The Qbead: putting a qubit in everyone's hands

Software Framework

Ard Geuze
Mack Chen
Victor Hoogendijk

Technische Universiteit Delft

TU Delft

The Qbead: putting a qubit in everyone's hands

Software Framework

by

Ard Geuze
Mack Chen
Victor Hoogendijk

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Tuesday June 24, 2025 at 11:00.

Student number:	Ard Geuze,	5743052
	Mack Chen,	5469058
	Victor Hoogendijk,	5824559
Project duration:	April 22, 2025 – June 27, 2025	
Thesis committee:	C. Errando Herranz	TU Delft, supervisor
	Dr.ir. N.P. van der Meijs	TU Delft
	Dr.ir. R. Santbergen	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

This thesis explores the development of the software framework for the Qbead. A physical representation of a quantum bit (qubit) designed to be held in the hand. Shaped as a sphere, the Qbead visualises the Bloch sphere from quantum mechanics, with internal LEDs that illuminate to display the qubit's state. The main goal of the Qbead is to provide students with a more intuitive and accessible way to learn about quantum computing.

While the concept shows promise, the existing software framework lacks the necessary features to function as an effective educational tool. This project focuses on further developing the codebase to create a solid foundation for future use. By expanding the software and implementing quantum-related experiments, the Qbead can better support hands-on education in quantum mechanics.

The new classes are added to create a better framework for other functions. The X, Y, Z, and Hadamard single-qubit gates are implemented. 3 ways to detect input, rotating, shaking, and tapping have been added. Some experiments are developed to show off the single-qubit gate functions and decoherence of the state.

Preface

This thesis ”The Qbead: putting a qubit in everyone’s hands” is written in regard to the Bachelor Graduation Project for Electrical Engineering at the Delft University of Technology. This project was proposed by C. Errando Herranz with the goal of further developing the Qbead. There are two subgroups of 3 working on this project. This thesis focuses on the software part of the development. The other part will dive into the hardware part.

This project made us much more knowledgeable about the world of quantum computing. And, it would indeed be a dream if this subject could be taught in a simple manner. We hope that with the help of the Qbead, we can make teaching quantum computing easier. We would like to thank our supervisor, C. Errando Herranz, for giving us this opportunity and guiding us through this project. Also, Stefan, from the University of Massachusetts Amherst, who has been working on this project from the beginning, has helped us with the software development on the Qbead and gave useful feedback on the code. And finally, we would also like to give our gratitude to our fellow team members Fynn van der Wal, Henk Bakker, and Rutger Gosselink, who have been working with us for these three months.

Ard Geuze
Mack Chen
Victor Hoogendijk
Delft, June 2025

Contents

1

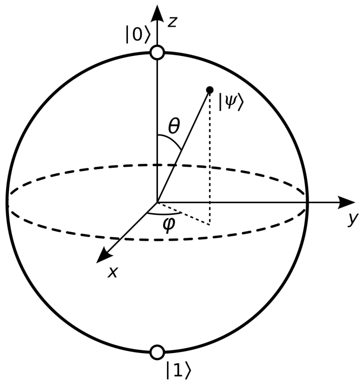
Introduction

1.1. Classical and quantum computers

One of the most exciting scientific fields to date is quantum mechanics. The research on quantum mechanics that is currently being worked on can have a significant impact on human lives [1]. This field also opens up many possibilities that are not possible with classical mechanics. This thesis will mainly dive into the field of quantum computing.
In the last few decades, computers have evolved into a product that everyone uses in their daily lives. Computers use classical bits to perform calculations. These classical bits can only hold two states, either 0 or 1. Currently, complex calculations can be run on supercomputers, which significantly shortens the calculation duration compared to ordinary computers. However, for even more complex calculations, there might be a limit on how much faster supercomputers can be made [2]. A quantum computer could unlock even more computational power that would not be possible through classical computers. Some calculations that would be too long to perform on classical computers can be done exponentially faster on a quantum computer. Classical computers are, at the time of writing this, still miles ahead in terms of computational power; however, current developments on quantum computers are narrowing this gap [3].

1.2. Quantum computing

Quantum computers store information differently than classical computers. The information in a quantum computer is stored with quantum bits (qubits). These are similar to classical bits represented in binary, but differ in some ways [4]. Qubits can exist in a superposition of the two states, which means the qubit can be both 0 and 1 simultaneously. However, when the qubit is measured, the outcome will always be either a 0 or a 1. The outcome of this measurement depends on the internal state of the qubit.
A popular way to visualise the state of a qubit is by a Bloch sphere [5], seen in Figure 1.1a. The 0 and 1 states are on the north and south poles, respectively. Everything between the north and south poles represents a superposition of 0 and 1. Every point on the surface of the sphere thus represents a state of the qubit, shown with vector ψ. The direction of vector ψ is determined by θ and φ, which are bounded to $[0, \pi]$ and $[0, 2\pi)$ respectively.



(a) The Bloch sphere [6]

(b) Qbead in idle state

Figure 1.1

1.3. Qbead

A product that is currently in development, called the ”Qbead”, makes it possible to deliver a qubit in someone’s hands. The Qbead is a physical representation of a qubit with the size of a golf ball, shown in Figure 1.1b. It is shaped as a sphere and represents the Bloch sphere. Inside the shell, there is a flexible PCB that wraps around the inner shell that contains the microcontroller and battery. For showing the state on such a sphere, LEDs are placed on the PCB, and these can be illuminated to represent a state. Having converted a theoretical model to a physical product, this product can serve as an educational tool for students to build more intuition about quantum computing. Another project [7] tried to create a physical Bloch sphere simulator as well. There, the Bloch sphere is placed on a rotation system, which is controlled by a mobile phone app. It can show the quantum states and execute gate operations. However, the complete setup is far more complicated and bigger in scale.
Getting young people engaged with quantum mechanics is quite a challenge [8]; the Qbead could be used to engage them. Hu [8] mentioned that the Bloch sphere could be an effective tool to help students better understand possible qubit states. However, the Bloch sphere might still be difficult to grasp for students. Therefore, the Qbead can also be a useful tool to help better understand this subject.

1.4. Problem

Since the Qbead is still in development, there are still improvements that can be made on both the hardware and software sides of the Qbead. For this project, the software framework is the main focus. The software framework still lacks features to be used as an educational tool. By further developing the code, it will have a better foundation for future experiments. These experiments will be worked out as Arduino sketches that will perform quantum-related experiments on the Qbead to educate students. The goal is to make this product ready, so it can be used as a handy educational tool to engage more people in science and quantum mechanics and teach them the basics of quantum computing.

1.5. Microcontroller

The software of the Qbead is run on a microcontroller. A microcontroller is essentially a very small computer that runs the Arduino code. For this project, the **Seeed XIAO nRF52840 Sense** was initially chosen as the microcontroller.

Seeed XIAO nRF52840

The code that was already developed was developed for the Seeed XIAO nRF52840 Sense. This is a cheap and open-source microcontroller that is quite energy efficient. It is also easily programmable via the Arduino IDE. This made it a good choice to use. However, for hardware reasons, another microcontroller was chosen [9].

ESP32 implantation

During this bachelor thesis project, there were also some hardware improvements made to the Qbead. The XIAO nRF52840 Sense was replaced by the XIAO ESP32S3, which meant that the software framework needed a rework to fully support the ESP32. Most of the code could easily be ported to the ESP32; however, the Bluetooth and IMU drivers were different, so those parts had to be modified. The ESP32 is also easily programmable via the Arduino IDE.

1.6. IMU

The inertial measurement unit (IMU) is also a very important hardware component for the Qbead. The IMU is responsible for measuring acceleration and angular velocity. Some IMUs can also measure the magnetic field. The devices that measure these quantities are called the accelerometer, gyroscope, and magnetometer, respectively. The XIAO nRF52840 has a built-in IMU with 6 axes, which includes only the acceleration and the angular velocity. The XIAO ESP32S3 does not have a built-in IMU. The hardware team chose the ICM-20948 as the new IMU. This is a 9-axis IMU, which means that it also measures the magnetic field.

1.7. Structure of this thesis

This thesis is divided into multiple chapters about the changes that have been made to the software framework. Chapter 2 outlines the programme of requirements for this project. Foreknowledge for this project is written in Chapter 3, which will be the foundation for the classes and implementation. This chapter is recommended to read through when the reader has no prior knowledge about quantum computing. Chapter 4 goes into the classes that were added to the software framework. Chapter 5 describes the appliances for the Qbead and explains the working of the code. At last, the discussion and conclusion can be read in Chapter 6. In Appendix A, the equations are shown, used for the gate animations on the Qbead. Lastly, Appendix B shows the header files and the experiment's code.

2

Programme of requirements

To justify the choices that have been made for the Qbead. A Programme of Requirements (PoR) has been drafted. The requirements are set for the software features that the Qbead must/should have.

2.1. Mandatory requirements

These requirements must always be complied with. This section is divided into two parts: functional requirements, which will go into the use of the product; and non-functional requirements, which will describe the qualities that the product has.

2.1.1. Functional requirements

(A1) The Qbead must be able to connect to a computer via Bluetooth or USB;

(A2) The Qbead must display its current state on the LEDs;

(A3) The Qbead must be able to do the single qubit gates: Pauli-X, -Y, and -Z and the Hadamard gate, and display it;

(A4) The Qbead must be able to collapse to $|0\rangle$ or $|1\rangle$;

(A5) The software must process the state updates and user inputs within 100 ms after the gesture is finished.

2.1.2. Non-functional requirements

(B1) The Qbead's axes must be identifiable;

(B2) The software and required packages shall be installed on the user's computer;

(B3) The software shall be run in Arduino IDE to execute operations;

(B4) The whole system and code must stay open source.

2.2. Trade-off requirements

Criteria that would preferably be complied with.

(C1) The product should preferably be able to display decoherence;

(C2) The product should preferably be able to allow dynamic decoupling;

(C3) The rotation should be correctly identified 90% of the time;

(C4) The shaking should be correctly identified 90% of the time;

(C5) The tapping should be correctly identified 90% of the time.

3

Theoretical Background

In this chapter, all the theory that is behind the experiments will be discussed. This theoretical background supports the material in the following chapters. If prior knowledge is recommended for a future section, it will refer to the dedicated section in this chapter.

3.1. Quantum Bits

Quantum bits (qubits) are the fundamental building blocks of quantum computers. Qubits are the quantum equivalent of the classical bits of an ordinary computer, but qubits differ in several key ways [4]:

(I) **Qubits do not always have a definite value**. They can be 0 and 1 at the same time. This is called superposition. In this superposition, the qubit has a probability for the 0 and 1 state; the exact state cannot be known at this moment. When the qubit is measured, only a 0 or 1 will be observed; this action is called ”collapsing”.

(II) **Qubits can not be copied and read without changing their value.** Because of the No-cloning theorem [10], an unknown quantum state can not be perfectly copied before measurement. An identical copy is not possible without measuring the state first. The qubit must collapse first to obtain the value; after that, the qubit can then be copied or read.

(III) **Reading a qubit can have an effect on another quantum bit.** This is called entanglement.

3.1.1. Mathematical model

A mathematical representation of the qubit’s state can be found in Equation 3.1 [4].

$$|\psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle\right) \tag{3.1}$$

Here $|\psi\rangle$ represents the state of the qubit in spherical coordinates, $|0\rangle$ is the 0 state, and $|1\rangle$ is the 1 state. These states can be seen as vectors, so $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$. This way of representing the vectors is called **bra-ket notation** [11]. This representation also indicates that the quantum state lies in a complex vector space. Further, γ is the global phase and φ the relative phase (see Subsection 3.1.3). The position of the state is determined by θ and φ, which are bounded to $[0, \pi]$ and $[0, 2\pi)$, respectively. Here θ refers to the polar angle, and φ refers to the azimuthal angle. A visualisation can be seen in Figure 3.1.

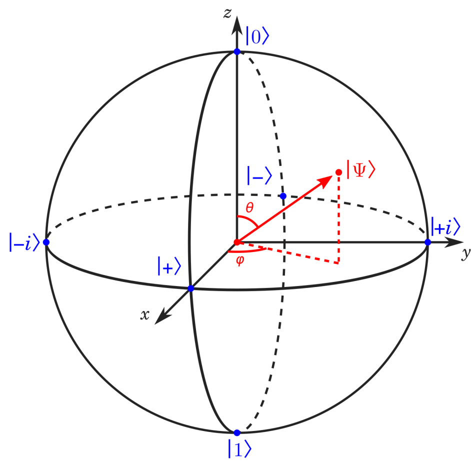


Figure 3.1: The Bloch Sphere with all axes shown [12]

For simplicity, Equation 3.1 is often split into two parts [4]:

$$\alpha = e^{i\gamma} \cos \frac{\theta}{2} \tag{3.2}$$

$$\beta = e^{i(\varphi+\gamma)} \sin \frac{\theta}{2} \tag{3.3}$$

Combining Equations 3.2 and 3.3 gives $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$.

3.1.2. Superposition and measurement

As discussed in Item (I) of Subsection 3.1, qubits can be in a superposition. This means the qubit has both a 0 and a 1 component. When measuring the qubit, the chance that it collapses to 0 is $|\alpha|^2 = \left(\cos \frac{\theta}{2}\right)^2$ and the chance that it collapses to 1 is $|\beta|^2 = \left(\sin \frac{\theta}{2}\right)^2$ [4]. The phase would not make a difference in this case, since the absolute value is taken. Note that $|\alpha|^2 + |\beta|^2 = 1$, this means that the total probability that either a 0 or a 1 is measured must be 1.

3.1.3. Phase

Normally, the quantum state is written with the global phase factor $e^{i\gamma}$ as shown in Equation 3.1. Here γ represents the global phase, and the φ represents the relative phase. When the quantum state is measured, the global phase does not affect the outcome [13]. Since the absolute value is taken of the two probability amplitudes α and β, the results are the same as without the global phase. As the Equations 3.4 and 3.5 show, the global phase does not change the quantum state measurement. For this reason, the global phase can be ignored. The simplified quantum state expression without global phase is shown in Equation 3.6.

$$|\alpha|^2 = |e^{i\gamma}\alpha|^2 \tag{3.4}$$

$$|\beta|^2 = |e^{i\gamma}\beta|^2 \tag{3.5}$$

$$|\psi\rangle = \cos \frac{\theta}{2}|0\rangle + e^{i\varphi} \sin \frac{\theta}{2}|1\rangle \tag{3.6}$$

The relative phase only changes the phase of the β component; this makes it possible to have a relative phase for two states when the α stays the same. States that differ in relative phase can have a physically observable difference when measuring the qubit. Thus, relative phase does not make two states equivalent, whereas with global phase, the states stay the same. [5].

