



CG4002 Computer Engineering Capstone Project
2022/2023 Semester 1

**“Laser Tag++”
Design Report**

Group B16	Name	Student #	Specific Contribution
Member #1	Alexander Tan Jun An	A0199267J	1, 2, Hw Sensors
Member #2	Pranav Venkatram	A0220015E	1, 2.2, 2.3, Hw AI
Member #3	Braden Teo Wei Ren	A0218202W	1, 2.1, Comms Internal
Member #4	Danzel Ong Jing Hern	A0199331Y	1, 2, Comms External
Member #5	Hu Xuefei	A0220693H	1, 2.3, Sw Visualizer

Section 1 System Functionalities

This section will list the system functionalities for the laser tag game, through user stories.

As a...	I want...	So that...
FPS gamer	To be able to fire shots at another player	I will be able to do damage to the opponent
AR game enthusiast	To be able to use motion controls to execute actions such as reloading, throwing a grenade, activating a shield and logging out of the system	I can maximise immersion, by not requiring the need to press buttons
AR game enthusiast	To be able to get visual feedback of my actions on the visualiser, such as when firing my gun or activating a motion control such as reload, grenade or shield	I will be able to understand what is happening in the AR game
First time AR game player	The hardware to be easy and comfortable to use	It will not be too overwhelming for a first timer
FPS Gamer	To see my HP at all times	I know how well I am doing
FPS Gamer	To see the HP of my current shield, and when it will be available for activation again	I can plan strategies ahead
FPS Gamer	To see how many bullets are left in the magazine	I know when to reload
FPS Gamer	To see how many grenades and shields I have left	I can plan strategies
Avid gamer	Every shot I made to be successfully registered, where shots do not “disappear” after firing	I can have an enjoyable and fair experience
First time AR game player	Every action I make to be registered by both me and my opponent’s visualiser with low latency	I can be fully immersed in a seamless AR game
AR game enthusiast	To not be randomly disconnected from the game, or be able to easily reconnect and continue playing	The gameplay experience is not compromised

Section 2 Overall System Architecture

This section of the report will provide details on the planned system architecture of the project, the system's intended final form, as well as the main algorithm for the game.

2.1 Overview of System Architecture

The following diagram illustrates the high-level system architecture of the system.

