave device. Subsequently writes a single byte out on the I2C bus from a slave device in response to a request from a master.
Wire.beginTransmission(address); Wire.write(data, length);	i2c.write(address, *data, length, repeated=false);	Begins a transmission to the I2C slave device with the given address. Subsequently writes an array of bytes to an I2C slave and queues bytes for transmission from a master to slave device.
Wire.endTransmission();	i2c.stop();	Ends a transmission to a slave device and creates a stop condition on the I2C bus.
Serial.begin(speed);	static BufferedSerial serial_port(USBTX, USBRX, baud);	Creates a serial port and sets the data rate in bits per second (baud) for serial data transmission.
Serial.print();	printf();	Prints data to the serial port as human-readable ASCII text.
Serial.println();	printf("……..\n");	Prints data to the serial port as human-readable ASCII text followed by a carriage return character (ASCII 13, or '\r') and a newline character (ASCII 10, or '\n').
Byte	unsigned char	Stores an 8-bit unsigned number, from 0 to 255, that represents a character in ASCII
delay(ms);	wait_ms(ms);	Pauses the program for the amount of time (in milliseconds) specified as parameter. (There are 1000 milliseconds in a second.) These functions spin the CPU to produce a small delay so they should only be used for short delays.
randomSeed(seed);	srand(seed);	Initialises the pseudo-random number generator, causing it to start at an arbitrary point in its random sequence.

millis();	Timer timer;
	timer.start();
	millis = timer.read_ms();
		These do not have equivalent functionality but can both be used to track how much time has passed in milliseconds

micros();	Timer timer;
	timer.start();
	millis = timer.read_us();
		These do not have equivalent functionality but can both be used to track how much time has passed in microseconds

Table 6 Name counterparts

7.1.3 Similarities and differences of parameters and functionality

Arduino	**Parameters** *(incl. bespoke functionality and returned values)*	**Mbed**	**Parameters** *(incl. bespoke functionality and returned values)*
pinMode(pin, INPUT);	**pin:** the Arduino pin number to set the mode of. **mode:** INPUT, OUTPUT, or INPUT_PULLUP	DigitalIn myled(pin, mode)	**pin:** DigitalIn pin to connect to **mode** (optional): The initial mode of the pin: PullUp, PullDown, PullNone, OpenDrain. The mode is automatically set to input if nothing is specified.
analogRead(pin);	**pin**: the name of the analog input pin to read from (A0 to A5). It will map input voltages between 0 and the operating voltage(5V or 3.3V) into integer values between 0 and 1023.	analogIn(pin);	**pin**: AnalogIn pin to connect to AnalogIn() reads the voltage as a fraction of the system voltage from 0 to 1. For example, if you have a 3.3V system and the applied voltage is 1.65V, then AnalogIn() reads 0.5 as the value.
digitalRead(pin)	**pin**: the Arduino pin number you want to read. The value returned is either HIGH or LOW.	myled.read()	Read the input, represented as 0 or 1 (int)
pinMode(pin, OUTPUT);	**pin:** the Arduino pin number to set the mode of. **mode:** INPUT, OUTPUT, or INPUT_PULLUP It is not possible to set the value of the pin at this stage	DigitalOut myled(pin, value)	**pin:** DigitalOut pin to connect to **value** (optional): the initial pin value. The mode will automatically be set to output. It is possible to immediately set the value of the pin at this stage

analogWrite(pin, value);	**pin**: the Arduino pin to write to.
	Value (int)**:** the duty cycle: between 0 (always off) and 255 (always on).
		PwmOut myled(pin);
		
		myled.period(seconds);
		
		myled.write(value);
			pin: PwmOut pin to connect to
			
			seconds: Change the period of a PWM signal in seconds (float) without modifying the duty

			cycle **value:** A floating-point value representing the output duty-cycle, specified as a percentage. The value should lie between 0.0f (representing on 0%) and 1.0f (representing on 100%). Use the PwmOut interface to control the frequency and duty cycle of a PWM signal.
DigitalWrite(pin, value);	Write a `HIGH` or a `LOW` value to the pin.	myled.write(value)	Set the pin logic level to 1 or 0
#include <Wire.h>	This is a library. It reads addresses as 7 bits.	I2C i2c(I2C_SDA, I2C_SCL)	This is an API. It reads addresses as 8 bits, so bits need to be shifted over by 1 when going from Arduino to Mbed.
Wire.begin(address);	The argument is a 7-bit slave **address**(optional); if not specified, join the bus as a master instead of a slave.	i2c.start();	

Wire.requestFrom(address, quantity, stop);
While(Wire.available())
{
Wire.read();
}
	address: the 7-bit address of the device to request bytes from
	quantity: the number of bytes to request
	stop (Boolean)**:** true will send a stop message after the request, releasing the bus. false will continually send a restart after the request, keeping the connection active.
	This argument mirrors the corresponding
		i2c.read(address, *data, length, repeated=false);	**address**: 8-bit I2C slave address. The bottom bit of the address is forced to 1 to indicate a read.
			data: Pointer to the byte-array to read data in to
			length: Number of bytes to read
			repeated: Repeated start, true – don't send stop at end default value is false.
			Returns: 0 on success (ack), nonzero on failure (nack)