3.2. Bloch Sphere

The Bloch sphere is an intuitive way to represent the state of a single qubit [4], which is shown in Figure 3.1. The θ shows the angle from the $|0\rangle$ state to the $|1\rangle$ state on the **X-Z plane**. And, φ shows the angle on the **X-Y plane**. The states that lie on the X and Y axis are listed below:

$$|+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \quad |-\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix}, \quad |i\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{bmatrix}, \quad |-i\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-i}{\sqrt{2}} \end{bmatrix} \tag{3.7}$$

The Bloch sphere maps all relevant quantities (relative phase and probability amplitudes), but discards the global phase. Because the global phase is not accounted for in the Bloch sphere, Equation 3.6 is used to define the state on the Bloch sphere. Due to its clarity and ease of interpretation, the Qbead uses the Bloch sphere to visualise its internal state. One more thing to note about the Bloch sphere, orthogonal states are antipodal (opposite of the centre), not at 90°[4].

3.3. Quantum Gates

The state represented in a Bloch sphere can be manipulated by applying quantum gates to the qubit [5], [14]. They change the state of one or more qubits, similar to classical logic gates, but they operate differently. Quantum gates can be divided into two types: single-qubit gates and multiple-qubit gates. Single-qubit gates are generally simpler and change the state vector of only a single qubit. One of the advantages of the Bloch sphere is that these single-qubit gates can be viewed as rotations around an axis on this sphere.

3.3.1. Common gates

The most common single-qubit gates used in quantum computing are the Pauli-X, -Y, and -Z gates, and the Hadamard gate [15], [16]. The Pauli gates flip the state with respect to the specified axis. For example, the Pauli-X gate rotates the state 180 degrees around the X axis, seen in Figure 3.2. The Hadamard gate is different from the other gates; it makes it possible to put a qubit into a superposition, seen in Figure 3.3. For example, when the qubit is in state $|0\rangle$, by applying the Hadamard gate, the state goes to the $|+\rangle$ state. The Hadamard gate can also be seen as a 180-degree rotation around the diagonal axis that lies in the X-Z plane. One property of quantum gates is that they are unitary, which implies they are reversible [3].

3.3.2. Matrices

One of the common ways to mathematically model these gates is with matrix transformations to the vector representation of the state. In Equation 3.9, the gates can be seen in matrix form. These gates can then be applied through matrix multiplication. The state vector can be rewritten as a vector shown in Equation 3.8. Multiplying the X-gate from Equation 3.9 with Equation 3.8 gives the result shown in Equation 3.10.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{3.8}$$

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.9}$$

$$X \cdot (|\psi\rangle) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} \tag{3.10}$$

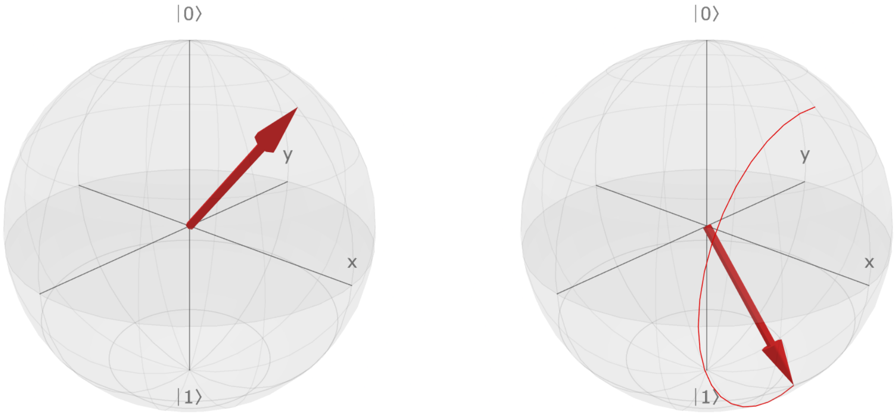


Figure 3.2: Pauli-X gate execution [17]

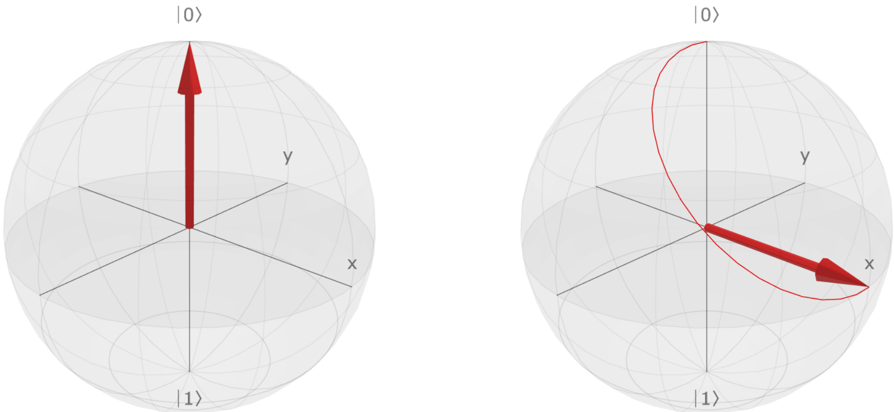


Figure 3.3: Hadamard gate execution [17]

4
Classes

This chapter describes the additional classes that have been added to the main header file of the Qbead's software. These classes are needed for the development of the future operations that were added to the Qbead.

4.1. Coordinates Class

The previous version of the software already had a clever system to determine which LED is illuminated on the Qbead. The Qbead was arranged into legs and sections. The ”legs” refer to the legs of the PCB, and the ”sections” relate to the LEDs on one leg. The PCB can be seen in Figure 4.1. This system was fully customisable when changing the number of legs and sections. However, this system was limited in terms of expansion, and it has the problem that the LED density around the equator of the sphere is lower than that of the rest of the sphere. Looking for alternative designs for the PCB would possibly step away from this ”leg and section”, and would require a different way to choose the LEDs.

4.1.1. Cartesian vector

The new coordinate class that has been implemented in the software addresses this problem by introducing a coordinate class with Cartesian vectors. Each state is set as a 3D vector (x, y, z); the orientation is the same as the Bloch sphere in Figure 1.1a. Also, each LED is in the new software set as a 3D vector, and then all these LEDs are put in an LED position map. This system is more flexible than the old system when working with different PCB designs. However, for each new PCB design

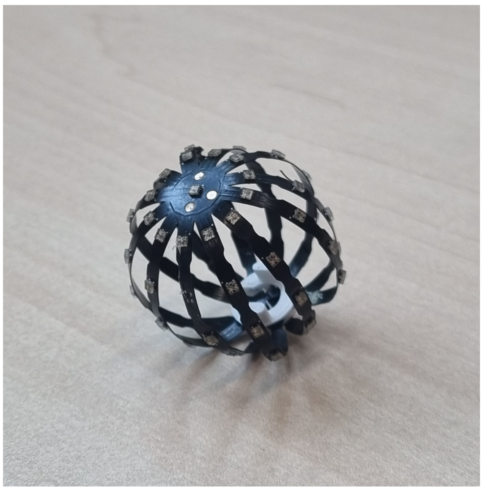


Figure 4.1: Flexible PCB of the Qbead

for the Qbead, a new LED mapping must be done. Nevertheless, it would be a better way for future development to continue with an LED mapping, since it uses a universal system now to control the LEDs. In addition to the benefits, the new class supports the development of future operations, such as the gates and collapse function. These functions can read out the coordinates with this class, which is made easier currently. The exact position of an LED is known, which was harder to tell with the previous system.

4.1.2. LED control

How each LED is shown on the sphere was previously done with the legs and sections. Similar to the Bloch sphere in Section 3.1, θ and φ are used for Qbead to determine the state vector. The θ and φ are split into sections depending on the number of LEDs. This would divide the sphere into square sections. When a certain reference state vector lies in one of the squares, the LED that corresponds to that square will be lit up. The Coordinate class can also work with this system because it has helper functions to get the φ and θ of the coordinate. Despite this, the new code also uses coordinates for the LEDs because it makes it more generic, and with this system, it is also possible to calculate the second closest LED. This can be used to show a way of decoherence, for example.

The new way to control the LED is done by measuring the distance between two coordinates. As mentioned in the last section, each LED has a fixed coordinate. The distance between a reference vector can be calculated for each LED. The LED that has the shortest distance to the reference vector will only be switched on. There is also an option to light more LEDs; this can be used to make motion smoother or to show decoherence.

4.2. Quantum State Class

A new class was implemented to keep track of the quantum state of the Qbead. This class was not present in the previous version of the code. This class keeps track of a coordinate with the Coordinates class (see Section 4.1). It also has helper functions to perform gates on the current state and collapse the state.

4.2.1. Collapsing

To replicate the real behaviour of a qubit, a collapse function was added to make measurement of the qubit possible (see Requirement **(A4)**). When measuring the state of a qubit, the state collapses to either 0 or 1 (see Subsection 3.1.2). The calculations are based on a qubit's behaviour when collapsing. The probability of the $|0\rangle$ state is compared to a randomly generated number from 0 to 100. First, the probability of $|0\rangle$ is scaled to [0,100] as well. Then, if this randomly generated number is less than the value of $|0\rangle$, it will collapse to $|0\rangle$, and otherwise it will collapse to $|1\rangle$. Then, the coordinate is set to either (0, 0, 1) or (0, 0, -1), which represents the 0 and 1 state, respectively.

4.2.2. Gates

One of the main requirements for the Qbead is to show the single qubit gates (see Requirement **(A3)**). These are the Pauli-X, -Y, and -Z gates, as well as the Hadamard gate (see Section 3.3).
When a 180-degree rotation around the axis of a circle is done, the state is flipped around that axis. This flipping is the same as taking the inverse of the other two axes. Thus, a simple solution for a Pauli gate would be to negate the two other axes. So, for example, performing a Pauli-X gate will transform the coordinate (x, y, z) into $(x, -y, -z)$. For the Hadamard gate, the X and Z axes can be swapped. This results in a 180-degree rotation around the XZ-axis. However, for implementing animations, the whole path of the rotations needed to be known. The old approach only calculated the position at the end of the rotation, which resulted in an incorrect animation being shown. A different approach was to implement this with matrices (see Subsection 3.3.2). The rotation for all the gates mentioned previously can be done with matrix multiplications. To compute the path of the gate matrices, these matrices can be modified into rotation operators. These are constructed via Equation 4.1 [5], where 'I' stands for the identity matrix and 'G' for the gate matrix defined in Subsection 3.3.2 corresponding to the Rotation axis. Filling in the gate matrices results in Rotation operators A.1, A.2, A.3 and A.4. The matrix operations are computed using the EIGEN C++ library. To show the animations, the Qbead uses the QuantumState class twice, once for the actual state of the qubit and once for the visual state

of the system, which is calculated by applying the gate matrix in small intervals of θ to get a smooth animation of the rotation.

$$R(\theta) = cos(\theta/2) * I - isin(\theta/2) * G \tag{4.1}$$

4.3. Qbead Class

The Qbead class serves as an abstraction for all the hardware functions of a Qbead. Every Qbead class instance should correspond to a physical Qbead. The initial form of the Qbead's software already had a lot of functions implemented. These functions implement the LED control using "legs" and "sections" discussed in Subsection 4.1.2, initialising the firmware, reading out the accelerometer, and using the Bluetooth functions of the XIAO nRF52840 microcontroller. This was also later rewritten for the XIAO ESP32S3.

4.3.1. User input

One of the main requirements of the Qbead class is to let people operate the Qbead by themselves when holding it in their hands. To let people do operations on the Qbead, it needs to be able to read inputs in some way. The only way to read physical inputs with the Qbead is by using the IMU present on the Qbead hardware.

There are three possible ways to detect user inputs: rotating, shaking, and tapping. There are four different qubit gates implemented. There are also gestures to collapse the state and to get to a random state. This results in six different inputs that need to be differentiated. The following methods can be used for user input:

- Rotation of the Qbead in 3 axes using the gyroscope
- Tapping on the Qbead in 3 axes
- Shaking of the Qbead in 3 axes

These are sufficient inputs to map every required user input.

All three axes of the gyroscope are used to sense rotations. Since it senses the X, Y, and Z axes, these axes can be connected to the respective Pauli gates. This way, the user easily knows which gate is executed. To avoid confusion about which side the Qbead must be tapped for a certain function, tapping is used in 2 axes, on the poles (Z-axis) and all around the equator (XY-plane). This makes the usability more understandable for the users. Lastly, the user might not know the difference between the axes when shaking the Qbead. It is harder to differentiate the axes when detecting shaking compared to the other function, so shaking detection does not differentiate between axes and would only execute one function. In conclusion, 3 different gestures are mapped by rotating, 2 gestures by tapping and only 1 by shaking.

4.3.2. Rotation detection

To detect rotations, the gyroscope on the IMU can be used. There are a few possible ways to implement the detection:

- Integrate the gyroscope and look at the degrees rotated
- Set a threshold for the amplitude of the angular velocity
- Use a time filter and a threshold for the angular velocity

Integrating the gyroscope

The absolute rotation in degrees is calculated via integration of the gyroscope, or by looking at the magnetometer implemented on the new hardware[9]. This gives a way to couple a certain length of rotation to an input. Because a Pauli gate rotates the state 180 degrees, the action needed to perform the gate and the result are essentially the same. Because the user might want to look at the other side of the Qbead without performing a gate, turning the Qbead slowly should not trigger a rotation input. Thus, rotation inputs have to be fast, and users are likely to lose track of the amount of rotation.

Using angular velocity
Reading the angular velocity of the gyroscope would be a better choice in this case. To trigger the input, the user just has to rotate the Qbead quickly. The threshold is set to 11 radians per second. This is retrieved from testing, where it was found to strike a balance between the user easily being able to trigger it, but also not accidentally triggering the input. Some users might first have trouble finding the threshold, but with some practice, this is learned quickly.

Using a timefilter
To eliminate wrong inputs coming from shocks and shaking, the gyroscope is first put into a time filter function. This function was already present in the firmware. It filters high-frequency spikes in the gyroscope measurement by combining the old measurements with the new measurement, which makes the data more reliable.

4.3.3. Tap detection

The IMU of the XIAO nRF52840 has built-in tap detection. The IMU could be configured to enable tap detection. The tap detection had settings for the amount of force that needs to be applied, enabling double-tapping and detecting different axes for tapping. It uses an interrupt system, but the rest of the system uses polling for getting its data. Therefore, a counter was implemented that keeps track of the number of double taps. The system polls the counter to know if a double tap has happened.
For the XIAO ESP32S3, the IMU ICM-20948 was used. This IMU does not have native tap detection. Therefore, the tap detection was built with a small FSM as seen in Figure 4.2. The FSM has 4 states:

1. Nothing detected
2. 1 tap detected and waiting for debounce
3. 1 tap detected and not waiting for debounce
4. 2 taps detected

The FSM begins in state 1. If there is an increase in acceleration of more than 5 g/s, then it will go to the second state. After 50 ms, it automatically goes to the third state. This is called debouncing, and it prevents 1 tap from counting as 2 taps. When a new tap is detected within 400 ms of the initial tap, the FSM goes to state 4. From state 4, it will go to state 1 in the next clock cycle. When no new tap is detected, the FSM also goes to the first state again.