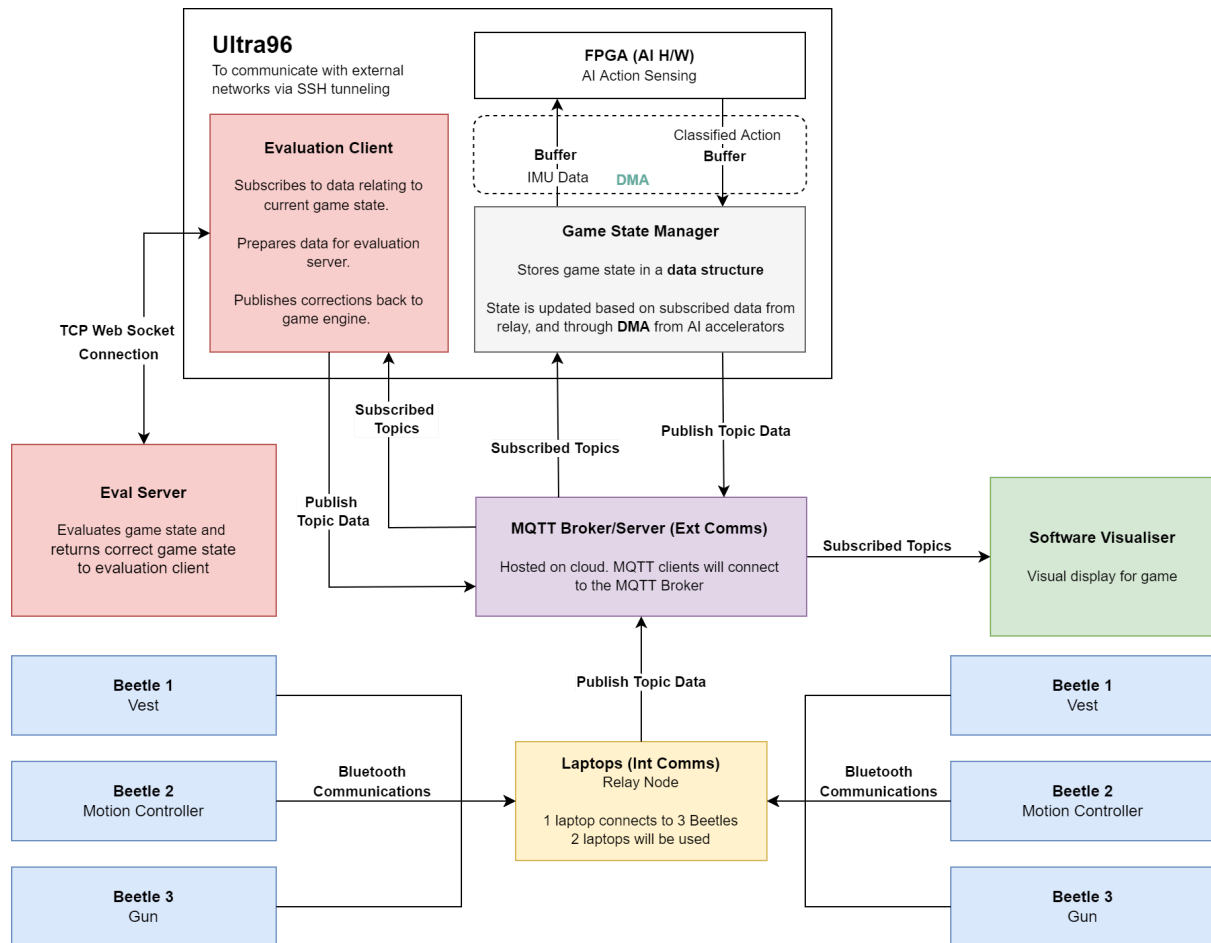


Figure 2.1: Laser Tag Game System Architecture

2.1.1 Beetle and Ultra96 Implementations

For the implementation, each beetle will serve a different role in the game, receiving raw data from the different sensors such as our IR sensors, triggers, and IMUs. As bluetooth connections are low bandwidth and unstable, each beetle will also carry out simple feature extraction functions, to reduce the packet size, and hence data, needed to be transferred over bluetooth connection to the relay node laptop, improving reliability and latency.

The Ultra96 board will also run both the game state manager, as well as our evaluation client, on top of AI processing. The game state management system implemented will be a simple program that manages the current game state, stored in a local data structure. The evaluation client, on the other hand, prepares evaluation packets for the evaluation server, and will communicate directly with the evaluation server, and game state manager.

2.1.2 Communications and Software Interfaces

In general, most of the communications between systems will be conducted through MQTT protocol, where each component will subscribe or publish to the relevant topics on the MQTT broker, hosted on the cloud, allowing different components on different networks to communicate with each other reliably, as shown in Figure 2.1 above. Each MQTT client e.g

the internal comms relay node and software visualiser, will run the MQTT client program to interface with the MQTT broker, and receive their required information.

The only places where MQTT communications will not be used, are between the beetles and the relay node, which uses short range, local bluetooth communications, the game state manager and AI accelerators/classifier which relies on DMA (Direct Memory Access) on the Ultra96, and between the evaluation client and server, which will communicate through TCP Web Socket protocols, where the eval client web socket script will be run on the Ultra96.

More detailed information on communications, protocols used, and software interfaces for the system can be found in *Section 6: External Communications* of this report.

2.2 System Final Form

The following figure shows the system's intended final form. It will be composed of a glove, vest, and a shooting apparatus.

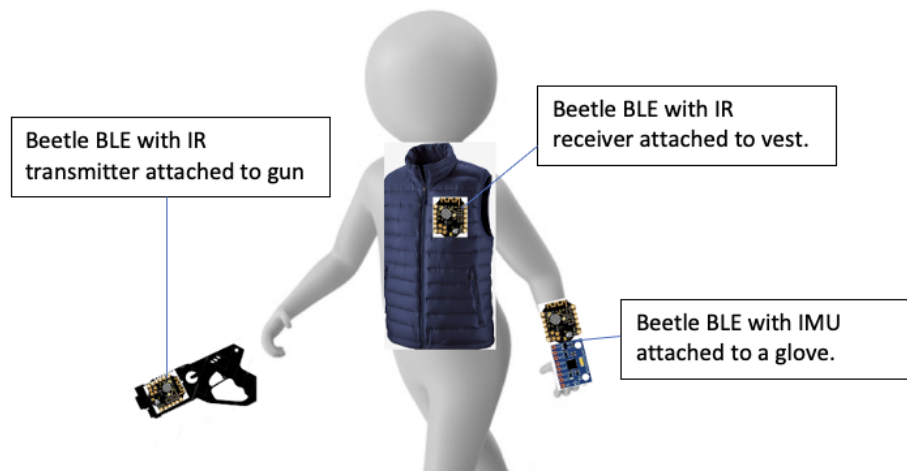


Figure 2.2: Diagram for system final form

2.3 System Main Algorithm

The following figure illustrates the high level overview of the main 6-step algorithm for the laser tag game.

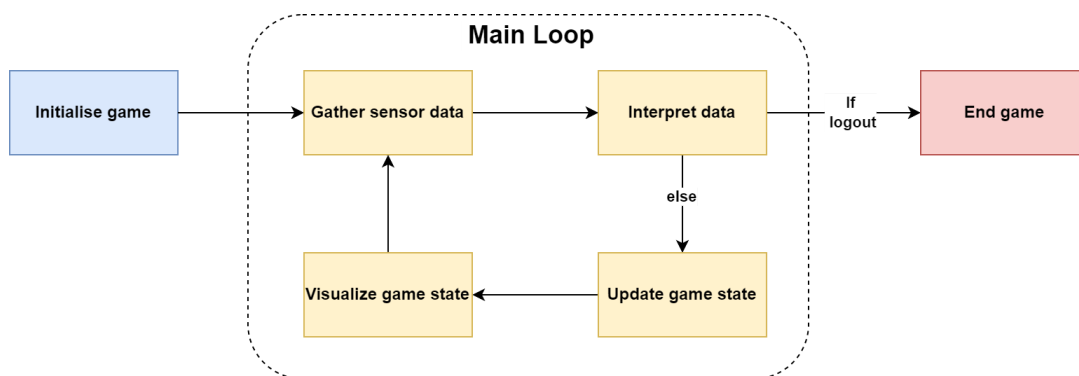


Figure 2.3: Main Game Algorithm

2.3.1 Initialise Game

In this first step, the game state manager is initialised, where the game state for all players are set to their default, specified values. Communications between systems, such as bluetooth between each beetle and the relay node, and between the MQTT broker and clients are established.