	Mbed repeated argument in read(). available() returns the number of bytes available for retrieval with read(), which reads a byte that was transmitted from a slave device to a master or was transmitted from a master to a slave. Returns: The next byte received		
Wire.beginTransmission(address); Wire.write(value);	Begins transmission to the given 7-bit I2C slave device address. Sends a **value** as a single byte.	i2c.write(data);	Writes a single byte of **data** out on the I2C bus. Returns: '0' - NAK was received '1' - ACK was received, '2' - timeout
Wire.beginTransmission(address); Wire.write(data, length);	Begins transmission to the given 7-bit I2C slave device address. **data:** an array of data to send as bytes **length:** the number of bytes to transmit. Returns: the number of bytes written.	i2c.write(address, *data, length, repeated=false);	Transmits the bytes that are queued. Performs a complete write transaction. **address:** 8-bit I2C slave address. The bottom bit of the address is forced to 0 to indicate a write. **data:** Pointer to the byte-array data to send **length:** Number of bytes to send **repeated:** Repeated start (bool), true – do not send stop at end and default value is false. Returns: 0 on success (ack), nonzero on failure (nack)
Wire.endTransmission();	Accepts a boolean argument changing its behaviour for compatibility with certain I2C devices. If true, endTransmission() sends a stop message after transmission, releasing the I2C	i2c.stop();	

	bus. If false, endTransmission sends a restart message after transmission. The bus will not be released, which prevents another master device from transmitting between messages. This allows one master device to send multiple transmissions while in control. The default value is true. **This is similar to the repeated argument for Mbed in the above function write().** Returns a byte, which indicates the status of the transmission: 0: success 1:data too long to fit in transmit buffer 2:received NACK on transmit of address 3:received NACK on transmit of data 4:other error		
Serial.begin(speed)	**Speed** (long)**:** in bits per second (baud).	static BufferedSerial serial_port(USBTX, USBRX, baud);	**tx:** Transmit pin **rx:** Receive pin **baud** (int): The baud rate of the serial port (optional, defaults to 9600.)
Serial.print()	Numbers are printed using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is.	printf()	Accepts any string to be printed to the serial port
Serial.println();	Numbers are printed	printf("……..\n");	Accepts any string to

	using an ASCII character for each digit. Floats are similarly printed as ASCII digits, defaulting to two decimal places. Bytes are sent as a single character. Characters and strings are sent as is.		be printed to the serial port and will create a newline afterwards as long as it is finished with \n
Byte	Byte is a pre-defined type which has the same functionality as unsigned char in C. For consistency of Arduino programming style, the *byte* data type is to be preferred.	unsigned char	This is the default data-type in C and C++ for storing this sort of variable
delay(ms);	**ms** (unsigned long): the number of milliseconds to pause	wait_ms(ms);	**ms** (int)**:** the whole number of milliseconds to wait. However, in version Mbed OS 6.2 wait_ms has been deprecated, so one may have to use wait_us for microseconds and multiply all values by 1000. This was done to create precise wait capabilities.
randomSeed(seed);	**seed:** number to initialize the pseudo-random sequence. Allowed data types: unsigned long	srand();	**seed:** An integer value to be used as seed by the pseudo-random number generator algorithm.

millis();	Returns the number of milliseconds passed since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.	Timer timer;
		timer.start();
		millis = timer.read_ms();
			Returns the time passed in milliseconds. However, in version Mbed OS 6.2 read_ms has been deprecated, so one may have to use read_us for microseconds and multiply all values by 1000

micros();	Returns the number of microseconds	Timer timer; timer.start();	Returns the time passed in

	passed since the Arduino board began running the current program. This number will overflow (go back to zero), after approximately 50 days.	millis = timer.read_us();	microseconds.

Table 7 Similarities and differences of name counterpart parameters and functionality

7.1.4 Requirements fulfilled

Mandatory Requirements	Fulfilled	Reason if not fulfilled
Translate L36.h and L36.cpp from Arduino for use on FRDM-K64F	☑	
Translate LSM303.h and LSM303.cpp from Arduino for use on FRDM-K64F	☑	
Translate PololuBuzzer.h and PololuBuzzer.cpp from Arduino for use on FRDM-K64F	☑	
Translate Pushbutton.h and Pushbutton.cpp from Arduino for use on FRDM-K64F	☑	
Translate QTRSensors.h and QTRSensors.cpp from Arduino for use on FRDM-K64F	☑	
Translate ZumoMotors.h and ZumoMotors.cpp from Arduino for use on FRDM-K64F	☑	
Translate ZumoBuzzer.h from Arduino for use on FRDM-K64F	☑	
Translate ZumoReflectanceSensorArray.h from Arduino for use on FRDM-K64F	☑	
Translate ZumoShield.h from Arduino for use on FRDM-K64F	☑	
Translate example code BorderDetect from Arduino for use on FRDM-K64F	☑	
Translate example code Compass from Arduino for use on FRDM-K64F	☑	
Translate example code LineFollower from Arduino for use on FRDM-K64F	Mostly	The author did not succeed in eradicating certain bugs. The LineFollower example works for the most part, although the Zumo shifts marginally from side to side at a rapid pace and sometimes falls off the track especially at sharp turns.
Translate example code MazeSolver from Arduino for use on FRDM-K64F		The author failed at translating this example code successfully as it was heavily based on the LineFollower example working which is not the case
Translate example code PushbuttonExample from Arduino for use on FRDM-K64F	☑	

Translate example code SensorCalibration from Arduino for use on FRDM-K64F	☑	
Translate example code SumoCollisionDetect from Arduino for use on FRDM-K64F	☑	
Translate example code ZumoBuzzerExample, ZumoBuzzerExample2 and ZumoBuzzerExample3 from Arduino for use on FRDM-K64F	Partially	Only the ZumoBuzzerExample was translated as the author thought this was the prime example of the buzzer functionality. Additionally, much of the original Arduino code was written in a bespoke manner, creating a barrier while translating certain functionality as it was tricky to envision a matching Mbed method. This is also why only one example was successfully ported.
Translate example code ZumoMotorExample from Arduino for use on FRDM-K64F	☑	