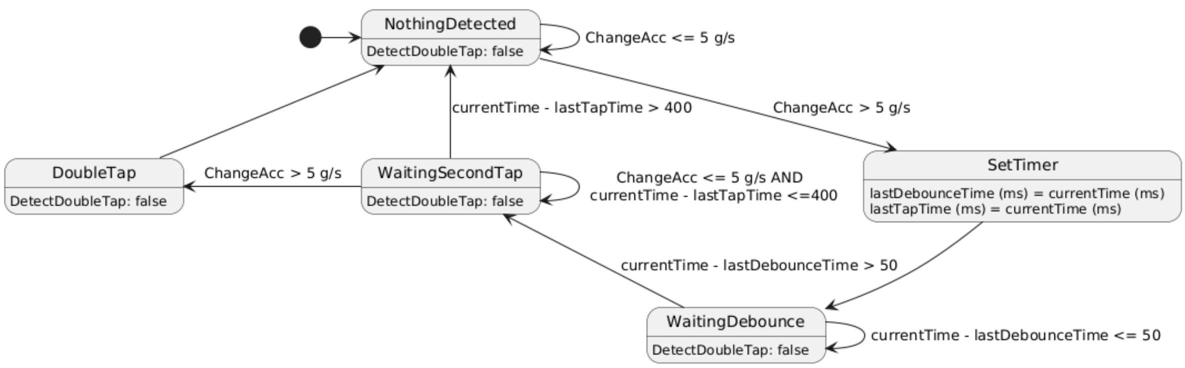


Figure 4.2: FSM diagram of tapping detection

4.3.4. Shaking

Shaking the Qbead results in spikes of high acceleration. This principle is used in the detection of shaking. The function should differentiate single shocks from shaking. To make this possible, the function checks if it keeps sensing shocks for a prolonged period. This is also implemented with a small FSM. This FSM can be seen in Figure 4.3. First, the total acceleration is calculated by taking the length of the gyroscope vector. The threshold for strong enough shakes to trigger the FSM is set to 3G force. If this is sensed, the FSM goes into the CheckShaking state, and the Qbead starts counting in milliseconds when it enters this state. After 300 milliseconds, it starts checking if the shocks are still present. If this is the case, it will trigger the action tied to shaking. It will check for another 500 milliseconds, and if in that time no other shock is sensed, the FSM goes back to the Idle state, and no action is triggered.

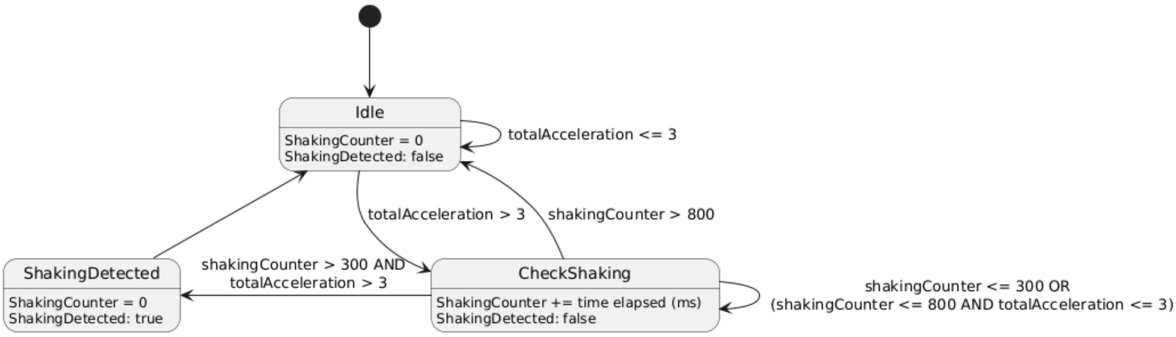


Figure 4.3: FSM diagram of shaking detection

4.3.5. BLE changes for ESP32S3

Currently, the BLE, standing for Bluetooth Low Energy, on the Qbead does not have a real function and can only be used to read out IMU values. One of the future features planned for the Qbead is to allow for entanglement and multiple Qbeads connecting. For this, BLE is needed, and that is why the basics are already implemented. On the XIAO nRF52840, the BLE was already implemented using a package from Adafruit. It uses characteristics, which are essentially packets of information holding a value of a certain variable. On the Qbead, there are four of these: the colour of the Qbead, the coordinates, the accelerometer readout, and the gyroscope readout. These characteristics can be read or written to depending on the configuration. The characteristics are packaged into a service. Every characteristic and service has its own UUID(universally unique identifier), a string of unique characters acting as an identifier. This is then advertised, which means the Qbead broadcasts the data through the air.

The old package that implemented the BLE on the XIAO nRF52840 microcontroller is not supported on the XIAO ESP32S3. This means that the BLE code had to be rewritten using the BLE package built into the ESP32. Similar to the old package, this package uses services and characteristics. The following things were changed compared to the XIAO nRF52840:

- The Qbead now creates its own server, which could hold multiple services (the Qbead still only has one). The new hierarchy is shown in Figure 4.4.
- The server, service, and characteristic classes are now stored in pointer variables and created using the pointer from the class above it.
- Every characteristic now has a descriptor that enables notification for the readouts and gives every characteristic a name.
- The UUID format was changed to a version 4 UUID format because the new package could not handle the old format.

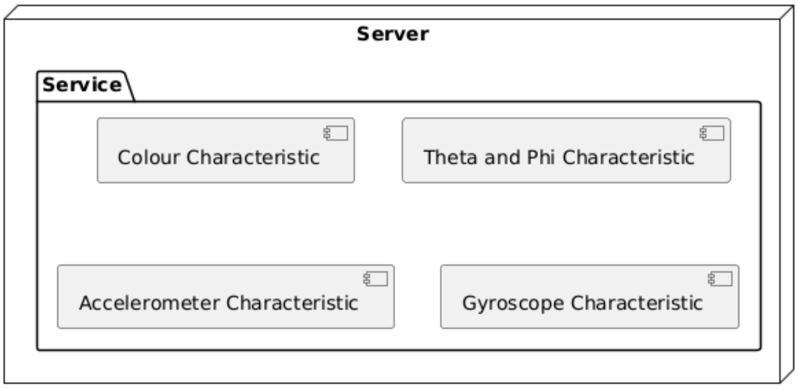


Figure 4.4: BLE hierarchy of the Qbead

5
Applications of the Qbead

A few experiments were designed to teach people about quantum computing. From an educational standpoint, all experiments must have a learning goal. Those experiments will be discussed in this chapter.

5.1. UI/UX

To indicate the axes on the Qbead, which helps to find the orientation of the Qbead; there are 7 LEDs that always light up (see Requirement **(B1)**). First, the 0 state vector is shown in red, and the 1 state vector is shown in green. These colours were chosen because those are the colours that people associate with ”on” and ”off”. At the equator, there are four equally spaced LEDs that are shown in blue to represent the X and Y axes, which are also the $|+\rangle$, $|-\rangle$, $|i\rangle$, and $|-i\rangle$ states. Lastly, a white LED is shown to indicate the current quantum state.

5.2. Gates

To learn about gates, an experiment was set up. In this experiment, people can apply the Pauli gates by turning the Qbead around an axis. For example, if someone turns the Qbead around its Z-axis, the Pauli-Z gate will be applied. This will be done by showing an animation where the state travels around the Z-axis. The Hadamard gate can also be applied. This can be done by double-tapping on either the X or Y-axis. See Subsection 4.2.2 for the implementation of these gates. Double-tapping the Z-axis collapses the state to 0 or 1. The Hadamard gate has the same animation as the Pauli gates. Collapsing does not have an animation since qubits collapse almost instantly. However, after collapsing, it has a cool-down of 2 seconds to prevent accidental gestures. There is also a way to get to a random state. This can be done by shaking the Qbead. The rotation operations have a certain threshold, which prevents the Qbead from activating without triggering any of the gates when inspecting the Qbead. All the implementation can be read back in Section 4.3.

5.3. Decoherence

Decoherence is a phenomenon in qubits where, while in superposition, the complex quantum state gradually decays over time, leading to a loss of coherence [18], [19]. This means that the phase of the state has some drift. Since the superposition state is sensitive to external interference, even minimal interaction with the external environment can induce decoherence [20], [21]. For this project, a decoherence experiment was also created. In this experiment, the gates of Section 5.2 can still be applied in the same way. The only difference is that decoherence is applied to the state.

There are a few ways to show this experiment on the Qbead. First, the decoherence can be shown by spreading out the state. So, around the quantum state, more LEDs would turn on when no actions are performed. With the support of the coordinate system, the nth closest LED can easily be calculated. The only problem is that this way of implementing the decoherence is not fully generic. When the Qbead has a higher LED resolution, the decoherence will be slower.

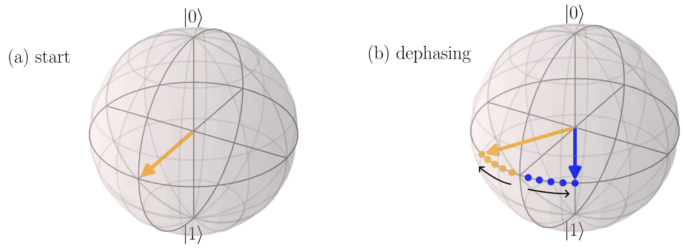


Figure 5.1: A qubit dephasing to both side from its original state [22]

Another method, the one that is currently used, is to give the phase a random and positive speed. The state dephases from its original state. In Figure 5.1, an example of dephasing can be seen. For this method, the Qbead only dephases to one side, since that is the easiest model of decoherence.

This way is simpler than the previous model. It makes it easier to understand for people without any knowledge about quantum, and is also easy to display. The old state is shown in purple, and the actual state is still shown in white on the Qbead.

5.3.1. Dynamic Decoupling

Dynamic decoupling is a way to counteract the quantum state error on a qubit. By applying a sequence of gates on the qubit, it reduces the effect of decoherence on the qubit [23], [24]. Dynamic decoupling is one of the simpler methods to suppress the quantum error [25], which is why it would be a great addition to the Qbeads software to teach more knowledge about the qubit.

With the gate and decoherence implemented in the software, there was no need to make drastic changes to the software. A separate sketch was made to show the decoherence on the Qbead, after which the user can perform gates to replicate dynamic decoupling. With this dynamic decoupling experiment, users will learn that rotating the Qbead in a certain manner will cause the quantum state to become stable.

6
Testing

In this chapter, the usability of the Qbead is tested to see if the Programme of Requirements is met. The Programme of Requirements can be found in Chapter 2. The current changes made to the software framework are published on the public GitHub repository of the Qbead. The whole code can also be seen in Appendix B.

6.1. Installation

To establish a connection with the Qbead, the Qbead must be connected via USB or Bluetooth. However, the Bluetooth capabilities are currently limited, so during this project, everything was done with a USB connection. The Seeed XIAO nRF52840 Sense and the XIAO ESP32S3 use the Arduino IDE for writing, compiling, and uploading the code. Before the Qbead's software framework can be uploaded to the board, some required packages must be installed first. The following packages must be installed:

- **Adafruit Neopixel**, for the communication between the LEDs and board.
- **Seeed Arduino LSM5DS3**, to read out sensor data of the onboard IMU of the nRF52840.
- **ArduinoEigen**, to support matrix calculations.
- **SparkFun_ICM-20948_ArduinoLibrary**, to controll the IMU of the ESP32.
- **ICM20948_WE**, to set the gyro range of the IMU of the ESP32.

When the software is successfully uploaded to the board, Arduino sketches can be uploaded as well to show the decoherence experiment, for example.

6.2. User input measurements

6.2.1. Responsiveness

The Qbead is designed to process user inputs within 100 ms after a gesture is executed. The short processing time makes sure there is minimal time delay between performing a gesture and getting an output. This improves the user experience, since it feels close to real-time. The decoherence experiment has a cycle time of 20 ms. Which means that the main loop is executed every 20 ms. This is the most extensive experiment that was written, so the cycle time of the other experiments is expected to be faster or the same. Shaking and rotating are both executed in the same loop cycle as they are detected. This operation would take 20 ms. Tapping with the interrupt system may take a little bit more than 1 clock cycle, which results in a maximum cycle time of 40 ms. All operations and experiments are well under the threshold of 100 ms.

6.2.2. Gesture Detection Accuracy

The Qbead aims to accurately identify rotation, shaking, and tapping movements. The accuracy of recognising the correct movement should be at least 90%. It is hard to measure this accuracy. Since the user is prone to making mistakes. So, for example, tapping should work as intended, but it is possible the user has not tapped with enough force. Or tapping off-axis will sometimes result in registering both axes. Then the software cannot determine which axis is meant. Therefore, the user needs to tap on one of the highlighted axes. The rotation gesture has big margins for error, since the user can easily rotate the Qbead around an axis with the help of the axis indicators. So, it is hard to imagine that two rotations are measured at once. When shaking, it is important not to rotate the Qbead around an axis, as rotations will trigger gate operations as well and interfere with the shaking feature. But with these instructions, the accuracy for rotating and shaking is essentially 100%. The tapping, however, sometimes just detects 2 axes even if tapped very carefully, but that is only once every 30 times during testing.

6.2.3. Conclusion

After testing, the Qbead software framework shows consistent and adequate results. With state updates processed within 100 ms. Further, the battery life is performing well above expectations, with battery life up to 90 minutes. Lastly, the gesture recognition accuracy meets the specifications of 90% accuracy. Therefore, the Requirements **(A5)** and **(C3)** - **(C5)** are met.

7

Discussion and Conclusion

In this chapter, the outcomes of the project are evaluated based on the established Programme of Requirements (PoR). The achieved results, encountered limitations, and recommendations for future improvements are discussed in detail.

7.1. Conclusion

In this project, the software was greatly improved. A solid, generic, and extendable foundation was built. A few experiments were implemented on this foundation. Both the code that was built for the XIAO nRF52840 and the code developed for the XIAO ESP32S3 do meet all mandatory requirements. The XIAO nRF52840 also meets all trade-off requirements. However, at the time of writing, the code for the XIAO ESP32S3 does not meet all trade-off requirements. The tapping and shaking don't meet the 90% accuracy threshold. The reason this works well on the XIAO nRF52840 but not on the XIAO ESP32S3 is that the XIAO nRF52840 has built-in tap detection. Therefore, the tap detection on the XIAO ESP32S3 was self-built, which was not equally reliable as the one on the XIAO nRF52840.

The software was developed using the Incremental model. First, the PoR was created. Then, a requirement or feature was picked, and that was implemented and tested. This model makes it very easy to work with a team because every team member can work on their own cycle. In the first few cycles, the focus was on improving the foundation of the code. So, for example, implementing the Coordinate class. After the foundation was made, the focus went on implementing the experiments. When implementing some experiments, the foundation still needed some changes, but those were relatively minor.