2.3.2 Main Algorithm Loop

After initialisation, the system transits into the main loop, where gameplay occurs. This main loop consists of 4 separate steps.

The first step, “Gather Sensor Data” happens when the IMU and IR sensor data is gathered on the beetles. The raw data from the sensors are cleaned, before being sent over bluetooth to the relay node, for the next step, “Interpret Data”.

In this step, data from the relay node is transmitted to the Ultra96 which hosts our game state manager and AI classifier, via MQTT protocol. IMU data is fed to the AI model to classify the action, while other data collected from the beetles to log shots being fired or getting shot, are decrypted and read by the game state manager.

The AI model will have been trained on IMU sensor data such as acceleration and angular velocity along the x, y and z axes. Further, features will be extracted before passing data to the model using Spectral Analysis. This will reduce model complexity and enable better accuracy. While all subsystems are being developed, we will use sample IMU data obtained from Kaggle [12] for training, validation and testing. Once the hardware sensor subsystem is complete, we will re-train, re-validate and re-test the model using our own IMU sensor data.

The third step in the loop is to “Update Game State”. After interpreting the players actions, the game state which is stored as a data structure, will be updated with the received data.

Finally, in the “Visualise Game State” step, the software visualiser receives the updated game state for all players to be displayed from the game state manager, by subscribing to the MQTT topics published by the game state manager, before looping back to the first step of the loop again. This 4-step loop continues infinitely, as long as gameplay continues.

2.3.3 End Game

When the “logout” action is interpreted for all players, the system enters the “End Game” part of the main algorithm. At this stage, all gameplay is stopped, and the game state is frozen by the game state manager. Final scores will be displayed on the software visualiser. A new game can be started by re-initialising the system.

Section 3 Hardware Sensors [Alexander Tan Jun An]

This section will provide details on the various devices/components that will be used in the overall system.

Section 3.1 Hardware Devices

In this section, the devices required for our wearable vest and gun will be listed. For each device, the relevant supporting components will be described, as well as the datasheet of the device itself.

1. Beetle BLE:

The Beetle BLE (formerly known as Bluno Beetle) is a board based on Arduino Uno with Bluetooth 4.0 (BLE). The main role of Beetle BLE will be to retrieve the data generated by the connected sensors, do some pre-processing, and then feed the cleaned data to the machine learning models running on the Ultra96. Bluetooth will be used for communication between the Beetle BLE and Ultra96.

Datasheet:

- https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339
- <https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf>

2. MPU-6050 IMU:

The MPU-6050 IMU will be worn on the player's left hand. Being able to measure velocity, orientation, acceleration, displacement, and other motion like features will allow us to identify the actions performed by the player.

Datasheet:

- <https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf>
- <https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf>

3. IR Transmitter TSAL6200:

The IR transmitter will be present on the guns of both players. Upon triggering, a pulse will be fired from the transmitter. The pulse released will be between 940-950nm. A resistor will be connected in series with the transmitter to limit the current going through it.

To improve the range, we can either add a lens in front of the transmitter to focus the beam of IR light, or to increase the current going through the transmitter.

Datasheet:

- <https://www.farnell.com/datasheets/1864140.pdf>

4. IR Receiver Diode TSOP38238:

The IR receiver diode will be present on the vest of both players. It will be able to detect IR pulses from the IR transmitter of the other player. Upon receiving

Datasheet:

- <https://www.farnell.com/datasheets/2049183.pdf>

5. Red LED HUR334d:

This red LED will be the feedback unit found on both the vest and the shooting apparatus.

Datasheet:

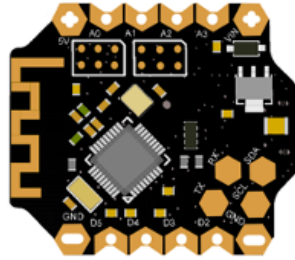
- http://www.sgbotic.com/products/datasheets/display/5MM_LED_colors.pdf

Section 3.2 Pin Table

In this section, the pinout diagram of each individual component will be provided. The individual connections between components will also be provided.

1. Player's glove/Beetle BLE to MPU-6050 IMU:

The following figures show the pinout diagrams for both the Beetle BLE and MPU-6050 IMU respectively.



2. Player's gun/Beetle BLE to IR Transmitter

The following is a pinout diagram of the IR transmitter.



Figure 3.3: Pinout diagram of IR Transmitter

For clarity, the following pin table shows the connections across Beetle BLE and the IR Transmitter.

Beetle BLE	IR Transmitter TSAL6200
D3 (PWM Channel 3)	+
GND	-

3. Player's vest/Beetle BLE to IR Receiver

The following is a pinout diagram of the IR receiver.