Table 8 Mandatory requirements fulfilled

Optional Requirements	**Fulfilled**	**Reason if not fulfilled**
Translate example code QTRAExample, QTRARawValuesExample, QTRRCExample, and QTRRCRawValuesExample from Arduino for use on FRDM-K64F		These examples are designed for use with six QTR-1A sensors or the first six sensors of a QTR-8A module. This project only used the Zumo Reflectance Sensor Array.
Creating original examples that only support FRDM-K64F specific functionality e.g. RGB LED, SD Cards, Bluetooth, WiFi, and other FRDM-K64F pin signals	☑	Example code was created to support the functionality of the RGB LED and SD Card reader, but not of the Bluetooth and WiFi. The required sensor modules would be the JY-MCU BT board V1.05 BT for Bluetooth, and the ESP8266 module for WiFi.
Translate example code RCControl from Arduino for use on FRDM-K64F while connecting an RC servo receiver to Arduino UNO R3 and FRDM-K64F		By connecting an RC receiver and running this example sketch, the Zumo can be turned into a radio-controlled vehicle. The author did not have a receiver nor a soldering kit available to achieve this.

Table 9 Optional requirements fulfilled

7.2 Software Porting Investigation

7.2.1 Programming Environments

Over the course of the software porting process, it became evident that the compile time of Mbed Studio in contrast to the Arduino IDE was significantly prolonged. This is most likely due to the greatly increased volume of files required each time a new program is formed and compiled in Mbed Studio, as the entire Mbed OS is downloaded again upon its creation. For instance, the ZumoMotorExample in Mbed takes up 231,687 files, producing complications in memory space owing to it occupying 1GB of disk space. This has been rectified in the latest 1.2 release however the author coded this when only the Mbed Studio 1.1 or below releases were available.

It was also observed that Mbed Studio did not invariably detect potential errors via its linting tool, such as occasionally neglecting to flag missing imports for class instantiations. On the other hand, Arduino IDE does not appear to have any sort of linting tool which is detrimental to both newcomers and professionals as it increases debugging time. This is a setback for newcomers in particular as a linting tool would point out obvious or common syntax problems, for example if (val = 0) that often replaces the forgotten ==. Another factor that increased debugging time in Arduino in contrast to Mbed was the lack of a debugger, as pointed out in the background research in *Section 3.1.2*. The author ended up debugging primarily using printf() statements [35], equivalent to Serial.print() in Arduino. A serial monitor is available on both IDEs, simplifying the way in which the programmer can observe the data and behaviour of the microcontroller board. The problem with the Arduino Serial.print() is that, due to the fact that only .ino files and sketches can be opened in the IDE, one cannot print statements to analyse values in any other library file (.cpp or .h). At the same time, the Serial connection significantly slowed the operation of the Mbed board compared to the Arduino. This was slowed but not to the same degree, in some cases a change being entirely unnoticeable. Unexpected delay or interruption of the processing of code on a board can cause severe disruption in the desired performance and output of the device, especially if a timer is involved.

A frequent bug that emerged with the Mbed Studio IDE was that it occasionally detected non-existent errors. Sometimes, the IDE would state compile errors that had been cleared, as it would be compiling an older version of what had been typed earlier. This would in most cases return to normality by way of merely restarting the IDE or your computer. Right-clicking a program and selecting “Duplicate” does not duplicate the program effectively, giving an error stating “no linker script found”. The only effective method of duplicating a program is to create a new one and then either drag and drop the files in on the IDE or through your PC’s File Explorer. These bugs come as no surprise given how recently this IDE was formally released for the first time.

The mandatory int main() function in Mbed lacking in the Arduino library and IDE at times muddled the author as this would be the final aspect for appending that would be overlooked as it is not once defined in the Arduino library. This is due to the fact that Arduino does not require a main function to be stated, the equivalent being setup() and loop() called from the compiler in that order, as mentioned above.

A frustrating element of the latest Mbed OS 6.2 is that the wait_ms and timer.read_ms are deprecated as Mbed tries to discourage you from making long delays. This makes porting from a piece of Arduino code with frequently appearing millis() is difficult as all the timings must be multiplied by 1000, however to ensure one does this consistently is problematic.

For the duration of this project, it was noticed that the Mbed OS updated far more frequently than Arduino. This meant a lot of code completed by the author on Mbed a few months prior caused errors due to features being deprecated, unlike on Arduino for which the ZumoShield library was completed in early 2018 and still works well across the platform. These errors included the Serial API being replaced by BufferedSerial and the printf() for

floats being discontinued to reduce memory usage when the OS updated from 5.0 to 6.0. This forced “minimal-printf” was dealt with by adding an mbed_app.json file to the program to enable floats being accepted again. This was also unhelpful as it made a lot of example code submitted by Mbed users a few years ago harder to use as guidance as it was now quite inconsistent with the current OS. This is another indicator that the Arduino platform is far more established. For this reason, one must be prepared and mindful of the fact that the Mbed OS is more likely to update when porting from Mbed to Arduino, and extra work will have to be undertaken to get around this.