7.2. Discussion

As already discussed in the conclusion, the tap detection of the XIAO ESP32S3 does not work that well. We still want to improve that ourselves after this paper is finished.
Due to a misunderstanding and a difference in opinion in the beginning of the project, we first tried to make the Qbeads axes fixed to the outside world so that the Z axis always points to the sky/ground. We spent a lot of time on this, and this could have been prevented with more frequent meetings and this would have left more time to develop other functions.

Recommendations for Future Work:

- Further optimisation of the gesture detection.
- Exploration of magnetometer integration for enhanced spatial awareness or more advanced gestures.
- Implementation of entanglement between Qbeads.
- Development of a dedicated Bluetooth interface, accompanied by a control app or website.

A

Formulas

A.1. Matrices used for single qubit gates

$$R_x(\theta) = cos(\theta/2) * I - isin(\theta/2) * \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} cos(\theta/2) & -isin(\theta/2) \\ -isin(\theta/2) & cos(\theta/2) \end{bmatrix} \tag{A.1}$$

$$R_y(\theta) = cos(\theta/2) * I - isin(\theta/2) * \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = \begin{bmatrix} cos(\theta/2) & -sin(\theta/2) \\ sin(\theta/2) & cos(\theta/2) \end{bmatrix} \tag{A.2}$$

$$R_z(\theta) = cos(\theta/2) * I - isin(\theta/2) * \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \tag{A.3}$$

$$R_H(\theta) = cos(\theta/2) * I - \frac{isin(\theta/2)}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} cos(\theta/2) - \frac{isin(\theta/2)}{\sqrt{2}} & -\frac{isin(\theta/2)}{\sqrt{2}} \\ -\frac{isin(\theta/2)}{\sqrt{2}} & cos(\theta/2) + \frac{isin(\theta/2)}{\sqrt{2}} \end{bmatrix} \tag{A.4}$$