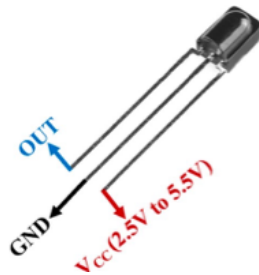


Figure 3.4: Pinout diagram of IR Receiver

For clarity, the following pin table shows the connections across Beetle BLE and the IR Receiver Diode.

Beetle BLE	IR Receiver Diode TSOP38238
5V	VCC
GND	GND
D2	OUT

Section 3.3 Schematics

This section briefly describes how the Beetle BLE and various other components should be connected. They are separated into the player's gun, vest, and glove.

1. Player's glove/Beetle BLE to MPU-6050 IMU:

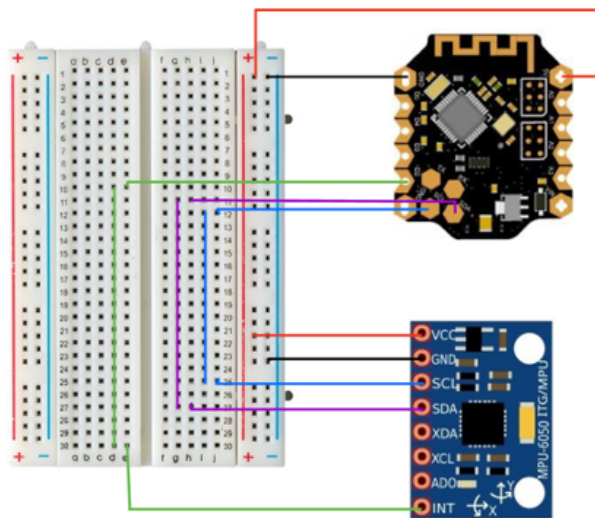


Figure 3.5: Proposed schematic of Beetle BLE and MPU-6050 IMU

2. Player's gun/Beetle BLE to IR Transmitter:

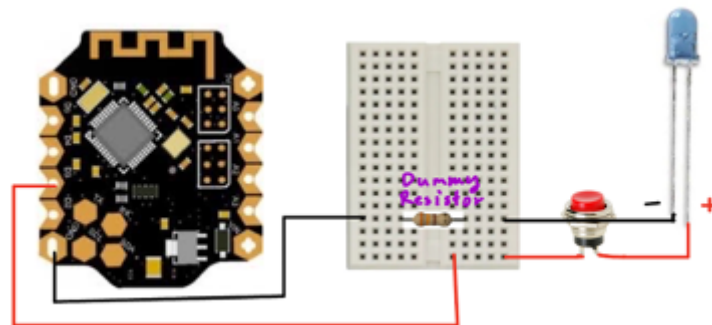


Figure 3.6: Proposed schematic of Beetle BLE and IR Transmitter

3. Player's vest/Beetle BLE to IR Receiver:

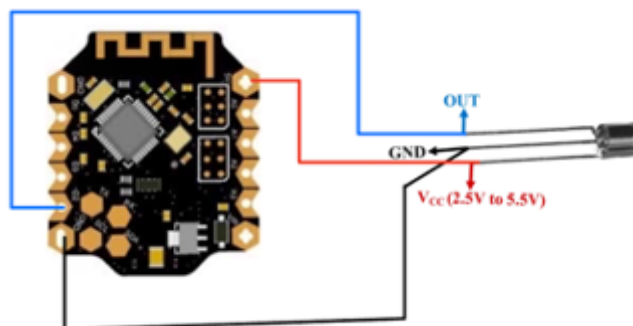


Figure 3.7: Proposed schematic of Beetle BLE and IR Receiver Diode

Section 3.4 Operating Voltage and Current

This section describes the operational voltage level and current drawn by each component/device.

1. Beetle BLE

- Operational Voltage: 5V DC
- Operational Current: 10 mA
- Power draw = $5V \times 10mA = 50mW$

2. MPU-6050 IMU

- Operational Voltage: 2.375V - 3.46V
- Operational Current: 3.6mA - 3.9mA
- Power draw = $3V \times 3.8mA = 11.4mW$

3. IR Transmitter TSAL6200

- Operational Voltage: 1.35V
- Operational Current: 100mA
- Power draw = $1.35V \times 100mA = 125mW$

Powering off a 5V pin, $R = \frac{V}{I} = \frac{5V - 1.35V}{0.1} = 36.5\Omega$. Therefore, to prevent high current from damaging the transmitter, resistance of more than 36.5Ω should be used.

4. IR Receiver Diode TSOP38238

- Operational Voltage: 2.5V - 5V
- Operational Current: 0.27mA - 0.45mA
- Power draw = $5V \times 0.35mA = 1.75mW$

5. Red LED HUR334d

- Operational Voltage: 2V
- Operational Current: 20mA
- Power draw = $2V \times 20mA = 40mW$

The estimated power for each major component of the system is in the table below.

Component/Device	Estimated Power Draw (mW)
Glove (Beetle BLE + IMU)	61.4mW
Gun (Beetle BLE + IR Transmitter + Red LED)	215mW
Vest (Beetle BLE + IR Receiver + Red LED)	91.75mW

As each Beetle oversees one component, the battery capacity required will not be extremely high. As the devices will be donned by the player, weight is a major concern. Batteries that are light in weight will be ideal for this use case.

Furthermore, since the project spans over a duration of 3 months, using rechargeable batteries will be both environmentally friendly and cost-effective.