7.2.2 Code translation observations

A characteristic of the Arduino ZumoShield library that ultimately proved to be an asset in code translation was the information in the #if, #else, #endif and #ifdef directives. This library could instead have been coded entirely in a bespoke manner, solely designed for certain selected Arduino boards with specific board requirements and capabilities. Instead, the ZumoMotors.cpp file provided optional code for alternatives as seen in *Figure 13* below, giving the author an idea of the figures and functions needed to perform this desired behaviour.

```
#if defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined
(__AVR_ATmega32U4__)
  #define USE_20KHZ_PWM
#endif

…

#ifdef USE_20KHZ_PWM
  OCR1B = speed;
#else
  analogWrite(PWM_L, speed * 51 / 80); // default to using analogWrite, mapping 400 to
255
#endif
```

Figure 13 Code snippet from Arduino ZumoShield library file ZumoMotors.cpp

OCR1B in the above figure refers to an Arduino MCU specific timer that, as one can see, is interacted with in quite a unique way by comparison with analogWrite. As presented in *Table 6*, analogWrite can be replaced with PwmOut in Mbed; in this way the Mbed equivalent ZumoMotors.cpp file was developed. In the absence of the information provided for the #else directive, the author would have found it rather complex deciphering the parallel Mbed code that would result in the same performance if this file was more tailor-made. Unfortunately, this was the case for certain customised features that revolve around Arduino-specific parameters, impeding the porting process in other regions of the library. This included the PololuBuzzer files, a snippet displayed below in *Figure 14*. Once again, the code references timers that are unique to Arduino microcontrollers, yet this time no alternative functionality is immediately obvious to hint toward the manner in which it might be achievable to replicate this behaviour in Mbed. This answers why the author only achieved the conversion of the ZumoBuzzerExample while not including ZumoBuzzerExample2 and ZumoBuzzerExample3. Nonetheless, the author believed this to be reasonable in providing a replication and illustration of this buzzer functionality.

```
#ifdef __AVR_ATmega32U4__

// Timer4 overflow interrupt
ISR (TIMER4_OVF_vect)
{
  if (buzzerTimeout-- == 0)
  {
    DISABLE_TIMER_INTERRUPT();
    sei();                                  // re-enable global interrupts (nextNote()
is very slow)
    TCCR4B = (TCCR4B & 0xF0) | TIMER4_CLK_8;  // select IO clock
    unsigned int top = (F_CPU/16) / 1000;     // set TOP for freq = 1 kHz:
    TC4H = top >> 8;                          // top 2 bits... (TC4H temporarily stores
top 2 bits of 10-bit accesses)
    OCR4C = top;                              // and bottom 8 bits
    TC4H = 0;                                 // 0% duty cycle: top 2 bits...
    OCR4D = 0;                                // and bottom 8 bits
    buzzerFinished = 1;
    if (buzzerSequence && (play_mode_setting == PLAY_AUTOMATIC))
      nextNote();
  }
}

#else // 168P or 328P

// Timer2 overflow interrupt
ISR (TIMER2_OVF_vect)
{
  if (buzzerTimeout-- == 0)
  {
    DISABLE_TIMER_INTERRUPT();
    sei();                                  // re-enable global interrupts (nextNote()
is very slow)
    TCCR2B = (TCCR2B & 0xF8) | TIMER2_CLK_32; // select IO clock
    OCR2A = (F_CPU/64) / 1000;                // set TOP for freq = 1 kHz
    OCR2B = 0;                                // 0% duty cycle
    buzzerFinished = 1;
    if (buzzerSequence && (play_mode_setting == PLAY_AUTOMATIC))
      nextNote();
  }
}

#endif
```

Figure 14 Code snippet from Arduino ZumoShield library file PololuBuzzer.cpp

The author removed the code within the directives provided for the ATmega32U4, as this applies to the Zumo 32U4 robot instead of the ZumoShield robot used in this project.

The author ran into a number of obstacles while converting the ZumoMotorExample, as it was one of the initial example codes converted hence the author was not as familiar with the process at this stage. Following subsequent failed attempts with seemingly no forum-based knowledge on this conversion, the author did some further research to discover the motor driver on Zumo Shield named DRV8835. It was then apparent that the ZumoMotors.h and ZumoMotors.cpp files were based on the Arduino library for the Pololu DRV8835 Dual Motor Driver Shield [36]. Thereafter, the author fell upon a user-submitted example conversion of this library on the Mbed forum [37], applying it to produce the Mbed adaptation of the ZumoMotor files instead. This operated more effectively than former efforts, perhaps due to the insight of an Mbed programmer with high-level expertise. Another set of files for which their development was aided by the high-level expertise of an Mbed

forum member were QTRSensors.h and QTRSensors.cpp. A well-constructed set of translated files were provided by this member [38], nevertheless they state within the comments that they are unsure on what direction to take for some sections of the code as they do not have a test sensor on them. This indicates that the code was not at all tested but was built on the basis of the user's extensive knowledge in the field, as a proposed solution to the translation of these files. The author thereby utilised elements of this proposed solution, refining it during and after ample testing with the Zumo Reflectance Sensor Array attached to the robot.

Another important factor to remember was that, due to custom functions on Arduino often expecting a parameter between the range of 0 to 255, this value would often be passed around as a variable stored as an integer. In Mbed this range is always mapped to the range of 0 to 1, so the decimal point values between these two values are highly important to represent the variance and therefore should be passed around in a variable stored as a float.

Arduino uses a few Arduino-specific names for elements of the C++ language that already have their own name. This includes unsigned char being named Byte and srand() being named randomSeed(), despite having an identical functionality. The theory is that, despite not being particularly necessary, a newer programmer will find these names easier to understand and write simply. Arduino explains, "For consistency of Arduino programming style, the byte data type is to be preferred [39]."