B
Software framework

B.1. Source code XIAO Seeed nRF52870

```
#ifndef QBEAD_H
#define QBEAD_H

#include <Arduino.h>
#include <Adafruit_NeoPixel.h>
#include <LSM6DS3.h>
#include <math.h>
#include <ArduinoEigen.h>
#include <bluefruit.h>

using namespace Eigen;

// default configs
#define QB_LEDPIN 10
#define QB_PIXELCONFIG NEO_BRG + NEO_KHZ800
#define QB_IMU_ADDR 0x6A
#define QB_IX 1
#define QB_IY 0
#define QB_IZ 2
#define QB_SX 0
#define QB_SY 0
#define QB_SZ 1
#define GYRO_GATE_THRESHOLD 8
#define QB_PIXEL_COUNT 62
#define QB_MAX_PRPH_CONNECTION 2
#define T_ACC 100000
#define T_GYRO 10000

const uint8_t QB_UUID_SERVICE[] =
{0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6,0x1f,0x0c,0xe3};
const uint8_t QB_UUID_COL_CHAR[] =
{0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6+1,0x1f,0x0c,0xe3};
const uint8_t QB_UUID_SPH_CHAR[] =
{0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6+2,0x1f,0x0c,0xe3};
const uint8_t QB_UUID_ACC_CHAR[] =
{0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6+3,0x1f,0x0c,0xe3};
const uint8_t QB_UUID_GYR_CHAR[] =
{0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6+4,0x1f,0x0c,0xe3};

const uint8_t zerobuffer20[] = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0
    x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
const std::complex<float>i(0, 1);

// TODO manage namespaces better
// The setPixelColor switches blue and green
static uint32_t color(uint8_t r, uint8_t g, uint8_t b) {
  return ((uint32_t)r << 16) | ((uint16_t)b << 8) | g;
```

```
}

static uint8_t redch(uint32_t rgb) {
  return rgb >> 16;
}

static uint8_t greench(uint32_t rgb) {
  return 0x0000ff & rgb;
}

static uint8_t bluech(uint32_t rgb) {
  return (0x00ff00 & rgb) >> 8;
}

uint32_t colorWheel(uint8_t wheelPos) {
  wheelPos = 255 - wheelPos;
  if (wheelPos < 85) {
    return color(255 - wheelPos * 3, 0, wheelPos * 3);
  }
  if (wheelPos < 170) {
    wheelPos -= 85;
    return color(0, wheelPos * 3, 255 - wheelPos * 3);
  }
  wheelPos -= 170;
  return color(wheelPos * 3, 255 - wheelPos * 3, 0);
}

uint32_t colorWheel_deg(float wheelPos) {
  return colorWheel(wheelPos * 255 / 360);
}

float sign(float x) {
  if (x > 0) return +1;
  else return -1;
}

// z = cos(t)
// x = cos(p)sin(t)
// y = sin(p)sin(t)
// Return the angle in radians between the x-axis and the line to the point (x, y)
float phi(float x, float y) {
  return atan2(y, x);
}

float phi(float x, float y, float z) {
  return phi(x, y);
}

float theta(float x, float y, float z) {
  float ll = x * x + y * y + z * z;
  float l = sqrt(ll);
  float theta = acos(z / l);
  return theta;
}

bool checkThetaAndPhi(float theta, float phi) {
  return theta >= 0 && theta <= 180 && phi >= 0 && phi <= 360;
}

void connect_callback(uint16_t conn_handle)
{
  // Get the reference to current connection
  BLEConnection* connection = Bluefruit.Connection(conn_handle);

  char central_name[32] = { 0 };
  connection->getPeerName(central_name, sizeof(central_name));

  Serial.print("[INFO]{BLE} Connected to "); // TODO take care of cases where Serial is not
    available
  Serial.println(central_name);
}
```

```

// In rads
void sphericalToCartesian(float theta, float phi, float& x, float& y, float& z)
{
  // Normalize yaw to be between 0 and 2*PI
  phi = fmod(phi, 2 * PI);
  if (phi < 0)
  {
    phi += 2 * PI;
  }
  if (!checkThetaAndPhi(theta * 180 / PI, phi * 180 / PI))
  {
    Serial.print("Theta or Phi out of range when creating coordinates class, initializing as
    1");
    Serial.print("Theta: ");
    Serial.print(theta);
    Serial.print("Phi: ");
    Serial.println(phi);
    x = 0;
    y = 0;
    z = 1;
    return;
  }

  x = sin(theta) * cos(phi);
  y = sin(theta) * sin(phi);
  z = cos(theta);
}

// Tap detection
LSM6DS3 myIMU(I2C_MODE, QB_IMU_ADDR); // Create an instance of the IMU
uint8_t interruptCount = 0; // Amount of received interrupts
uint8_t prevInterruptCount = 0; // Interrupt Counter from last loop

void setupTapInterrupt() {
  uint8_t error = 0;
  uint8_t dataToWrite = 0;

  // Double Tap Config
  myIMU.writeRegister(LSM6DS3_ACC_GYRO_CTRL1_XL, 0x60);
  myIMU.writeRegister(LSM6DS3_ACC_GYRO_TAP_CFG1, 0x8E);// INTERRUPTS_ENABLE, SLOPE_FDS
  myIMU.writeRegister(LSM6DS3_ACC_GYRO_TAP_THS_6D, 0x6C);
  myIMU.writeRegister(LSM6DS3_ACC_GYRO_INT_DUR2, 0x7F);
  myIMU.writeRegister(LSM6DS3_ACC_GYRO_WAKE_UP_THS, 0x80);
  myIMU.writeRegister(LSM6DS3_ACC_GYRO_MD1_CFG, 0x08);
}

void int1ISR()
{
  interruptCount++;
}

namespace Qbead {

class Coordinates
{
public:
  Vector3d v;

  Coordinates(float argx, float argy, float argz)
  {
    v = Vector3d(argx, argy, argz);
    v.normalize();
  }

  // In rads
  Coordinates(float theta, float phi)
  {
    float x, y, z = 0;
    sphericalToCartesian(theta, phi, x, y, z);
    v = Vector3d(x, y, z);
```

```
  }

  Coordinates(Vector3d vector)
  {
    v = vector;
    v.normalize();
  }

  // In rads
  float theta()
  {
    return acos(v(2));
  }

  // In rads
  float phi()
  {
    return atan2(v(1), v(0));
  }

  Vector2cf stateVector2D()
  {
    std::complex<float> alpha = cos(theta()/2);
    std::complex<float> beta = exp(i*phi()) * sin(theta()/2);
    return {alpha, beta};
  }

  float dist(Vector3d other) const
  {
    Vector3d diff = v - other;
    return diff.norm();
  }

  void set(float argx, float argy, float argz)
  {
    v = Vector3d(argx, argy, argz);
    v.normalize();
  }

  // in rads
  void set(float theta, float phi)
  {
    float x, y, z = 0;
    sphericalToCartesian(theta, phi, x, y, z);
    v = Vector3d(x, y, z);
  }

  void set(Vector3d vector) {
    v = vector;
    v.normalize();
  }

  // in rads
  void setTheta(float theta)
  {
    set(theta, phi());
  }

  // in rads
  void setPhi(float phi)
  {
    set(theta(), phi);
  }
};

class QuantumState
{
private:
  Coordinates stateCoordinates;

public:
```

```
QuantumState(Coordinates argStateCoordinates) : stateCoordinates(argStateCoordinates) {}
QuantumState() : stateCoordinates(0, 0, 1) {}

void setCoordinates(Coordinates argStateCoordinates)
{
  stateCoordinates.set(argStateCoordinates.v);
}

Coordinates getCoordinates()
{
  return stateCoordinates;
}

void collapse()
{
  const float a = (stateCoordinates.v(2) + 1) / 2; // probability of measuring |0>
  if (a < 0.0001) {
    stateCoordinates.set(0, 0, -1);
    return;
  } else if (a > 0.9999) {
    stateCoordinates.set(0, 0, 1);
    return;
  }
  const bool is1 = random(0, 100) <= a * a * 100;
  this->stateCoordinates.set(0, 0, is1 ? 1 : -1);
}

void applyGate(Matrix2cf gate)
{
  Vector2cf stateVector = stateCoordinates.stateVector2D();
  stateVector = gate * stateVector;
  stateVector.normalize();
  stateCoordinates.set(2*acos(abs(stateVector.x())), arg(stateVector.y()) - arg(stateVector.x()));
}

void applyGateType(uint16_t gateType, float rotationDegree = PI)
{
  switch (gateType)
  {
  case 1:
    gateX(-rotationDegree);
    break;
  case 2:
    gateY(-rotationDegree);
    break;
  case 3:
    gateZ(rotationDegree);
    break;
  case 4:
    gateX(rotationDegree);
    break;
  case 5:
    gateY(rotationDegree);
    break;
  case 6:
    gateZ(-rotationDegree);
    break;
  case 7:
    gateH(rotationDegree);
    break;
  default:
    break;
  }
}

// Rotate PI around the x axis
void gateX(float rotationDegree = PI)
{
  Matrix2cf gateMatrix;
  gateMatrix << cos(rotationDegree / 2.0f), -sin(rotationDegree / 2.0f) * i,
```

```
        -sin(rotationDegree / 2.0f) * i, cos(rotationDegree / 2.0f); // global phase differs from pauli gates but this doesn't matter for bloch sphere
    applyGate(gateMatrix);
  }

  // Rotate PI around the y axis
  void gateZ(float rotationDegree = PI)
  {
    Matrix2cf gateMatrix;
    gateMatrix << exp(-i * rotationDegree / 2.0f), 0,
        0, exp(i * rotationDegree / 2.0f);
    applyGate(gateMatrix);
  }

  // Rotate PI around the z axis
  void gateY(float rotationDegree = PI)
  {
    Matrix2cf gateMatrix;
    gateMatrix << cos(rotationDegree / 2.0f), -sin(rotationDegree / 2.0f),
        sin(rotationDegree / 2.0f), cos(rotationDegree / 2.0f);
    applyGate(gateMatrix);
  }

  // Rotate PI around the xz axis
  void gateH(float rotationDegree = PI)
  {
    Matrix2cf gateMatrix;
    gateMatrix << (cos(rotationDegree / 2.0f) - i * sin(rotationDegree / 2.0f) / sqrt(2.0f)), -i * sin(rotationDegree / 2.0f) / sqrt(2.0f),
        -i * sin(rotationDegree / 2.0f) / sqrt(2.0f), (cos(rotationDegree / 2.0f) + i * sin(rotationDegree / 2.0f) / sqrt(2.0f));
    applyGate(gateMatrix);
  }
};

class Qbead {
public:
  Qbead(const uint16_t pin00 = QB_LEDPIN,
        const uint16_t pixelconfig = QB_PIXELCONFIG,
        const uint8_t imu_addr = QB_IMU_ADDR)
      : imu(LSM6DS3(I2C_MODE, imu_addr)),
        pixels(Adafruit_NeoPixel(QB_PIXEL_COUNT, pin00, pixelconfig)),
        bleservice(QB_UUID_SERVICE),
        blecharcol(QB_UUID_COL_CHAR),
        blecharsph(QB_UUID_SPH_CHAR),
        blecharacc(QB_UUID_ACC_CHAR),
        blechargyr(QB_UUID_GYR_CHAR)
  {}

  static Qbead *singletoninstance; // we need a global singleton static instance because bluefruit callbacks do not support context variables -- thankfully this is fine because there is indeed only one Qbead in existence at any time

  LSM6DS3 imu;
  Adafruit_NeoPixel pixels;

  BLEService bleservice;
  BLECharacteristic blecharcol;
  BLECharacteristic blecharsph;
  BLECharacteristic blecharacc;
  BLECharacteristic blechargyr;

  float rbuffer[3], rgyrobuffer[3];
  float T_imu;                // last update from the IMU
  float T_freeze = 0;
  float T_shaking = 0;
  bool frozen = false; // frozen means that there is an animation in progress
  bool shakingState = false; // if ShakingState is 1 detected shaking and if shaking keeps happening randomising state
  QuantumState state = QuantumState(Coordinates(-0.866, 0.25, -0.433));
  Coordinates visualState = Coordinates(-0.866, 0.25, -0.433);
```

```
Vector3d gravityVector = Vector3d(0, 0, 1);
Vector3d gyroVector = Vector3d(0, 0, 1);
float yaw = 0;

float t_ble, p_ble; // theta and phi as sent over BLE connection
uint32_t c_ble = 0xffffff; // color as sent over BLE connection

// led map index to Coordinates
// This map is for the first version of the flex-pcb
Coordinates led_map_v1[62] = {
  Coordinates(-1, -0, -0),
  Coordinates(-0.866, 0, -0.5),
  Coordinates(-0.5, 0, -0.866),
  Coordinates(-0, 0, -1),
  Coordinates(0.5, 0, -0.866),
  Coordinates(0.866, 0, -0.5),
  Coordinates(1, 0, 0),
  Coordinates(-0.866, 0.25, -0.433),
  Coordinates(-0.5, 0.433, -0.75),
  Coordinates(-0, 0.5, -0.866),
  Coordinates(0.5, 0.433, -0.75),
  Coordinates(0.866, 0.25, -0.433),
  Coordinates(-0.866, 0.433, -0.25),
  Coordinates(-0.5, 0.75, -0.433),
  Coordinates(-0, 0.866, -0.5),
  Coordinates(0.5, 0.75, -0.433),
  Coordinates(0.866, 0.433, -0.25),
  Coordinates(-0.866, 0.5, 0),
  Coordinates(-0.5, 0.866, 0),
  Coordinates(-0, 1, 0),
  Coordinates(0.5, 0.866, 0),
  Coordinates(0.866, 0.5, 0),
  Coordinates(-0.866, 0.433, 0.25),
  Coordinates(-0.5, 0.75, 0.433),
  Coordinates(-0, 0.866, 0.5),
  Coordinates(0.5, 0.75, 0.433),
  Coordinates(0.866, 0.433, 0.25),
  Coordinates(-0.866, 0.25, 0.433),
  Coordinates(-0.5, 0.433, 0.75),
  Coordinates(-0, 0.5, 0.866),
  Coordinates(0.5, 0.433, 0.75),
  Coordinates(0.866, 0.25, 0.433),
  Coordinates(-0.866, -0, 0.5),
  Coordinates(-0.5, -0, 0.866),
  Coordinates(-0, -0, 1),
  Coordinates(0.5, -0, 0.866),
  Coordinates(0.866, -0, 0.5),
  Coordinates(-0.866, -0.25, 0.433),
  Coordinates(-0.5, -0.433, 0.75),
  Coordinates(-0, -0.5, 0.866),
  Coordinates(0.5, -0.433, 0.75),
  Coordinates(0.866, -0.25, 0.433),
  Coordinates(-0.866, -0.433, 0.25),
  Coordinates(-0.5, -0.75, 0.433),
  Coordinates(-0, -0.866, 0.5),
  Coordinates(0.5, -0.75, 0.433),
  Coordinates(0.866, -0.433, 0.25),
  Coordinates(-0.866, -0.5, -0),
  Coordinates(-0.5, -0.866, -0),
  Coordinates(-0, -1, -0),
  Coordinates(0.5, -0.866, -0),
  Coordinates(0.866, -0.5, -0),
  Coordinates(-0.866, -0.433, -0.25),
  Coordinates(-0.5, -0.75, -0.433),
  Coordinates(-0, -0.866, -0.5),
  Coordinates(0.5, -0.75, -0.433),
  Coordinates(0.866, -0.433, -0.25),
  Coordinates(-0.866, -0.25, -0.433),
  Coordinates(-0.5, -0.433, -0.75),
  Coordinates(-0, -0.5, -0.866),
  Coordinates(0.5, -0.433, -0.75),
```

```
  Coordinates(0.866, -0.25, -0.433),
};

static void ble_callback_color(uint16_t conn_hdl, BLECharacteristic* chr, uint8_t* data,
   uint16_t len) {
  Serial.println("[INFO]{BLE} Received a write on the color characteristic");
  singletoninstance->c_ble =  (data[2] << 16) | (data[1] << 8) | data[0];
  Serial.print("[DEBUG]{BLE} Received");
  Serial.println(singletoninstance->c_ble, HEX);
}

static void ble_callback_theta_phi(uint16_t conn_hdl, BLECharacteristic* chr, uint8_t* data
   , uint16_t len){
  Serial.println("[INFO]{BLE} Received a write on the spherical coordinates characteristic"
   );
  singletoninstance->t_ble = ((uint32_t)data[0])*180/255;
  singletoninstance->p_ble = ((uint32_t)data[1])*360/255;
  Serial.print("[DEBUG]{BLE} Received t=");
  Serial.print(singletoninstance->t_ble);
  Serial.print(" p=");
  Serial.println(singletoninstance->p_ble);
}

void startAccelerometer() {
  // BLE Characteristic IMU xyz accelerometer readout
  blecharacc.setProperties(CHR_PROPS_READ | CHR_PROPS_NOTIFY);
  blecharacc.setPermission(SECMODE_OPEN, SECMODE_OPEN);
  blecharacc.setUserDescriptor("xyz acceleration");
  blecharacc.setFixedLen(3*sizeof(float));
  blecharacc.begin();
  blecharacc.write(zerobuffer20, 3*sizeof(float));
}

void begin() {
  singletoninstance = this;
  Serial.begin(9600);
  for (int waitCount = 0; waitCount < 50; waitCount++)
  {
    if (Serial) {break;}
    delay(100);
  }

  pixels.begin();
  clear();
  setBrightness(10);

  Serial.println("[INFO] Booting... Qbead on XIAO BLE Sense + LSM6DS3 compiled on "
   __DATE__ " at " __TIME__);
  if (!imu.begin()) {
    Serial.println("[DEBUG]{IMU} IMU initialized correctly");
  } else {
    Serial.println("[ERROR]{IMU} IMU failed to initialize");
  }

  // BLE Peripheral service setup
  Bluefruit.begin(QB_MAX_PRPH_CONNECTION, 0);
  Bluefruit.setName("qbead | " __DATE__ " " __TIME__);
  Bluefruit.Periph.setConnectCallback(connect_callback);
  bleservice.begin();
  // BLE Characteristic Bloch Sphere Visualizer color setup
  blecharcol.setProperties(CHR_PROPS_READ | CHR_PROPS_WRITE);
  blecharcol.setPermission(SECMODE_OPEN, SECMODE_OPEN);
  blecharcol.setUserDescriptor("BSV rgb color");
  blecharcol.setFixedLen(3);
  blecharcol.setWriteCallback(ble_callback_color);
  blecharcol.begin();
  blecharcol.write(zerobuffer20, 3);
  // BLE Characteristic Bloch Sphere Visualizer spherical coordinate setup
  blecharsph.setProperties(CHR_PROPS_READ | CHR_PROPS_WRITE);
  blecharsph.setPermission(SECMODE_OPEN, SECMODE_OPEN);
  blecharsph.setUserDescriptor("BSV spherical coordinates");
```

```
  blecharsph.setFixedLen(2);
  blecharsph.setWriteCallback(ble_callback_theta_phi);
  blecharsph.begin();
  blecharsph.write(zerobuffer20, 2);
  // BLE Characteristic IMU xyz gyroscope readout
  blechargyr.setProperties(CHR_PROPS_READ | CHR_PROPS_NOTIFY);
  blechargyr.setPermission(SECMODE_OPEN, SECMODE_OPEN);
  blechargyr.setUserDescriptor("xyz gyroscope");
  blechargyr.setFixedLen(3*sizeof(float));
  blechargyr.begin();
  blechargyr.write(zerobuffer20, 3*sizeof(float));
  startBLEadv();

  // Tap detection
  setupTapInterrupt();
  pinMode(PIN_LSM6DS3TR_C_INT1, INPUT);
  attachInterrupt(digitalPinToInterrupt(PIN_LSM6DS3TR_C_INT1), int1ISR, RISING);
}

void clear() {
  pixels.clear();
}

void show() {
  pixels.show();
}

void setBrightness(uint8_t b) {
  pixels.setBrightness(b);
}

void setLed(Coordinates coordinates, uint32_t color, int leds = 1) {
  float theta = coordinates.theta() * 180 / PI;
  float phi = coordinates.phi() * 180 / PI;
  if (phi < 0) {
    phi += 360;
  }
  setBloch_deg(theta, phi, color, leds);
}

void showAxis() {
  setLed(Coordinates(1, 0, 0), color(0, 0, 122));
  setLed(Coordinates(-1, 0, 0), color(0, 0, 122));
  setLed(Coordinates(0, 1, 0), color(0, 0, 122));
  setLed(Coordinates(0, -1, 0), color(0, 0, 122));
  setLed(Coordinates(0, 0, 1), color(0, 255, 0));
  setLed(Coordinates(0, 0, -1), color(255, 0, 0));
}

// in rads
float getDistToLed(float theta, float phi, int index) {
  const Coordinates led = led_map_v1[index];
  const Coordinates reference(theta, phi);
  return led.dist(reference.v);
}

// Single bit is lit up on the Bloch sphere
void setBloch_deg(float theta, float phi, uint32_t c, int leds = 1) {
  int index[leds];
  float dist[leds];
  for (int i = 0; i < leds; i++) {
    index[i] = -1;
    dist[i] = 1000;
  }
  for (int i = 0; i < QB_PIXEL_COUNT; i++) {
    float d = getDistToLed(theta * PI / 180, phi * PI / 180, i);
    for (int j = 0; j < leds; j++) {
      if (d < dist[j]) {
        for (int k = leds - 1; k > j; k--) {
          index[k] = index[k - 1];
          dist[k] = dist[k - 1];
```

```
          }
          index[j] = i;
          dist[j] = d;
          break;
        }
      }
    }
    for (int i = 0; i < leds; i++) {
      if (index[i] != -1) {
        uint8_t r = redch(c);
        uint8_t g = greench(c);
        uint8_t b = bluech(c);
        float p2 =  pow(200, -dist[i]);
        pixels.setPixelColor(index[i], color(p2 * r, p2 * g, p2 * b));
      }
    }
  }

  void setBloch_deg_smooth(float theta, float phi, uint32_t c) {
    setBloch_deg(theta, phi, c, 2);
  }

  void animateTo(uint8_t gate, uint16_t animationLength = 2000)
  {
    if (frozen)
    {
      prevInterruptCount = interruptCount;
    }
    else if (gate == 0)
    {
      return;
    }
    if (gate == 9)
    {
      visualState.set(state.getCoordinates().v);
    }
    if (gate == 8)
    {
      state.collapse();
      visualState.set(state.getCoordinates().v);
    }
    float T_new = millis();
    float delta = T_new - T_freeze;
    if (delta > animationLength)
    {
      frozen = false;
      state.applyGateType(gate);
      Serial.println("Animation finished");
      return;
    }
    float d = delta * PI / float(animationLength);
    QuantumState from = state;
    from.applyGateType(gate, d);
    visualState.set(from.getCoordinates().v);
  }

  bool detectShaking()
  {
    float totalAcceleration = gravityVector.norm();
    if (shakingState)
    {
      float newTime = millis();
      float shakingCounter = newTime - T_shaking;
      if (shakingCounter < 300)
      {
        return false;
      }
      if (totalAcceleration > 11)
      {
        Serial.println("Randomizing");
        float randomTheta = (random(0, 1000)/1000.0f) * PI;
```

```
        float randomPhi = (random(0, 1000)/500.0f) * PI;
        state.setCoordinates(Coordinates(randomTheta, randomPhi));
        setLed(state.getCoordinates(), color(255, 0, 255));
        shakingState = false;
        return true;
      }
      if (shakingCounter > 800)
      {
        shakingState = false;
      }
      return false;
    }
    if (totalAcceleration > 11)
    {
      Serial.print("Detected shaking turning on shakingState, acc length: ");
      Serial.println(totalAcceleration);
      shakingState = true;
      T_shaking = millis();
    }
    return false;
  }

  int checkMotion()
  {
    if (frozen)
    {
      return 0;
    }
    frozen = true;
    T_freeze = micros();
    if (detectShaking())
    {
      return 9;
    }
    if (shakingState)
    {
      frozen = false;
      return 0;
    }
    // Handle tap interrupt
    if (interruptCount > prevInterruptCount)
    {
      uint8_t tapStatus = 0;
      myIMU.readRegister(&tapStatus, LSM6DS3_ACC_GYRO_TAP_SRC);
      prevInterruptCount = interruptCount;

      if (tapStatus & 0x01)
      {
        Serial.println("Collapsing");
        return 8;
      }
      else
      {
        Serial.println("Executing H gate");
        return 7;
      }
    }
    // Handle shaking
    for (int i = 0; i < 3; i++)
    {
      if (gyroVector[i] > GYRO_GATE_THRESHOLD)
      {
        return i + 1; // 1 = -x, 2 = -y, 3 = z
      }
    }
    for (int i = 0; i < 3; i++)
    {
      if (gyroVector[i] < - GYRO_GATE_THRESHOLD)
      {
        return i + 4; // 4 = x, 5 = y, 6 = -z
      }
```

```
  }
  frozen = false;
  return 0;
}

void writeToBLE(BLECharacteristic& destination, Vector3d vector) {
  float buffer[3] = {(float)vector(0), (float)vector(1), (float)vector(2)};
  destination.write(buffer, 3 * sizeof(float));
  for (uint16_t conn_hdl = 0; conn_hdl < QB_MAX_PRPH_CONNECTION; conn_hdl++)
  {
    if (Bluefruit.connected(conn_hdl) && destination.notifyEnabled(conn_hdl))
    {
      destination.notify(buffer, 3 * sizeof(float));
    }
  }
}

Vector3d getVectorFromBuffer(float *buffer) {
  // calibration of imu because imu is not aligned with bloch sphere
  float rx = (1 - 2 * QB_SX) * buffer[QB_IX];
  float ry = (1 - 2 * QB_SY) * buffer[QB_IY];
  float rz = (1 - 2 * QB_SZ) * buffer[QB_IZ];
  return Vector3d(rx, ry, rz);
}

void readIMU(bool print=true) {
  rbuffer[0] = imu.readFloatAccelX();
  rbuffer[1] = imu.readFloatAccelY();
  rbuffer[2] = imu.readFloatAccelZ();
  rgyrobuffer[0] = imu.readFloatGyroX();
  rgyrobuffer[1] = imu.readFloatGyroY();
  rgyrobuffer[2] = imu.readFloatGyroZ();

  float T_new = micros();
  float delta = T_new - T_imu;
  T_imu = T_new;

  Vector3d newGyro = getVectorFromBuffer(rgyrobuffer) * PI / 180;
  float d = min(delta / float(T_GYRO), 1.0f);
  gyroVector = d * newGyro + (1 - d) * gyroVector; // low pass filter

  Vector3d newGravity = getVectorFromBuffer(rbuffer);
  d = min(delta / float(T_ACC), 1.0f);
  gravityVector = d * newGravity + (1 - d) * gravityVector;

  yaw += gravityVector.dot(gyroVector);
  yaw = fmod(yaw, 2 * PI);

  if (print) {
    Serial.print(gravityVector(0));
    Serial.print("\t");
    Serial.print(gravityVector(1));
    Serial.print("\t");
    Serial.print(gravityVector(2));
    Serial.print("\t-1\t1\t");
    Serial.print(gyroVector(0));
    Serial.print("\t");
    Serial.print(gyroVector(1));
    Serial.print("\t");
    Serial.println(gyroVector(2));
  }

  writeToBLE(blecharacc, gravityVector);
  writeToBLE(blechargyr, gyroVector);
}

void startBLEadv(void)
{
  Serial.println("[INFO]{BLE} Start advertising...");
  // Advertising packet
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
```

```
    Bluefruit.Advertising.addTxPower();

    // Include HRM Service UUID
    Bluefruit.Advertising.addService(bleservice);

    // Secondary Scan Response packet (optional)
    // Since there is no room for 'Name' in Advertising packet
    Bluefruit.ScanResponse.addName();

    /* Start Advertising
     * - Enable auto advertising if disconnected
     * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
     * - Timeout for fast mode is 30 seconds
     * - Start(timeout) with timeout = 0 will advertise forever (until connected)
     *
     * For recommended advertising interval
     * https://developer.apple.com/library/content/qa/qa1931/_index.html
     */
    Bluefruit.Advertising.restartOnDisconnect(true);
    Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
    Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
    Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising after n
    seconds
  }

}; // end class

Qbead *Qbead::singletoninstance = nullptr;

} // end namespace

#endif // QBEAD_H
```