Due to the considerations above, we have decided to use rechargeable Nickel-metal Hydride batteries. Commonly found rechargeable AA batteries have a rated capacity of around 2000mAh and provide a voltage of 1.2V. Should a higher voltage be required by the devices, we can either use a voltage regulator or 3 of these batteries in series.

Section 3.5 Algorithms and Libraries

This section will describe the algorithms and libraries we plan to use on Beetle BLE to retrieve data from the sensors.

1. MPU-6050 IMU:

As I2C communication will be employed, the main library used will be I2CDevLib by Jeff Rowberg. This library is found at <https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050>.

2. IR transmitting and receiving:

For transmitting and receiving IR signals between guns and vests, we will be adapting and modifying the Arduino-IRremote library found at <https://github.com/Arduino-IRremote/Arduino-IRremote>.

3. Feature Extraction

As a form of data pre-processing, feature extraction will be done on the Beetle BLE. We will be using the library found here <https://docs.edgeimpulse.com/docs/tutorials/continuous-motion-recognition>, without the machine learning portion. This will allow us to only transmit data that the library thinks might be an action.

Section 4: Hardware AI [Pranav Venkatram]

In this section, we discuss the functioning of the Hardware AI subsystem. This component is meant to identify particular hand actions performed by the user in order to augment the gameplay. We intend to achieve this by using IMUs sensors to sense hand actions and classify them using an appropriate machine learning model. We propose the following development process flow:

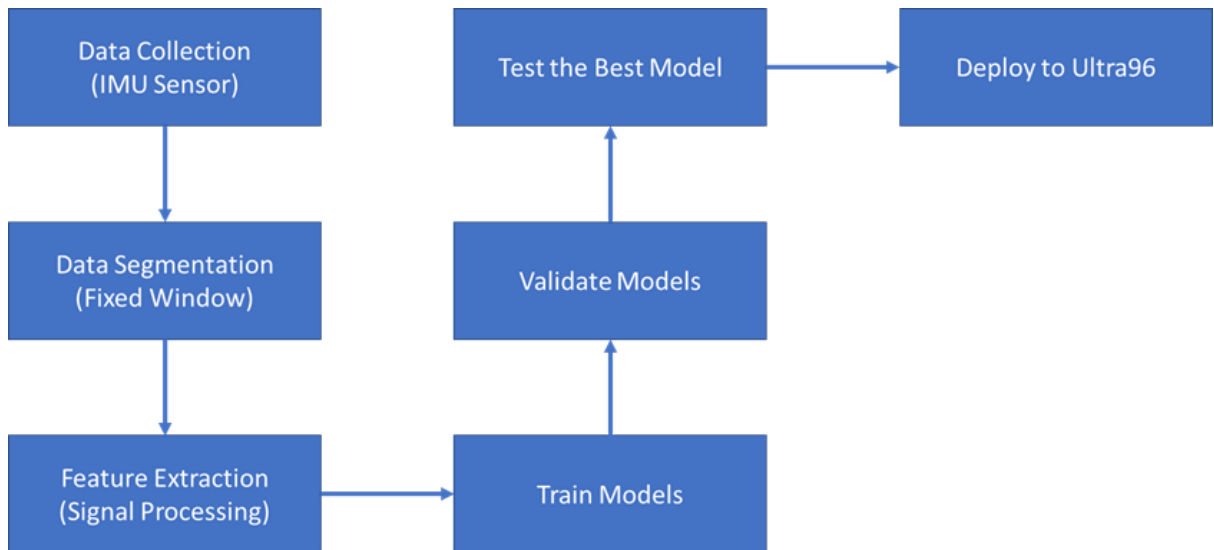


Figure 4.1: HW AI Development Process Flow

In the following subsections, we explore these steps in further detail.

4.1 Data Collection & Segmentation

Each player can perform 4 actions: Shield, Reload, Throw Grenade and Exit. Since all actions are executed by arm movements, we have chosen to mount an IMU to each player's wrist. This will provide us with movement data caused by shoulder, elbow movement and wrist rotation. Thus, we will obtain 6 classes of raw data - acceleration and angular velocity along the x, y, and z axes.

When collecting IMU data for each action, we will ensure smooth and repetitive movements. This is a key requirement for data segmentation as it ensures every segment contains relevant aspects of the action. Hence, data can be segmented into windows of fixed duration. We will begin model training with a window duration of 1000ms as a lower bound.

Further, the data will be collected while varying the speed and gait of the action. Since users may perform actions with differing speeds and posture, it is essential that the data contains such variation such that the model will be accurate. Depending on the model accuracy, the window duration can be tuned. Intuitively, players are likely to perform actions fast as they are looking to defeat their opponents. Hence, the window duration can be tuned using a small range of values such as 1000ms – 3000ms.

4.2 Feature Extraction

Raw data sensed by the IMU can be considerably large. Hence, transmitting this data from the Beetle to the Ultra96 will result in significant latency. Further, providing raw data directly to the model carries the risk of poor accuracy when sensing a player's action. Performing feature extraction on the Beetle to determine the most relevant data will reduce the number of packets transmitted (lower latency) as well as provide the model with more apparent patterns in the data (reduces model complexity and improves accuracy).