Regardless of whether converted class code appeared to work effectively for one example code, this did not guarantee that it would operate as successfully for another example code if the first was not producing the precise values that the Arduino compiler would have been expecting. In other words, an inaccuracy that one could get away with for a single example could carry over and cause a larger error in a future piece of code that could not be circumvented. As mentioned in *Section 7.1.*2, the author was oftentimes unclear when it was appropriate to dedicate time to making the functionality identical, as it was either a necessity or unnecessary.

The author predicts that the bug in the LineFollower function is based on a slightly inaccurate sensor reading, an inaccuracy that has no impact on the performance of the BorderDetect example but has an error that presents itself later in LineFollower. This was confirmed when the author read the sensor readings on the Arduino Uno R3, exchanged the board to the FRDM-K64F while ensuring the Zumo robot stayed in the exact same position, and read the new sensor readings. They were off in a sporadic manner, as the error would change at each of these tests. The author could not distinguish what was causing this slight error, hypothesising that the Mbed timer generates different results to the Arduino millis() function; changing the sensor readings. Although the LineFollower example works for periods of time, the impact of this error was that the robot would sometimes fall off the track, especially at sharp turns.

7.2.3 Hardware

It is advisable to be aware of the level of charge remaining inside of the Zumo Robot. The author experienced a period where they could not determine where in the code the haphazard behaviour of the robot was coming from, particularly in regards to its line following ability, only to realise that this was a result of a low battery level. While operating the Zumo robot, the Arduino Uno R3 was provided a sufficient power supply solely from the batteries inside the robot, powering the board and the robot movement. The FRDM-K64F appears to require a greater supply of power, as when it is relying on the power from the internal batteries alone, the robot can barely move in the expected manner. It turns out that the performance goes back to its expected behaviour when the FRDM-K64F is connected to the PC using a 5V micro USB wire. The limitation of a wire being connected to the PC during testing, holding the robot back, was entirely impractical especially given the purpose of the robot being mostly based around movement. The wire would often get tangled and twisted

within itself during testing with the LineFollower example that is following a cyclical path. To circumvent this troublesome issue, a small portable power pack was connected to the FRDM-K64F and held on board using a rubber band.

Similarly, the implementation of the SD Card functionality was useful in avoiding this repeated issue with the connected wire as well. The completion of this code fulfilled a secondary, optional task exposing the FRDM-K64F's extra abilities in contrast to the Arduino Uno R3. This functionality is present in the sdCard, SumoCollisionDetect and compass examples, with optional implementation depending on whether a directive is present or not. These examples made use of the FRDM-K64F's RGB LED and SD Card capabilities. The green LED indicates the program has started running, red to represent the SD Card being mounted, blue to indicate the SD Card was written to, and red again to indicate the file was closed and card unmounted. During Serial communication, both the Arduino Uno R3 and FRDM-K64F must be connected to the PC to send data via the USB Type-B and micro USB wires respectively. A key advantage of writing to an SD Card is that collecting data would not have to be achieved through a movement-hindering wire. The Arduino Uno R3 would benefit from this as well, as the author's USB Type-B wire is very short and stiff so the laptop it connected to had to be carried around with the robot when testing its movement.

Presently, the bootloader of the FRDM-K64F often gets corrupted when plugged into a Windows 10 machine. The author found the quickest solution to be to redownload the firmware [40], currently with the name k20dx_frdmk64f_if_crc_legacy_0x5000.bin, to your PC and reupload it to the FRDM-K64F in order to update the firmware. This is a known issue that has been around since at least 2016 or more. This is another instance where Arduino presents itself to be a more solid or reliable platform, as pointed out by the author of the s-lab article [9].

As warned by Styger, the sensor readings can be temperamental depending on the material the line it is expected to detect is made from [29]. On first attempt at building the line following track, the material used was simply marker pen ink against white card. Similar to the issue with ink Styger had encountered due to his line being made by a laser printer, both were solved by replacing these lines with black tape which worked as expected.

For evaluating the microcontroller board performance comparison, the SumoCollisionDetect and compass examples were coded on Mbed utilising the on board 6-axis combo Sensor Accelerometer and Magnetometer on the FRDM-K64F instead of those on the Zumo robot. For this to work, the PTE25 and PTE24 pins need to be bent away when attaching the board to the Zumo robot as they cause interference.

The Arduino Uno R3 did prove that it was adversely affected by having half the flash memory of the FRDM-K64F, as it was unable to store multiple calculation values that were required for testing.

All of these discoveries and challenges affected how the system was built.

8. Experiments

8.1 Experiment 1 – PushButton

The PushButton example on the FRDM-K64F was tested by observing whether the LED light turned on as expected after pressing the pushbutton on the robot. The original and translated example provide 3 techniques for implementing this within the code, so all except the one being tested had to be commented out. This experiment and all 3 methods were successful.

8.2 Experiment 2 – ZumoBuzzer

The ZumoBuzzer example experiment involved simply observing the melody produced by the robot with the FRDM-K64F and comparing it to the one produced on the original Arduino Uno R3. The melody produced was identical and thus this experiment was successful.

8.3 Experiment 3 – ZumoMotor

The ZumoMotor example experiment involved observing whether the robot's movement with the FRDM-K64F mirrored that of the robot with the Arduino Uno R3. During development, the author had noticed that robot was not slowing and speeding up or having any variety in speed for this example. This was fixed and after finishing development, the robot movement was identical to the original and thus this experiment was successful.