Listing B.1: Qbead.h

B.2. Source code XIAO Seeed ESP32S3

```
#ifndef QBEAD_H
#define QBEAD_H

#include <Arduino.h>
#include <Adafruit_NeoPixel.h>
#include <Wire.h>
#include <ICM_20948.h>
#include <ICM20948_WE.h>
#include <math.h>
#include <ArduinoEigen.h>
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLE2902.h>

using namespace Eigen;

// default configs
#define QB_LEDPIN 21
#define QB_PIXELCONFIG NEO_BRG + NEO_KHZ800
#define QB_IMU_ADDR 0x69
#define QB_IX 1
#define QB_IY 0
#define QB_IZ 2
#define QB_SX 0
#define QB_SY 0
#define QB_SZ 1
#define GYRO_GATE_THRESHOLD 12
#define QB_PIXEL_COUNT 62
#define QB_MAX_PRPH_CONNECTION 2
#define T_ACC 100000
#define T_GYRO 10000
#define TAP_THRESHOLD_TIME 400 // Threshold for tap detection in milliseconds
#define TAP_THRESHOLD 8 // Threshold for tap detection in g/s
```

```
#define DEBOUNCE_TIME 50 // Debounce time in milliseconds

const char QB_UUID_SERVICE[] = ”e5eaa0bd-babb-4e8c-a0f8-054ade68b043”;
// {0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6,0x1f,0x0c,0xe3};
const char QB_UUID_COL_CHAR[] = ”e5eaa0bd-babb-4e8c-a0f8-054ade68c043”;
// {0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6+1,0x1f,0x0c,0xe3};
const char QB_UUID_SPH_CHAR[] = ”e5eaa0bd-babb-4e8c-a0f8-054ade68d043”;
// {0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6+2,0x1f,0x0c,0xe3};
const char QB_UUID_ACC_CHAR[] = ”e5eaa0bd-babb-4e8c-a0f8-054ade68e043”;
// {0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6+3,0x1f,0x0c,0xe3};
const char QB_UUID_GYR_CHAR[] = ”e5eaa0bd-babb-4e8c-a0f8-054ade68f043”;
// {0x45,0x8d,0x08,0xaa,0xd6,0x63,0x44,0x25,0xbe,0x12,0x9c,0x35,0xc6+4,0x1f,0x0c,0xe3};

uint8_t zerobuffer20[] = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0
    x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
const std::complex<float>i(0, 1);
// We need both because ICM_20948_I2C does not support changing the gyro range
ICM20948_WE imuWE;
ICM_20948_I2C imuI2C;

// TODO manage namespaces better
// The setPixelColor switches blue and green
static uint32_t color(uint8_t r, uint8_t g, uint8_t b) {
  return ((uint32_t)r << 16) | ((uint16_t)b << 8) | g;
}

static uint8_t redch(uint32_t rgb) {
  return rgb >> 16;
}

static uint8_t greench(uint32_t rgb) {
  return 0x0000ff & rgb;
}

static uint8_t bluech(uint32_t rgb) {
  return (0x00ff00 & rgb) >> 8;
}

uint32_t colorWheel(uint8_t wheelPos) {
  wheelPos = 255 - wheelPos;
  if (wheelPos < 85) {
    return color(255 - wheelPos * 3, 0, wheelPos * 3);
  }
  if (wheelPos < 170) {
    wheelPos -= 85;
    return color(0, wheelPos * 3, 255 - wheelPos * 3);
  }
  wheelPos -= 170;
  return color(wheelPos * 3, 255 - wheelPos * 3, 0);
}

uint32_t colorWheel_deg(float wheelPos) {
  return colorWheel(wheelPos * 255 / 360);
}

float sign(float x) {
  if (x > 0) return +1;
  else return -1;
}

// z = cos(t)
// x = cos(p)sin(t)
// y = sin(p)sin(t)
// Return the angle in radians between the x-axis and the line to the point (x, y)
float phi(float x, float y) {
  return atan2(y, x);
}

float phi(float x, float y, float z) {
  return phi(x, y);
}
```

```
float theta(float x, float y, float z) {
  float ll = x * x + y * y + z * z;
  float l = sqrt(ll);
  float theta = acos(z / l);
  return theta;
}

bool checkThetaAndPhi(float theta, float phi) {
  return theta >= 0 && theta <= 180 && phi >= 0 && phi <= 360;
}

// In rads
void sphericalToCartesian(float theta, float phi, float& x, float& y, float& z)
{
  // Normalize yaw to be between 0 and 2*PI
  phi = fmod(phi, 2 * PI);
  if (phi < 0)
  {
    phi += 2 * PI;
  }
  if (!checkThetaAndPhi(theta * 180 / PI, phi * 180 / PI))
  {
    Serial.print("Theta or Phi out of range when creating coordinates class, initializing as
    1");
    Serial.print("Theta: ");
    Serial.print(theta);
    Serial.print("Phi: ");
    Serial.println(phi);
    x = 0;
    y = 0;
    z = 1;
    return;
  }

  x = sin(theta) * cos(phi);
  y = sin(theta) * sin(phi);
  z = cos(theta);
}

namespace Qbead {

class Coordinates
{
public:
  Vector3d v;

  Coordinates(float argx, float argy, float argz)
  {
    v = Vector3d(argx, argy, argz);
    v.normalize();
  }

  // In rads
  Coordinates(float theta, float phi)
  {
    float x, y, z = 0;
    sphericalToCartesian(theta, phi, x, y, z);
    v = Vector3d(x, y, z);
  }

  Coordinates(Vector3d vector)
  {
    v = vector;
    v.normalize();
  }

  // In rads
  float theta()
  {
    return acos(v(2));
```

```
  }

  // In rads
  float phi()
  {
    return atan2(v(1), v(0));
  }

  Vector2cf stateVector2D()
  {
    std::complex<float> alpha = cos(theta()/2);
    std::complex<float> beta = exp(i*phi()) * sin(theta()/2);
    return {alpha, beta};
  }

  float dist(Vector3d other) const
  {
    Vector3d diff = v - other;
    return diff.norm();
  }

  void set(float argx, float argy, float argz)
  {
    v = Vector3d(argx, argy, argz);
    v.normalize();
  }

  // in rads
  void set(float theta, float phi)
  {
    float x, y, z = 0;
    sphericalToCartesian(theta, phi, x, y, z);
    v = Vector3d(x, y, z);
  }

  void set(Vector3d vector) {
    v = vector;
    v.normalize();
  }

  // in rads
  void setTheta(float theta)
  {
    set(theta, phi());
  }

  // in rads
  void setPhi(float phi)
  {
    set(theta(), phi);
  }
};

class QuantumState
{
private:
  Coordinates stateCoordinates;

public:
  QuantumState(Coordinates argStateCoordinates) : stateCoordinates(argStateCoordinates) {}
  QuantumState() : stateCoordinates(0, 0, 1) {}

  void setCoordinates(Coordinates argStateCoordinates)
  {
    stateCoordinates.set(argStateCoordinates.v);
  }

  Coordinates getCoordinates()
  {
    return stateCoordinates;
  }
```

```

void collapse()
{
  const float a = (stateCoordinates.v(2) + 1) / 2; // probability of measuring |0>
  if (a < 0.0001) {
    stateCoordinates.set(0, 0, -1);
    return;
  } else if (a > 0.9999) {
    stateCoordinates.set(0, 0, 1);
    return;
  }
  const bool is1 = random(0, 100) <= a * a * 100;
  this->stateCoordinates.set(0, 0, is1 ? 1 : -1);
}

void applyGate(Matrix2cf gate)
{
  Vector2cf stateVector = stateCoordinates.stateVector2D();
  stateVector = gate * stateVector;
  stateVector.normalize();
  stateCoordinates.set(2*acos(abs(stateVector.x())), arg(stateVector.y()) - arg(stateVector
  .x()));
}

void applyGateType(uint16_t gateType, float rotationDegree = PI)
{
  switch (gateType)
  {
  case 1:
    gateX(-rotationDegree);
    break;
  case 2:
    gateY(-rotationDegree);
    break;
  case 3:
    gateZ(rotationDegree);
    break;
  case 4:
    gateX(rotationDegree);
    break;
  case 5:
    gateY(rotationDegree);
    break;
  case 6:
    gateZ(-rotationDegree);
    break;
  case 7:
    gateH(rotationDegree);
    break;
  default:
    break;
  }
}

// Rotate PI around the x axis
void gateX(float rotationDegree = PI)
{
  Matrix2cf gateMatrix;
  gateMatrix << cos(rotationDegree / 2.0f), -sin(rotationDegree / 2.0f) * i,
      -sin(rotationDegree / 2.0f) * i, cos(rotationDegree / 2.0f); // global phase differs
   from pauli gates but this doesn't matter for bloch sphere
  applyGate(gateMatrix);
}

// Rotate PI around the y axis
void gateZ(float rotationDegree = PI)
{
  Matrix2cf gateMatrix;
  gateMatrix << exp(-i * rotationDegree / 2.0f), 0,
      0, exp(i * rotationDegree / 2.0f);
  applyGate(gateMatrix);
```

```
  }

  // Rotate PI around the z axis
  void gateY(float rotationDegree = PI)
  {
    Matrix2cf gateMatrix;
    gateMatrix << cos(rotationDegree / 2.0f), -sin(rotationDegree / 2.0f),
        sin(rotationDegree / 2.0f), cos(rotationDegree / 2.0f);
    applyGate(gateMatrix);
  }

  // Rotate PI around the xz axis
  void gateH(float rotationDegree = PI)
  {
    Matrix2cf gateMatrix;
    gateMatrix << (cos(rotationDegree / 2.0f) - i * sin(rotationDegree / 2.0f) / sqrt(2.0f)), -i * sin(rotationDegree / 2.0f) / sqrt(2.0f),
        -i * sin(rotationDegree / 2.0f) / sqrt(2.0f), (cos(rotationDegree / 2.0f) + i * sin(rotationDegree / 2.0f) / sqrt(2.0f));
    applyGate(gateMatrix);
  }
};

class Qbead {
public:
  Qbead(const uint16_t pin00 = QB_LEDPIN,
        const uint16_t pixelconfig = QB_PIXELCONFIG,
        const uint8_t imu_addr = QB_IMU_ADDR)
      : pixels(Adafruit_NeoPixel(QB_PIXEL_COUNT, pin00, pixelconfig))
  {}

  static Qbead *singletoninstance; // we need a global singleton static instance because bluefruit callbacks do not support context variables -- thankfully this is fine because there is indeed only one Qbead in existence at any time

  Adafruit_NeoPixel pixels;

  BLEServer* bleserver;
  BLEService* bleservice;
  BLECharacteristic* blecharcol;
  BLECharacteristic* blecharsph;
  BLECharacteristic* blecharacc;
  BLECharacteristic* blechargyr;
  BLEAdvertising* bleadvertising;

  float rbuffer[3], rgyrobuffer[3];
  float T_imu;              // last update from the IMU
  float T_freeze = 0;
  float T_shaking = 0;
  float t_ble, p_ble; // theta and phi as sent over BLE connection
  uint32_t c_ble;
  bool frozen = false; // frozen means that there is an animation in progress
  bool shakingState = false; // if ShakingState is 1 detected shaking and if shaking keeps happening randomising state
  QuantumState state = QuantumState(Coordinates(-0.866, 0.25, -0.433));
  Coordinates visualState = Coordinates(-0.866, 0.25, -0.433);
  Vector3d gravityVector = Vector3d(0, 0, 1);
  Vector3d oldGravityVector = Vector3d(0, 0, 1);
  Vector3d gyroVector = Vector3d(0, 0, 1);
  float lastTapTime = 0;
  float lastDebounceTime = 0; // last time the tap was debounced
  bool waitingForSecondTap = false;
  float dt = 0; // time since the last IMU update

  // led map index to Coordinates
  // This map is for the first version of the flex-pcb
  Coordinates led_map_v1[62] = {
    Coordinates(-1, -0, -0),
    Coordinates(-0.866, 0, -0.5),
    Coordinates(-0.5, 0, -0.866),
    Coordinates(-0, 0, -1),
```

```
  Coordinates(0.5, 0, -0.866),
  Coordinates(0.866, 0, -0.5),
  Coordinates(1, 0, 0),
  Coordinates(-0.866, 0.25, -0.433),
  Coordinates(-0.5, 0.433, -0.75),
  Coordinates(-0, 0.5, -0.866),
  Coordinates(0.5, 0.433, -0.75),
  Coordinates(0.866, 0.25, -0.433),
  Coordinates(-0.866, 0.433, -0.25),
  Coordinates(-0.5, 0.75, -0.433),
  Coordinates(-0, 0.866, -0.5),
  Coordinates(0.5, 0.75, -0.433),
  Coordinates(0.866, 0.433, -0.25),
  Coordinates(-0.866, 0.5, 0),
  Coordinates(-0.5, 0.866, 0),
  Coordinates(-0, 1, 0),
  Coordinates(0.5, 0.866, 0),
  Coordinates(0.866, 0.5, 0),
  Coordinates(-0.866, 0.433, 0.25),
  Coordinates(-0.5, 0.75, 0.433),
  Coordinates(-0, 0.866, 0.5),
  Coordinates(0.5, 0.75, 0.433),
  Coordinates(0.866, 0.433, 0.25),
  Coordinates(-0.866, 0.25, 0.433),
  Coordinates(-0.5, 0.433, 0.75),
  Coordinates(-0, 0.5, 0.866),
  Coordinates(0.5, 0.433, 0.75),
  Coordinates(0.866, 0.25, 0.433),
  Coordinates(-0.866, -0, 0.5),
  Coordinates(-0.5, -0, 0.866),
  Coordinates(-0, -0, 1),
  Coordinates(0.5, -0, 0.866),
  Coordinates(0.866, -0, 0.5),
  Coordinates(-0.866, -0.25, 0.433),
  Coordinates(-0.5, -0.433, 0.75),
  Coordinates(-0, -0.5, 0.866),
  Coordinates(0.5, -0.433, 0.75),
  Coordinates(0.866, -0.25, 0.433),
  Coordinates(-0.866, -0.433, 0.25),
  Coordinates(-0.5, -0.75, 0.433),
  Coordinates(-0, -0.866, 0.5),
  Coordinates(0.5, -0.75, 0.433),
  Coordinates(0.866, -0.433, 0.25),
  Coordinates(-0.866, -0.5, -0),
  Coordinates(-0.5, -0.866, -0),
  Coordinates(-0, -1, -0),
  Coordinates(0.5, -0.866, -0),
  Coordinates(0.866, -0.5, -0),
  Coordinates(-0.866, -0.433, -0.25),
  Coordinates(-0.5, -0.75, -0.433),
  Coordinates(-0, -0.866, -0.5),
  Coordinates(0.5, -0.75, -0.433),
  Coordinates(0.866, -0.433, -0.25),
  Coordinates(-0.866, -0.25, -0.433),
  Coordinates(-0.5, -0.433, -0.75),
  Coordinates(-0, -0.5, -0.866),
  Coordinates(0.5, -0.433, -0.75),
  Coordinates(0.866, -0.25, -0.433),
};

void startAccelerometer()
{
  blecharacc = bleservice->createCharacteristic(QB_UUID_ACC_CHAR,
                 BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_NOTIFY);
  BLEDescriptor* pAccDesc = new BLEDescriptor("2901");
  pAccDesc->setValue("Accelerometer readout Characteristic");
  blecharacc->addDescriptor(pAccDesc);
  blecharacc->addDescriptor(new BLE2902());
  blecharacc->setValue(zerobuffer20, 3*sizeof(float));
}
```

```
// TODO: Check when the new flex-pcb has arrived
Coordinates led_map_v2[107] = {
  Coordinates(0.0, 0.0),
  Coordinates(0.39, 4.97),
  Coordinates(0.78, 4.97),
  Coordinates(1.18, 5.08),
  Coordinates(1.18, 4.87),
  Coordinates(1.57, 4.89),
  Coordinates(1.57, 5.06),
  Coordinates(1.95, 5.04),
  Coordinates(1.95, 4.91),
  Coordinates(2.34, 4.97),
  Coordinates(2.73, 4.97),
  Coordinates(0.78, 4.45),
  Coordinates(1.18, 4.55),
  Coordinates(1.18, 4.35),
  Coordinates(1.57, 4.37),
  Coordinates(1.57, 4.53),
  Coordinates(1.95, 4.51),
  Coordinates(1.95, 4.39),
  Coordinates(2.34, 4.45),
  Coordinates(0.39, 3.93),
  Coordinates(0.78, 3.93),
  Coordinates(1.18, 4.03),
  Coordinates(1.18, 3.82),
  Coordinates(1.57, 3.84),
  Coordinates(1.57, 4.01),
  Coordinates(1.95, 3.99),
  Coordinates(1.95, 3.87),
  Coordinates(2.34, 3.93),
  Coordinates(2.73, 3.93),
  Coordinates(0.78, 3.4),
  Coordinates(1.18, 3.3),
  Coordinates(1.57, 3.32),
  Coordinates(1.95, 3.34),
  Coordinates(2.34, 3.4),
  Coordinates(0.39, 2.88),
  Coordinates(0.78, 2.88),
  Coordinates(1.18, 2.98),
  Coordinates(1.18, 2.78),
  Coordinates(1.57, 2.8),
  Coordinates(1.57, 2.96),
  Coordinates(1.95, 2.94),
  Coordinates(1.95, 2.82),
  Coordinates(2.34, 2.88),
  Coordinates(2.73, 2.88),
  Coordinates(0.78, 2.36),
  Coordinates(1.18, 2.46),
  Coordinates(1.18, 2.25),
  Coordinates(1.57, 2.27),
  Coordinates(1.57, 2.44),
  Coordinates(1.95, 2.42),
  Coordinates(1.95, 2.29),
  Coordinates(2.34, 2.36),
  Coordinates(0.39, 1.83),
  Coordinates(0.78, 1.83),
  Coordinates(1.18, 1.93),
  Coordinates(1.18, 1.73),
  Coordinates(1.57, 1.75),
  Coordinates(1.57, 1.92),
  Coordinates(1.95, 1.89),
  Coordinates(1.95, 1.77),