Based on literature by Edge Impulse[24], feature extraction on IMU data can be done using Spectral Analysis. Since this technique has been experimentally validated on acceleration data, we will adopt it for our use case. The following features are extracted for each axis from gyroscope and accelerometer data, giving 24 features per segment of data (3 axes x 2 sensors x 4 features = 24 features):

- a. Root Mean Square
- b. Skewness – distortion with respect to the normal distribution of the data
- c. Kurtosis – probability of outliers in a distribution
- d. Spectral Power of 1 FFT bin of interest

As a result, we will only need to transmit 24 data points per second to the Ultra96 for a single IMU. As an additional optimization, we could transmit a window of sensor data only after a significant change in acceleration occurs, signifying an action. This will reduce the load on internal communications.

4.3 Model selection and optimization

Based on literature, we have decided to explore the following Machine Learning classifiers as they have been experimentally proven to be accurate on IMU data:

- a. Random Forest [17]
- b. Feedforward multi-layer Neural Network [20]

To select the appropriate model, factors such as model accuracy and precision will be considered. First, both models will be trained, and their respective parameters will be tuned e.g., Number of trees in the Random Forest, Number of layers in the Neural Network etc.

Next, the models' accuracy on the validation dataset will be compared for various precisions. This evaluation will provide us with an understanding of how each model's accuracy changes with respect to precision. Based on this, we will select an accurate model (at least 95%) as well as determine the appropriate precision to use.

An additional speed optimization is parallelism. While inference can be sped up, additional hardware will be needed, and more power will be consumed. Since the FPGA on the Ultra96 will be dedicated to our ML model and is constantly powered, parallelism is suitable for our use case.

4.4 Model Training

While other subsystems are being developed, the model will be trained on sample IMU data. The chosen dataset was obtained from Kaggle[17] and contains accelerometer and gyroscope

data from an IMU sensor mounted on an individual's wrist. The dataset contains 27 actions performed, of which certain actions are similar to what we need to sense e.g., throw action for grenade.

Hence, this dataset is an apt starting point to train the model. First, the sample data will be split into train, validation and test datasets using the following proportions: 50%, 25%, 25%. After the hardware subsystem is complete, we will collect our own IMU sensor data and re-train the model using this.

Since the model will only be deployed on the Ultra96, it will be trained and tested on a laptop using the Python stack (PyTorch or TensorFlow). The training data and their corresponding labels will undergo feature extraction using Spectral Analysis as mentioned earlier. Next, it will be provided to the classifier for training.

To simplify the model design and evaluation process, we will leverage the Edge Impulse ML platform. The platform is designed to be a user-friendly pipeline to collect data, train models and deploy machine learning algorithms to the edge. Using their development pipeline and dashboards, we will implement the above models using TensorFlow and perform training, validation, and testing. The corresponding Python code and the trained model will be exported from Edge Impulse and will later be converted to HLS for deployment on the Ultra96 FPGA.

4.5 Model Validation & Testing

We will perform validation by running the trained Random Forest and Feedforward Neural Network on the validation dataset and compare their accuracy.

Next, we choose the more accurate model and run it on the test dataset. Our goal is to achieve an accuracy of at least 95%. Should the accuracy of the model be less than this, we will look into iteratively improving model design and/or process the IMU data better (remove noise or extract different features) such that the classifier receives data with more exploitable patterns.

Eventually, when the model is deployed to the FPGA, we also need to validate the synthesised model by ensuring its classification output matches the model trained on the Python stack. We intend to use a Python package called hls4ml[21] which converts a TensorFlow model to HLS and compiles it to produce a bit-accurate emulation on the CPU. Since it can be evaluated in Python, the process of validating the HLS form of the model is greatly simplified.

4.6 Anomaly Detection

The model will be trained to classify input data as one of four types of actions. Hence, when a user does not perform a valid action, we need to ensure that the model does not accidentally classify this noise and cause an unintended action. Upon classification, the model will provide a label as well as the probability of the input data belonging to this class. As a postprocess, we will check if the probability is at least a certain threshold for the classification to be considered valid. If not, the classified action is discarded as an anomaly. Since we train the model to achieve at least 95% accuracy, we will begin by setting the threshold to 90% and observe if this threshold is appropriate. Based on the system's sensitivity to invalid actions, the threshold will be adjusted.

4.7 Ultra96 Synthesis & Simulation

As stated earlier, we intend to use hls4ml to simplify the process of deploying our model to the FPGA. Using hls4ml, we will first generate an HLS project from the Keras model. Next, hls4ml will run the following steps and generate an IP core [21]:

- a. C simulation – Pre-synthesis validation that the C program correctly implements the Keras model
- b. C/RTL synthesis
- c. C/RTL co-simulation – Post-synthesis validation that the RTL functionally matches the C code
- d. Vivado synthesis – Transform RTL into gate level design

Hence, we will be certain that the bitstream running on the FPGA will classify data the same as the Keras model.

4.8 Realising the Neural Network on FPGA

Once our Feedforward Neural Network is trained, validated and tested on the Python stack, it is ready to be deployed to the FPGA. Traditionally, this is done by inspecting the weights of the model and manually implementing the model logic in C++ HLS code e.g. each layer of the Neural Network can be manually implemented.