8.4 Experiment 4 – Border Detect

The BorderDetect example was tested against a black granite surface, the robot encircled by a white line made of masking tape, as depicted in . To ensure the code was effective, it was observed whether or not the robot with the FRDM-K64F stopped precisely at the border representing the line. During code development the robot was passing this line and stopping slightly after it, but after correcting the code communicating with the sensors the robot stopped exactly at the line just as it did with the Arduino board. This experiment was completed successfully after numerous repeated tests, confirming that the code translation for this example was successful.

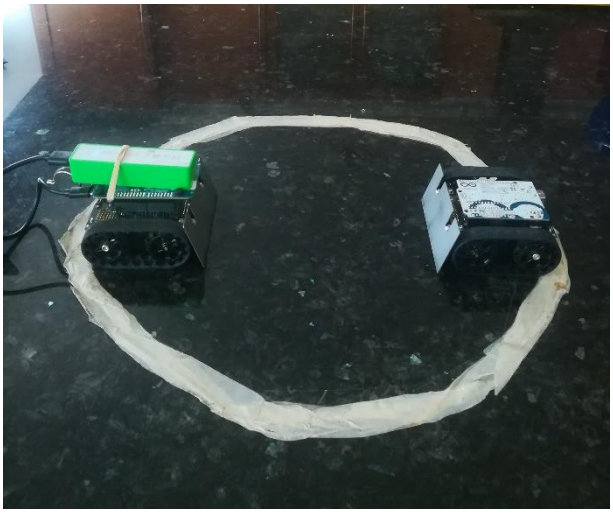

Figure 15 Robot battle arena

8.5 Experiment 5 – Line Follower

The Line Follower experiment observed how effectively the robot followed the path presented in *Figure 16*. In this experiment robot mostly followed the line, although shifting marginally from side to side at a rapid pace and sometimes falling off the track especially at sharp turns.

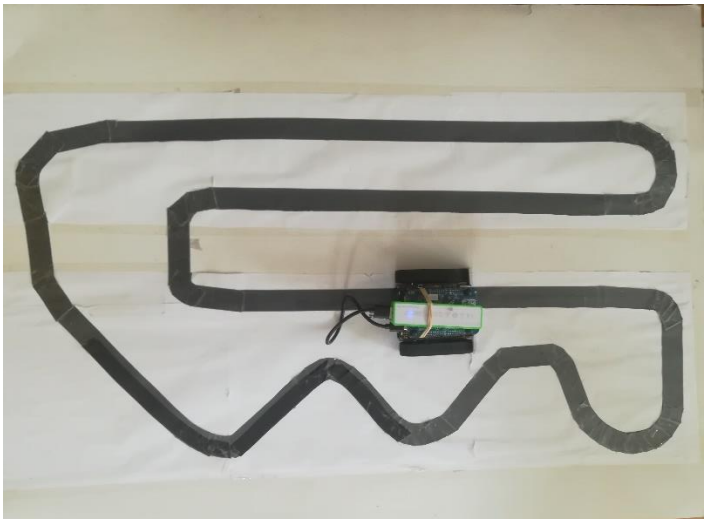

Figure 16 Line Follower path

8.6 Experiment 6 – Sumo Collision Detect

The SumoCollisionDetect example was tested in the same arena as that used when testing the BorderDetect example. Unlike Experiment 1, evaluating the success of this translated example code would not be feasible or reliable without a second robot. The author bought another Zumo robot, one using the original library's SumoCollisionDetect example code on the Arduino Uno R3, the other using the translated example code on the FRDM-K64F. This experiment would enable a battle between the two robots, the loser being the one to fall past the white line. Not only would this experiment compare the general board performance, the FRDM-K64F utilises its on-board Accelerometer and Magnetometer instead of those on the Zumo robot and this would be compared as well. To ensure reliability of results, the robots were both placed in the same position in the battlefield at the start of each experiment as shown in *Figure 15*. The batteries used were also new and fully charged to minimise any outward factors that could impact the results. 35 tests were executed, 5 were won by the robot with the Arduino Uno R3 board and 30 with the FRDM-K64F board. The robot with the FRDM-K64F board had a success rate of 86%.

8.7 Experiment 7 – Compass

The Compass example experiment observed how effectively the robot coordinated 90 degree turns in squares. A line was drawn with a ruler to keep track of the robot's direction and turn, and a compass used to measure this angle. The angle measured was 92°, although the tracking of the line was slightly tedious due to the robot moving quickly so the result may have been even more accurate.

8.8 Timing Experiments

The compile time on either IDE for an empty test program was measured, the Arduino taking exactly 5 seconds in contrast to the Mbed board taking 9 minutes and 45 seconds. As mentioned above, this is likely due to the necessity of having a copy the OS redownloaded and recompiled for every new program. This compile time has been reduced to 15 seconds in the latest version however it was released immediately after the author finished programming this project.

The time complexity of both libraries is roughly quite similar due to the close mirroring of code structure between the two. Unfortunately this was difficult to test exactly as a lot of the pre-defined code is hidden behind the compiler, so it is impossible to count this complexity in its entirety without full access to all the code utilised in both projects.

The response time of the robot was tested by setting a timer before and after turning the LED on. The results showed that the FRDM-K64F took 5 microseconds to execute this command and the Arduino Uno R3 took 8.