  Coordinates(2.34, 1.83),
  Coordinates(2.73, 1.83),
  Coordinates(0.78, 1.31),
  Coordinates(1.18, 1.41),
  Coordinates(1.18, 1.21),
  Coordinates(1.57, 1.23),
  Coordinates(1.57, 1.39),
  Coordinates(1.95, 1.37),
  Coordinates(1.95, 1.25),
```

```
    Coordinates(2.34, 1.31),
    Coordinates(0.39, 0.79),
    Coordinates(0.78, 0.79),
    Coordinates(1.18, 0.89),
    Coordinates(1.18, 0.68),
    Coordinates(1.57, 0.7),
    Coordinates(1.57, 0.87),
    Coordinates(1.95, 0.85),
    Coordinates(1.95, 0.72),
    Coordinates(2.34, 0.79),
    Coordinates(2.73, 0.79),
    Coordinates(0.78, 0.26),
    Coordinates(1.18, 0.36),
    Coordinates(1.18, 0.16),
    Coordinates(1.57, 0.18),
    Coordinates(1.57, 0.35),
    Coordinates(1.95, 0.32),
    Coordinates(1.95, 0.2),
    Coordinates(2.34, 0.26),
    Coordinates(0.39, 6.02),
    Coordinates(0.78, 6.02),
    Coordinates(1.18, 6.12),
    Coordinates(1.18, 5.92),
    Coordinates(1.57, 5.94),
    Coordinates(1.57, 6.1),
    Coordinates(1.95, 6.08),
    Coordinates(1.95, 5.96),
    Coordinates(2.34, 6.02),
    Coordinates(2.73, 6.02),
    Coordinates(0.78, 5.5),
    Coordinates(1.18, 5.6),
    Coordinates(1.18, 5.4),
    Coordinates(1.57, 5.41),
    Coordinates(1.57, 5.58),
    Coordinates(1.95, 5.56),
    Coordinates(1.95, 5.44),
    Coordinates(2.34, 5.5),
    Coordinates(3.14, 4.97),
  };

  class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
      Serial.println("BLE: Device connected");
    }
    void onDisconnect(BLEServer* pServer) {
      Serial.println("BLE: Device disconnected");
    }
  };

  class ColorCharCallbacks: public BLECharacteristicCallbacks {
    void onWrite(BLECharacteristic *pCharacteristic) {
      Serial.println("[INFO]{BLE} Received a write on the color characteristic");
      uint8_t* pData = pCharacteristic->getData();
      singletoninstance->c_ble =  (pData[2] << 16) | (pData[1] << 8) | pData[0];
      Serial.print("[DEBUG]{BLE}Qbead received");
      Serial.println(singletoninstance->c_ble, HEX);
    }
  };

  class ThetaPhiCharCallbacks: public BLECharacteristicCallbacks {
    void onWrite(BLECharacteristic *pCharacteristic) {
      Serial.println("[INFO]{BLE} Received a write on the spherical coordinates
    characteristic");
      uint8_t* pData = pCharacteristic->getData();
      singletoninstance->t_ble = ((uint32_t)pData[0])*180/255;
      singletoninstance->p_ble = ((uint32_t)pData[1])*360/255;
      Serial.print("[DEBUG]{BLE} Received t=");
      Serial.print(singletoninstance->t_ble);
      Serial.print(" p=");
      Serial.println(singletoninstance->p_ble);
    }
```

```
};

void
begin()
{
  Wire.begin(40, 39);
  Wire.setClock(50000);  // drop to 50kHz
  singletoninstance = this;
  Serial.begin(9600);
  for (int waitCount = 0; waitCount < 50; waitCount++)
  {
    if (Serial) {break;}
    delay(100);
  }

  pixels.begin();
  clear();
  setBrightness(10);

  Serial.println("[INFO] Booting... Qbead on XIAO ESP32 compiled on " __DATE__ " at "
  __TIME__);

  BLEDevice::init("qbead | " __DATE__ " " __TIME__);
  // Bluefruit.begin(QB_MAX_PRPH_CONNECTION, 0);
  // Bluefruit.setName("qbead | " __DATE__ " " __TIME__);
  // Bluefruit.Periph.setConnectCallback(connect_callback);
  bleserver = BLEDevice::createServer();
  bleserver->setCallbacks(new MyServerCallbacks());
  bleservice = bleserver->createService(QB_UUID_SERVICE);
  // BLE Characteristic Bloch Sphere Visualizer color setup

  uint8_t zerobuffer2[] = {0 ,0};
  float zerobufferfloat[] = {0.0f, 0.0f, 0.0f};
  blecharcol = bleservice->createCharacteristic(QB_UUID_COL_CHAR,
                BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_WRITE);
  BLEDescriptor* pColDesc = new BLEDescriptor("2901");
  pColDesc->setValue("Color Characteristic");
  blecharcol->addDescriptor(pColDesc);
  blecharcol->setCallbacks(new ColorCharCallbacks());
  blecharcol->setValue(zerobuffer20, 3);

  blecharsph = bleservice->createCharacteristic(QB_UUID_SPH_CHAR,
                BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_WRITE);
  BLEDescriptor* pSphDesc = new BLEDescriptor("2901");
  pSphDesc->setValue("Theta and Phi Characteristic");
  blecharsph->addDescriptor(pSphDesc);
  blecharsph->setCallbacks(new ThetaPhiCharCallbacks());
  blecharsph->setValue(zerobuffer20, 2);

  blechargyr = bleservice->createCharacteristic(QB_UUID_GYR_CHAR,
                BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_NOTIFY);
  BLEDescriptor* pGyrDesc = new BLEDescriptor("2901");
  pGyrDesc->setValue("Gyroscope readout Characteristic");
  blechargyr->addDescriptor(pGyrDesc);
  blechargyr->addDescriptor(new BLE2902());
  blechargyr->setValue(zerobuffer20, 3*sizeof(float));

  startAccelerometer();

  if (bleservice) {
    Serial.println("starting service");
    bleservice->start();
  } else {
    Serial.println("Service is null!");
  }
  startBLEadv();

  imuI2C.begin(Wire, QB_IMU_ADDR);
  imuWE = ICM20948_WE(&Wire, QB_IMU_ADDR);
  imuWE.setGyrRange(ICM20948_GYRO_RANGE_2000);
  imuWE.setAccDLPF(ICM20948_DLPF_6);
```

```
    imuWE.setAccRange(ICM20948_ACC_RANGE_8G);
  }

  void startBLEadv(void)
  {
    bleadvertising = bleserver->getAdvertising();
    bleadvertising->addServiceUUID(QB_UUID_SERVICE);
    Serial.println("[INFO]{BLE} Start advertising...");
    // Advertising packet
    BLEAdvertisementData advertisementData;
    advertisementData.setName("qbead | " __DATE__ " " __TIME__);
    advertisementData.setFlags(6); // BLE_SIG_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE = 6

    /* Start Advertising
    * - Enable auto advertising if disconnected
    * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
    * - Timeout for fast mode is 30 seconds
    * - Start(timeout) with timeout = 0 will advertise forever (until connected)
    *
    * For recommended advertising interval
    * https://developer.apple.com/library/content/qa/qa1931/_index.html
    */
    bleadvertising->setAdvertisementData(advertisementData);
    bleadvertising->setMinInterval(32);
    bleadvertising->setMaxInterval(244);

    BLEDevice::startAdvertising();
  }

  void clear() {
    pixels.clear();
  }

  void show() {
    pixels.show();
  }

  void setBrightness(uint8_t b) {
    pixels.setBrightness(b);
  }

  void setLed(Coordinates coordinates, uint32_t color, int leds = 1) {
    float theta = coordinates.theta() * 180 / PI;
    float phi = coordinates.phi() * 180 / PI;
    if (phi < 0) {
      phi += 360;
    }
    setBloch_deg(theta, phi, color, leds);
  }

  void showAxis() {
    setLed(Coordinates(1, 0, 0), color(0, 0, 122));
    setLed(Coordinates(-1, 0, 0), color(0, 0, 122));
    setLed(Coordinates(0, 1, 0), color(0, 0, 122));
    setLed(Coordinates(0, -1, 0), color(0, 0, 122));
    setLed(Coordinates(0, 0, 1), color(0, 255, 0));
    setLed(Coordinates(0, 0, -1), color(255, 0, 0));
  }

  // in rads
  float getDistToLed(float theta, float phi, int index) {
    const Coordinates led = led_map_v1[index];
    const Coordinates reference(theta, phi);
    return led.dist(reference.v);
  }

  // Single bit is lit up on the Bloch sphere
  void setBloch_deg(float theta, float phi, uint32_t c, int leds = 1) {
    int index[leds];
    float dist[leds];
    for (int i = 0; i < leds; i++) {
```

```
      index[i] = -1;
      dist[i] = 1000;
    }
    for (int i = 0; i < QB_PIXEL_COUNT; i++) {
      float d = getDistToLed(theta * PI / 180, phi * PI / 180, i);
      for (int j = 0; j < leds; j++) {
        if (d < dist[j]) {
          for (int k = leds - 1; k > j; k--) {
            index[k] = index[k - 1];
            dist[k] = dist[k - 1];
          }
          index[j] = i;
          dist[j] = d;
          break;
        }
      }
    }
    for (int i = 0; i < leds; i++) {
      if (index[i] != -1) {
        uint8_t r = redch(c);
        uint8_t g = greench(c);
        uint8_t b = bluech(c);
        float p2 =  pow(200, -dist[i]);
        pixels.setPixelColor(index[i], color(p2 * r, p2 * g, p2 * b));
      }
    }
  }

  void setBloch_deg_smooth(float theta, float phi, uint32_t c) {
    setBloch_deg(theta, phi, c, 2);
  }

  void animateTo(uint8_t gate, uint16_t animationLength = 2000)
  {
    if (gate == 0)
    {
      return;
    }
    if (gate == 9)
    {
      visualState.set(state.getCoordinates().v);
    }
    if (gate == 8)
    {
      state.collapse();
      visualState.set(state.getCoordinates().v);
    }
    float T_new = millis();
    float delta = T_new - T_freeze;
    if (delta > animationLength)
    {
      frozen = false;
      state.applyGateType(gate);
      Serial.println("Animation finished");
      return;
    }
    float d = delta * PI / float(animationLength);
    QuantumState from = state;
    from.applyGateType(gate, d);
    visualState.set(from.getCoordinates().v);
  }

  bool detectShaking()
  {
    float totalAcceleration = gravityVector.norm();
    if (shakingState)
    {
      float newTime = millis();
      float shakingCounter = newTime - T_shaking;
      if (shakingCounter < 300)
      {
```

```
        return false;
      }
      if (totalAcceleration > 3)
      {
        Serial.println("Randomizing");
        float randomTheta = (random(0, 1000)/1000.0f) * PI;
        float randomPhi = (random(0, 1000)/500.0f) * PI;
        state.setCoordinates(Coordinates(randomTheta, randomPhi));
        setLed(state.getCoordinates(), color(255, 0, 255));
        shakingState = false;
        return true;
      }
      if (shakingCounter > 800)
      {
        shakingState = false;
      }
      return false;
    }
    if (totalAcceleration > 3)
    {
      Serial.print("Detected shaking turning on shakingState, acc length: ");
      Serial.println(totalAcceleration);
      shakingState = true;
      T_shaking = millis();
    }
    return false;
  }

  bool detectDoubleTap(float acc)
  {
    float currentTime = millis();

    // If waiting too long for second tap, reset state
    if (waitingForSecondTap && (currentTime - lastTapTime > TAP_THRESHOLD_TIME))
    {
      waitingForSecondTap = false;
    }

    // Check for tap condition
    if (abs(acc) > TAP_THRESHOLD)
    {
      // Debounce: ensure enough time since last detected tap
      if (currentTime - lastDebounceTime > DEBOUNCE_TIME)
      {
        lastDebounceTime = currentTime;

        if (waitingForSecondTap)
        {
          waitingForSecondTap = false;
          return true; // Second tap detected within threshold time
        }
        else
        {
          // First tap detected
          lastTapTime = currentTime;
          waitingForSecondTap = true;
        }
      }
    }
    return false;
  }

  int checkMotion()
  {
    if (frozen)
    {
      return 0;
    }
    frozen = true;
    T_freeze = micros();
    if (detectShaking())
```

```
    {
      return 9;
    }
    if (shakingState)
    {
      frozen = false;
      return 0;
    }
    // Handle double tap
    float acc = (gravityVector(2) - oldGravityVector(2)) * 1000000 / dt;
    Serial.print("acc: ");
    Serial.println(acc);
    if (detectDoubleTap(acc))
    {
      Serial.println("Collapse detected");
      return 8; // collapse
    }
    float accX = (gravityVector(0) - oldGravityVector(0)) * 1000000 / dt;
    float accY = (gravityVector(1) - oldGravityVector(1)) * 1000000 / dt;
    if (detectDoubleTap(accX) || detectDoubleTap(accY))
    {
      Serial.println("Hadamard detected");
      return 7; // Hadamard
    }
    // Handle rotating
    for (int i = 0; i < 3; i++)
    {
      if (gyroVector[i] > GYRO_GATE_THRESHOLD)
      {
        return i + 1; // 1 = -x, 2 = -y, 3 = z
      }
    }
    for (int i = 0; i < 3; i++)
    {
      if (gyroVector[i] < - GYRO_GATE_THRESHOLD)
      {
        return i + 4; // 4 = x, 5 = y, 6 = -z
      }
    }
    frozen = false;
    return 0;
  }

  void writeToBLE(BLECharacteristic* destination, Vector3d vector) {
    float buffer[] = {(float)vector(0), (float)vector(1), (float)vector(2)};
    if (destination)
    {
      destination->setValue((uint8_t*)buffer, sizeof(buffer));
      destination->notify();
    } else
    {
      Serial.println("destination is null");
    }
  }

  Vector3d getVectorFromBuffer(float *buffer) {
    // calibration of imu because imu is not aligned with bloch sphere
    float rx = (1 - 2 * QB_SX) * buffer[QB_IX];
    float ry = (1 - 2 * QB_SY) * buffer[QB_IY];
    float rz = (1 - 2 * QB_SZ) * buffer[QB_IZ];
    return Vector3d(rx, ry, rz);
  }

  void readIMU(bool print=true) {
    while (!imuI2C.dataReady()) {
      delay(20);  // 1-2 ms delay is fine
    }

    imuI2C.getAGMT();
    rbuffer[0] = imuI2C.accX() / 1000.0f; // convert to g
    rbuffer[1] = imuI2C.accY() / 1000.0f;
```

```
    rbuffer[2] = imuI2C.accZ() / 1000.0f;
    rgyrobuffer[0] = imuI2C.gyrX();
    rgyrobuffer[1] = imuI2C.gyrY();
    rgyrobuffer[2] = imuI2C.gyrZ();

    float T_new = micros();
    dt = T_new - T_imu;
    T_imu = T_new;

    Vector3d newGyro = getVectorFromBuffer(rgyrobuffer) * PI / 180;
    float d = min(dt / float(T_GYRO), 1.0f);
    gyroVector = d * newGyro + (1 - d) * gyroVector; // low pass filter

    Vector3d newGravity = getVectorFromBuffer(rbuffer);
    d = min(dt / float(T_ACC), 1.0f);
    oldGravityVector = gravityVector;
    gravityVector = d * newGravity + (1 - d) * gravityVector;

    if (print) {
      Serial.print(gravityVector(0));
      Serial.print("\t");
      Serial.print(gravityVector(1));
      Serial.print("\t");
      Serial.print(gravityVector(2));
      Serial.print("\t-1\t1\t");
      Serial.print(gyroVector(0));
      Serial.print("\t");
      Serial.print(gyroVector(1));
      Serial.print("\t");
      Serial.println(gyroVector(2));
    }

    if (blecharacc) {
      writeToBLE(blecharacc, gravityVector);
    }
    if (blechargyr) {
      writeToBLE(blechargyr, gyroVector);
    }
  }
}; // end class

Qbead *Qbead::singletoninstance = nullptr;

} // end namespace

#endif // QBEAD_H
```