To optimise our deployment process, we have chosen to use hls4ml to convert a Keras model to an IP core directly. This will allow us to treat the classifier model IP as a modular block. By using such a tool, we can deploy complex models faster while reducing the likelihood of bugs due to manual translation. The following diagram indicates the aspects of the deployment that are abstracted and simplified by adopting hls4ml:

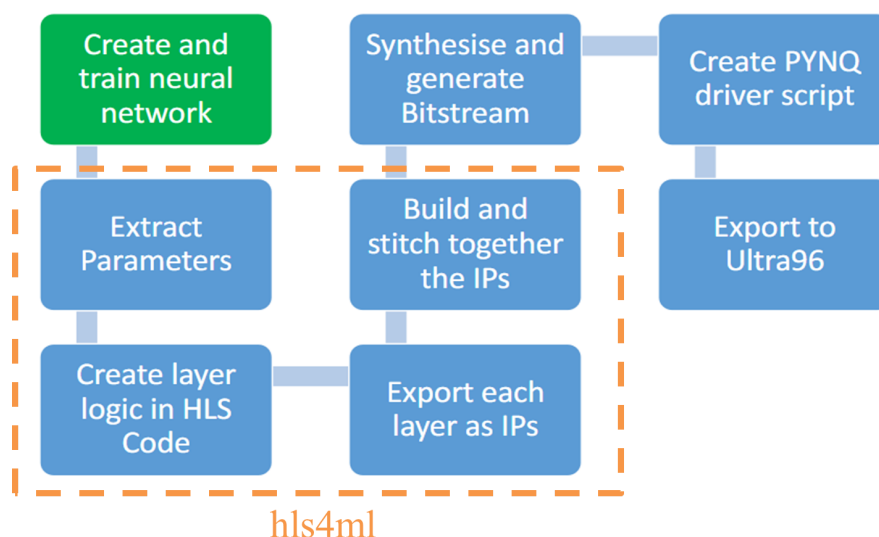
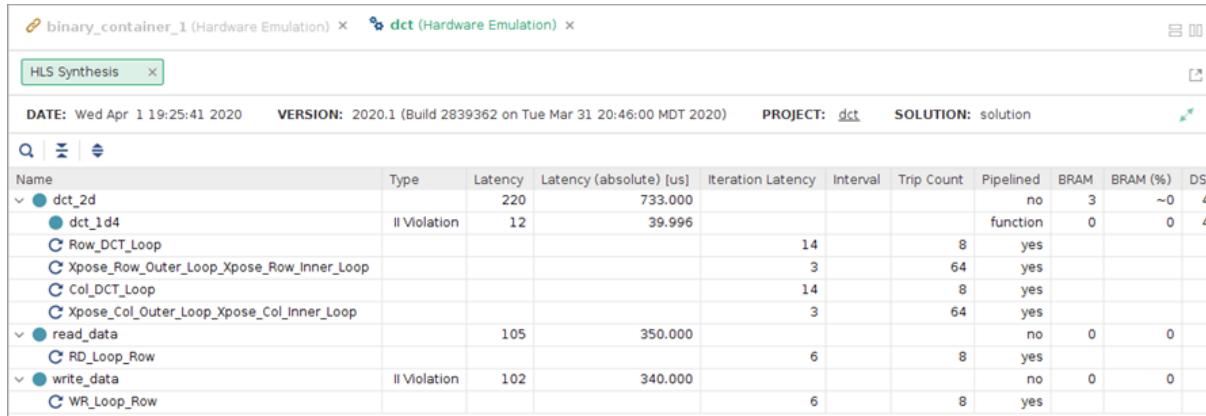


Figure 4.2 – Block Diagram of Model Deployment Process (modified from [16])

After generating the IP, we will integrate it with another HLS project containing code to read input data and write the classified action using the DMA such that the model can communicate with external subsystems.

4.9 Design Evaluation – Timing, Power Area

When Vivado performs synthesis, it generates a HLS report, detailing aspects such as resource usage, latency etc. This report can be viewed through Vitis, for example:



The screenshot shows the Vitis HLS Synthesis report interface. At the top, there are tabs for 'binary_container_1 (Hardware Emulation)' and 'dct (Hardware Emulation)'. Below the tabs, the report title is 'HLS Synthesis'. The report header includes the date 'Wed Apr 1 19:25:41 2020', version '2020.1 (Build 2839362 on Tue Mar 31 20:46:00 MDT 2020)', project 'dct', and solution 'solution'. The main table lists various design components with their timing metrics.

Name	Type	Latency	Latency (absolute) [us]	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSP
▼ dct_2d		220	733.000				no	3	~0	4
● dct_id4	II Violation	12	39.996				function	0	0	4
⌚ Row_DCT_Loop				14		8	yes			
⌚ Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop				3		64	yes			
⌚ Col_DCT_Loop				14		8	yes			
⌚ Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop				3		64	yes			
▼ read_data		105	350.000				no	0	0	
⌚ RD_Loop_Row				6		8	yes			
▼ write_data	II Violation	102	340.000				no	0	0	
⌚ WR_Loop_Row				6		8	yes			

Figure 4.3 – Sample Vivado Report [35]

Using the above report, the design's timing is measured by the Latency metric, and the area is determined by the number of Flip Flops (FF), Look Up Tables (LUT), Block RAM (BRAM), Digital Signal Processing blocks (DSP), used. Since we are using hls4ml for synthesis, we will use its corresponding Python method to obtain the Vivado HLS report.