9. System Evaluation

The results from the experiments in the previous section indicate that this system was correctly translated and implemented, and give some idea of where the differences in the boards lie. On the most part, most experiments conducted due to the scope of the library were limiting in that they could only really prove that the boards performed equally to each other. The key instances where there was a difference in performance was with the LineFollower and SumoCollisionDetect examples, the first would indicate the FRDM-K64F board performed worse at this task and the latter better. However, the problem with the LineFollower's performance was most likely not the fault of or a representation of the board but the program translation. The SumoCollisionDetect is the only experiment that provides substantial evidence that the FRDM-K64F can provide a superior performance. This experiment's success may indicate that the Accelerometer and Magnetometer on the FRDM-K64F are advanced to those on the Zumo robot, giving it an advantage by detecting a collision earlier. It may also be due to the robot's higher clock speed, flash memory and RAM *(Table 2)*, or that the robot receives more power to push the Arduino robot away due to the additional energy supplied via the portable charger. The superior results of the FRDM-K64F may be due to all or a few of these reasons.

10. Conclusion

This assessment was successful in terms of achievement of code and as an investigation as a whole. Unfortunately not all code translation objectives were completed, namely the LineFollower and MazeSolver examples. Perhaps with further time the author could have fixed the issues that they could not solve in the given time period.

To enhance the investigation aspect of this report, as a potential extension of this project the author could have developed examples for the two boards that were able to show more clearly a comparison in their performance. This could be achieved by researching online user-defined Arduino examples that do not exist in the ZumoShield library for additional functionality of the Zumo robot or by creating original programs. With a greater time constraint, the author could have investigated the additional 32 inner pins on the FRDM-K64F and tested what other signals and capabilities these pins uniquely provide to the Zumo robot. Another future project could be connecting an RC servo and receiver to both boards and turning the Zumo into a radio-controlled vehicle, again comparing the performance. An improved version of the maze solver could also be implemented, as Styger suggested, creating the ability for the robot to return from the finish to the start automatically, or exploring the full maze to find the smallest path [29].

The extension would also explore additional sensors that can be purchased for the Zumo robot, such as lidar or sonar range finders, and sharp distance sensors [41] for detecting nearby objects. To implement this, it is necessary to purchase, “Connector and jumper wires, for connecting additional sensors and components” [31]. Other sensor modules that can be attached to the boards include a temperature & humidity sensor [3], vibration, joystick, and digit display [3]. As much as this project made plenty of useful discoveries, there is vast scope for improvement.

One can conclude from this investigation that Mbed’s FRDM-K64F board potentially has more powerful capabilities, however, if one wants to utilise these they must prepare for a platform that is not as refined as Arduino’s. Mbed does seem quick to produce improvements, proven by the number of updates that were released over the duration of this project, so one can assume that the Mbed platform will become more solid in the near future. Despite there not being a vast number of resources for porting from Arduino to Mbed online, it is achievable with extensive research and time.

References

[1] Hackster, “Arduino IoT Cloud Amazon Alexa Integration,” 2019. [Online]. Available: https://www.hackster.io/303628/arduino-iot-cloud-amazon-alexa-integration-4e6078. [Accessed 21 November 2019].

[2] Arduino, “Arduino Uno Rev3,” [Online]. Available: https://store.arduino.cc/arduino-uno-rev3. [Accessed 19 November 2019].

[3] Arm Mbed, “FRDM-K64F,” [Online]. Available: https://os.mbed.com/platforms/FRDM-K64F/. [Accessed 19 November 2019].

[4] Pololu Corporation, “Pololu Zumo Shield Arduino Library,” 2018. [Online]. Available: https://pololu.github.io/zumo-shield-arduino-library/. [Accessed 19 November 2019].

[5] Arduino, “Language Reference,” [Online]. Available: https://www.arduino.cc/reference/en/. [Accessed 15 August 2020].

[6] Mbed, “Full API list,” [Online]. Available: https://os.mbed.com/docs/mbed-os/v6.2/apis/index.html. [Accessed 15 08 2020].

[7] B. Huang and R. Derek, The Arduino inventor's guide: learn electronics by making 10 awesome projects, San Francisco: Sparkfun Electronics, 2017, p. 72.

[8] J. N. Rejeev Grover, “Method of Porting Software”. USA Patent US 7,185,344 B2, 2007.

[9] s-lab, “Mbed VS Arduino,” 2013. [Online]. Available: http://slab.concordia.ca/2013/mbed/mbed-comparison-test/. [Accessed 2019 November 18].

[10] s-lab, “Mbed Overview,” 2013. [Online]. Available: http://slab.concordia.ca/2013/mbed/mbed-overview/. [Accessed 18 November 2019].

[11] J. Carver, “K64F eCompass,” 2014. [Online]. Available: https://os.mbed.com/users/JimCarver/code/K64F_eCompass/. [Accessed 19 November 2019].

[12] Mbed, “Handbook - Homepage,” [Online]. Available: https://os.mbed.com/handbook/Homepage. [Accessed 18 November 2019].

[13] Mbed, “Cookbook - Homepage,” [Online]. Available: https://os.mbed.com/cookbook/Homepage. [Accessed 18 November 2019].

[14] J. Yiu, “Technical Article - Basics of porting C-code to and between ARM CPUs: From 8-/16-Bit MCUs to Cortex-M0,” 2011. [Online]. Available: https://www.embedded.com/basics-of-porting-c-code-to-and-between-arm-cpus-from-8-16-bit-mcus-to-cortex-m0/. [Accessed 19 November 2019].

[15] Arduino, “Software,” [Online]. Available: https://www.arduino.cc/en/main/software. [Accessed 17 November 2019].

[16] Arduino, “Getting Started with Arduino UNO,” 2019. [Online]. Available: https://www.arduino.cc/en/Guide/ArduinoUno. [Accessed 17 November 2019].

[17] Arduino, “Environment,” 2015. [Online]. Available: https://www.arduino.cc/en/Guide/Environment. [Accessed 17 October 2019].