Listing B.2: QbeadESP32

B.3. Source code Pauli gate example

```
#include <Qbead.h>

Qbead::Qbead bead;
int rotationState = 0;
uint32_t stateColor = color(255, 255, 255);
const bool toggleAnimationOn = 1;

void setup() {
  bead.begin();
  bead.setBrightness(25); // way too bright
  Serial.println("testing all pixels discretely");
  for (int i = 0; i < bead.pixels.numPixels(); i++) {
    bead.pixels.setPixelColor(i, color(255, 255, 255));
    bead.pixels.show();
    delay(5);
  }
  Serial.println("testing smooth transition between pixels");
  for (int phi = 0; phi < 360; phi += 30) {
    for (int theta = 0; theta < 180; theta += 3) {
      bead.clear();
```

```
      bead.setBloch_deg(theta, phi, colorWheel_deg(phi));
      bead.show();
    }
  }
  Serial.println("starting inertial tracking");
}

void loop() {
  bead.readIMU(false);
  bead.clear();
  bead.showAxis();
  stateColor = color(255, 255, 255);
  Serial.print("rotationState: ");
  Serial.println(rotationState);
  if (bead.frozen)
  {
    stateColor = color(122, 122, 0);
  }
  else
  {
    rotationState = bead.checkMotion();
    if (rotationState != 0)
    {
      bead.frozen = true;
      bead.T_freeze = millis();
    }
  }
  bead.animateTo(rotationState, 2000);
  bead.setLed(bead.visualState, stateColor);
  bead.show();
}
```

Listing B.3: Pauli gate detection

B.4. Source code Decoherence

```
#include <Qbead.h>

Qbead::Qbead bead;
int rotationState = 0;
uint32_t stateColor = color(255, 255, 255);
uint32_t decoherenceColor = color(122, 0, 122);
const bool toggleAnimationOn = 1;
Qbead::Coordinates oldCoordinates(0, 0, 1);
int t = 0;
bool wasFrozen = false;

void setup()
{
    bead.begin();
    bead.setBrightness(25);
    Serial.println("testing all pixels discretely");
    for (int i = 0; i < bead.pixels.numPixels(); i++)
    {
        bead.pixels.setPixelColor(i, color(255, 255, 255));
        bead.pixels.show();
        delay(5);
    }
    Serial.println("testing smooth transition between pixels");
    for (int phi = 0; phi < 360; phi += 30)
    {
        for (int theta = 0; theta < 180; theta += 3)
        {
            bead.clear();
            bead.setBloch_deg(theta, phi, colorWheel_deg(phi));
            bead.show();
        }
    }
    Serial.println("starting inertial tracking");
    oldCoordinates = bead.state.getCoordinates();
    t = millis();
```

```
}

void loop()
{
    bead.readIMU(true);
    bead.clear();
    bead.showAxis();
    stateColor = color(255, 255, 255);
    Serial.print("rotationState: ");
    Serial.println(rotationState);
    if (bead.frozen)
    {
        stateColor = color(122, 122, 0);
        wasFrozen = true;
    }
    else
    {
        if (wasFrozen)
        {
            wasFrozen = false;
            oldCoordinates = bead.state.getCoordinates();
        }
        rotationState = bead.checkMotion();
        if (rotationState != 0)
        {
            bead.frozen = true;
            bead.T_freeze = millis();
        }
        float phi = bead.state.getCoordinates().phi();
        int dt = millis() - t;
        float randInt = random(200, 10000);
        phi += dt / randInt;
        if (phi > 2 * PI)
        {
            phi -= 2 * PI;
        }
        Qbead::Coordinates newCoordinates(bead.state.getCoordinates().theta(), phi);
        bead.state.setCoordinates(newCoordinates);
        bead.visualState = bead.state.getCoordinates();
        bead.setLed(oldCoordinates, decoherenceColor);
    }
    t = millis();
    bead.animateTo(rotationState, 2000);
    bead.setLed(bead.visualState, stateColor, 1);
    bead.show();
}
```

Listing B.4: Decoherence

Bibliography

[1] Quantum Technology and Application Consortium – QUTAC, Bayerstadler, Andreas, Becquin, Guillaume, *et al.*, "Industry quantum computing applications," *EPJ Quantum Technol.*, vol. 8, no. 1, p. 25, 2021. DOI: 10.1140/epjqt/s40507-021-00114-x. [Online]. Available: https://doi.org/10.1140/epjqt/s40507-021-00114-x.

[2] S. Lloyd, "Ultimate physical limits to computation," *Nature*, vol. 406, pp. 1047–1054, 2000, Quantum decoherence, ISSN: 1476-4687. DOI: https://doi.org/10.1038/35023282. [Online]. Available: https://www.nature.com/articles/35023282.

[3] T. Wong, *Introduction to Classical and Quantum Computing*. Rooted Grove, 2022.

[4] C. P. Williams, *Explorations in Quantum Computing*. London: Springer London, Jan. 2011. DOI: 10.1007/978-1-84628-887-6.

[5] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010, pp. 17–28, 174.

[6] *Bloch sphere - Wikipedia — en.wikipedia.org*, https://en.wikipedia.org/wiki/Bloch_sphere, [Accessed 30-04-2025].

[7] Y.-P. Liao, Y.-L. Cheng, Y.-T. Zhang, H.-X. Wu, and R.-C. Lu, "The interactive system of bloch sphere for quantum computing education," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022, pp. 718–723. DOI: 10.1109/QCE53715.2022.00097.

[8] P. Hu, Y. Li, R. S. K. Mong, and C. Singh, "Student understanding of the bloch sphere," *European Journal of Physics*, vol. 45, no. 2, p. 025 705, Feb. 2024. DOI: 10.1088/1361-6404/ad2393. [Online]. Available: https://dx.doi.org/10.1088/1361-6404/ad2393.

[9] F. van der Wal, H. Bakker, and R. Gosselink, "The qbead," Unpublished, will be published at the same time as this paper.

[10] G. Lindblad, "A general no-cloning theorem," *Letters in Mathematical Physics*, vol. 47, pp. 189–196, 1999, ISSN: 1573-0530. DOI: https://doi.org/10.1023/A:1007581027660. [Online]. Available: https://link.springer.com/article/10.1023/A:1007581027660.

[11] P. A. M. Dirac, "A new notation for quantum mechanics," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, no. 3, pp. 416–418, 1939. DOI: 10.1017/S0305004100021162.

[12] M. Newman. "Bloch sphere." (), [Online]. Available: https://prefetch.eu/know/concept/bloch-sphere/.

[13] P. Kaye, R. Laflamme, and M. Mosca, *An Introduction to Quantum Computing*. USA: Oxford University Press, Inc., 2007, ISBN: 0198570007.

[14] J. Preskill, *Quantum computing 40 years later*, 2023. arXiv: 2106.10522 [quant-ph]. [Online]. Available: https://arxiv.org/abs/2106.10522.

[15] A. Ekert, T. Hosgood, A. Kay, and C. Macchiavello. "Introduction to Quantum Information Science." (Dec. 8, 2024), [Online]. Available: https://qubit.guide.

[16] A. Ekert, P. M. Hayden, and H. Inamori, "Basic concepts in quantum computation," in *Coherent atomic matter waves*. Springer Berlin Heidelberg, 2024, pp. 661–701, ISBN: 9783540410478. DOI: 10.1007/3-540-45338-5_10. [Online]. Available: http://dx.doi.org/10.1007/3-540-45338-5_10.

[17] K. Herb. "Bloch sphere simulator." (), [Online]. Available: https://bloch.kherb.io/.

[18] H. E. Brandt, "Qubit devices and the issue of quantum decoherence," *Progress in Quantum Electronics*, vol. 22, no. 5, p. 260, 1999, ISSN: 0079-6727. DOI: https://doi.org/10.1016/S0079-6727(99)00003-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0079672799000038.

[19] V. Vedral and M. B. Plenio, "Basics of quantum computation," *Progress in Quantum Electronics*, vol. 22, no. 1, p. 28, 1998, ISSN: 0079-6727. DOI: https://doi.org/10.1016/S0079-6727(98)00004-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0079672798000044.

[20] M. Schlosshauer, "Quantum decoherence," *Physics Reports*, vol. 831, pp. 1–57, 2019, Quantum decoherence, ISSN: 0370-1573. DOI: https://doi.org/10.1016/j.physrep.2019.10.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0370157319303084.

[21] R. Sutor, *Dancing with Qubits, Second Edition*. Packt Publishing, 2019, pp. 454–476, ISBN: 9781837636754.

[22] P. Goiporia. "Suppressing errors with dynamical decoupling using pulse control on amazon braket." (2022), [Online]. Available: https://aws.amazon.com/blogs/quantum-computing/suppressing-errors-with-dynamical-decoupling-using-pulse-control-on-amazon-braket/.

[23] W. Yang, Z.-Y. Wang, and R.-B. Liu, "Preserving qubit coherence by dynamical decoupling," *Frontiers of Physics in China*, vol. 6, no. 1, pp. 2–14, Mar. 2011, ISSN: 2095-0470. DOI: 10.1007/s11467-010-0113-8. [Online]. Available: https://doi.org/10.1007/s11467-010-0113-8.

[24] X. Peng, D. Suter, and D. A. Lidar, "High fidelity quantum memory via dynamical decoupling: Theory and experiment," *Journal of Physics B: Atomic, Molecular and Optical Physics*, vol. 44, no. 15, p. 154 003, Jul. 2011. DOI: 10.1088/0953-4075/44/15/154003. [Online]. Available: https://dx.doi.org/10.1088/0953-4075/44/15/154003.

[25] N. Ezzell, B. Pokharel, L. Tewala, G. Quiroz, and D. A. Lidar, "Dynamical decoupling for superconducting qubits: A performance survey," *Physical Review Applied*, vol. 20, no. 6, Dec. 2023, ISSN: 2331-7019. DOI: 10.1103/physrevapplied.20.064027. [Online]. Available: http://dx.doi.org/10.1103/PhysRevApplied.20.064027.