[18] Mbed, “TCP IP protocols and APIs,” 21 November 2019. [Online]. Available: https://os.mbed.com/handbook/TCP-IP-protocols-and-APIs.

[19] Mbed, “The Mbed implementation of PSA,” [Online]. Available: https://os.mbed.com/docs/mbed-os/v6.0/apis/security-concepts.html. [Accessed 15 August 2020].

[20] B. Lim, “Moving from Arduino to Arm,” 2018. [Online]. Available: http://westsideelectronics.com/moving-from-arduino-to-arm/. [Accessed 17 November 2019].

[21] E. Styger, “The Freedom Zumo Robot,” 2013. [Online]. Available:

https://mcuoneclipse.com/2013/01/31/the-freedom-zumo-robot/. [Accessed 19 November 2019].

[22] NXP, “PROCESSOR-EXPERT: Processor Expert software - Integration with CodeWarrior tool,” [Online]. Available: https://www.nxp.com/design/software/development-software/processor-expert-software/processor-expert-software-integration-with-codewarrior-tool:PROCESSOR-EXPERT. [Accessed 19 November 2019].

[23] NXP, “Get Started with the FRDM-K64F,” [Online]. Available: https://www.nxp.com/document/guide/get-started-with-the-frdm-k64f:NGS-FRDM-K64F. [Accessed 19 November 2019].

[24] NXP, “FRDM-K64F: Freedom Development Platform for Kinetis® K64, K63, and K24 MCUs,” [Online]. Available: https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F. [Accessed 19 November 2019].

[25] Components101, “Arduino Uno,” [Online]. Available: Arduino Uno. [Accessed 15 August 2020].

[26] Arduino, “ATmega168/328P-Arduino Pin Mapping,” [Online]. Available: https://www.arduino.cc/en/Hacking/PinMapping168. [Accessed 18 November 2019].

[27] Atmel, “ATmega328P Datasheet,” 2015. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf. [Accessed 18 November 2019].

[28] NXP, “Kinetis K64F Sub-Family Data Sheet, Rev. 7,” 2016. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/K64P144M120SF5.pdf. [Accessed 18 November 2019].

[29] E. Styger, “Freedom Robot solves the Maze,” 2013. [Online]. Available: https://mcuoneclipse.com/2013/03/20/freedom-robot-solves-the-maze/. [Accessed 19 November 2019].

[30] L. Levin, “Getting Started with NXP FRDM-K64F and the Mbed Environment,” 2016. [Online]. Available: https://www.hackster.io/leroy2le/getting-started-with-nxp-frdm-k64f-and-the-mbed-environment-139d8d. [Accessed 19 November 2019].

[31] Pololu Corporation, “Pololu Zumo Shield for Arduino User’s Guide,” 2001-2019. [Online]. Available: https://www.pololu.com/docs/0J57/all. [Accessed 19 November 2019].

[32] E. Styger, “Zumo Line Following with FRDM-KL25Z,” 2013. [Online]. Available: https://mcuoneclipse.com/2013/02/08/zumo-line-following-with-frdm-kl25z/. [Accessed 19 November 2019].

[33] Mbed, “FRDM-K64F rev3 pinout/pin-multiplexing,” [Online]. Available: https://os.mbed.com/media/uploads/GregC/frdm-k64f_pinout_reve3.xls. [Accessed 17 August 2020].

[34] Arduino, “millis(),” [Online]. Available: https://www.arduino.cc/reference/en/language/functions/time/millis/. [Accessed 21 August 2020].

[35] Mbed, “Debugging using printf() statements,” [Online]. Available: https://os.mbed.com/docs/mbed-os/v6.2/debug-test/debugging-using-printf-statements.html. [Accessed 21 August 2020].

[36] Pololu, “Arduino library for the Pololu DRV8835 Dual Motor Driver Shield,” 30 March 2018. [Online]. Available: https://github.com/pololu/drv8835-motor-shield. [Accessed 21 August 2020].

[37] E. Coyle, “DRV8835,” 25 August 2015. [Online]. Available: https://os.mbed.com/users/DrCoyle/code/DRV8835/. [Accessed 21 August 2020].

[38] M. Phillipps, “PololuQTRSensors,” 27 August 2015. [Online]. Available:

https://os.mbed.com/users/phillippsm/code/PololuQTRSensors/. [Accessed 21 August 2020].

[39] Arduino, “unsigned char,” [Online]. Available: https://www.arduino.cc/reference/en/language/variables/data-types/unsignedchar/. [Accessed 21 August 2020].

[40] NXP, “OPENSDA: OpenSDA Serial and Debug Adapter,” [Online]. Available: https://www.nxp.com/design/microcontrollers-developer-resources/ides-for-kinetis-mcus/opensda-serial-and-debug-adapter:OPENSDA?&tid=vanOpenSDA#FRDM-K64F. [Accessed 21 August 2020].

[41] Pololu, “Proximity Sensors and Range Finders,” [Online]. Available: https://www.pololu.com/category/189/proximity-sensors-and-range-finders. [Accessed 19 November 2019].

[42] Pololu, “Maze Solver,” [Online]. Available: https://www.pololu.com/docs/0J57/all#7.e. [Accessed 19 November 2019].

Appendix

Interim Log

Date	Meeting Contents
7/10/19	Overview of deadlines, useful resources, useful beginner practise tasks to familiarise oneself with ZUMO, relevant suggestions, Trello platform
13/11/19	Reminder of essential requirements for Interim report, signing the ethical compliance form
03/07/20	Suggestions for structure and detail to add for first draft to send of the report