To analyse the design's power usage, we will use the PYNQ Python package to monitor power rails in real time. Using the test dataset that was segmented earlier, we can evaluate the design's power usage by monitoring the power rails while inference is performed.

Section 5 Internal Communications [Braden Teo Wei Ren]

5.1.1 Managing Task(s) on Beetles

The three Beetles are in charge of different sensors - IR sensor, IMU, and gun trigger sensor. The Beetle attached to the IMU would need to perform feature extraction (covered in Section 4) first. Then, the resultant data would be packaged and sent to the laptop. The Beetle attached to the IR sensor and the one attached to the gun trigger sensor would not need to do feature extraction, and would be immediately packaged and sent to the laptop once new sensor data is available.

5.1.2 Managing Threads on Laptop

Multithreading is required to maintain concurrency of internal communications. Four threads would be running on the laptop – three threads that each receive/transmit data from/to each Beetle, and one thread to transmit data to the Ultra96 via external communications. These threads are implemented using Python's *threading* module. A point to note is that due to the Global Interpreter Lock (GIL) in Python, only one thread holds control of the Python interpreter at any one time, preventing threads from truly running in parallel. This would likely not be an issue for all of the threads, as the IR sensor and gun trigger sensor would not be constantly receiving new data, while the Beetle attached to the IMU would be transferring its packages at a lower rate, thus the Python interpreter would be able to switch between threads and not lose any data. However, if upon testing we notice that the threads cannot keep up with the data rate, we would adopt multiprocessing instead, where we use processes instead of threads, and they run in parallel, bypassing the GIL.

5.2 Setting up and Configuring BLE interfaces

5.2.1 Pre-requisite Information

Generic Access Profile (GAP) controls Bluetooth connections and advertising, allowing a device to be visible to other devices [40]. The peripheral (Beetle) would transmit an advertising packet outwards in a specified advertising interval to inform the central device (laptop) of its presence. The laptop would continuously scan for those packets and then establish a dedicated connection between the 2 devices. Since the BLE chip on the Beetle uses the Generic ATtribute Profile (GATT) [42], the Beetle would then stop advertising itself, until the connection between the Beetle and laptop is broken [43]. GATT uses the concept of Services and Characteristics to transfer data between devices.

5.2.2 Specific steps to configure BLE interfaces

BLE connections between the central device (laptop) and each peripheral device (3 Beetles) are set up individually.

By following the steps detailed below [44], the 3 Beetles would be set up as peripheral devices.

1. Connect a Beetle to the laptop.
2. Open the Arduino IDE.
3. Select the correct serial port.
4. Open the Serial Monitor, and select "No line ending" and "115200 baud".
5. Type "+++" in the dialog box and click "Send".
If successful, you should receive the "Enter AT Mode" message.
6. Select "Both NL & CR" and "115200 baud".
7. Type the AT command "AT+SETTING=DEFAULT" in the dialog box and click "Send".

If the BLE connection is successfully set up, you will receive the "OK" message.

If you receive "ERROR CMD" instead, check whether the command you have typed is correct and try sending it again.

8. Type "AT+MAC" to request the MAC address of the Beetle. Copy the MAC address down.
9. Input "AT+EXIT" to exit AT mode.
10. Repeat steps 1 to 9 for all 3 Beetles.

On the laptop, we would use the Bleak library to perform BLE functions. Bleak is an acronym for Bluetooth Low Energy platform Agnostic Klient, and it is capable of connecting to BLE devices acting as GATT servers.

Specifically, Python invokes the *connect()* method of the BLE Client for BlueZ on Linux to connect to the GATT server (Beetle) via the previously noted MAC address [45]. Call the *connect()* method to connect the two other Beetles too.

As such, the BLE interface is now set up and configured.

5.3 Detailed Communication Protocol

5.3.1 Packet Type

The following table illustrates the packet types we would be transmitting via BLE.

Packet type	Packet code
ACK	0
NAK	1
Hello	2
Read	3

5.3.2 Packet Format

The packet format used for the BLE communication is as follows:

[1-bit **sequence number**][16-bit **total length**][3-bit **receiver id**][3-bit **transceiver id**][2-bit **packet type**][**data**][32-bit **checksum**]

The maximum data payload for a BLE data channel Protocol Data Unit (PDU) is 246 bytes [46], which our packet format does not exceed. The *total length* section indicates the number of bits in the *data* section to use, while the other bits in the section are just padding.

5.3.3 Baud Rate

The baud rate used is 115200 bits per second, to match the settings recommended by the official documentation in Section 5.2.2. This means that $115200/246/8=58$ samples can be transferred per second.

5.3.4 Handshaking

Assuming 3 Beetles connect to a laptop, the laptop will consistently transmit 'hello' messages to search for each Beetle, until all 3 Beetles have completed the 3-way handshake, signalling all devices are awake and data is ready to be sent from the Beetles to the laptop.

Example:

Let Laptop ID: 1, Beetle ID: 2

Laptop sends a 'hello' message to the Beetle ([0b'0][010][001][010][0b'101]), Beetle recognises 'hello' and sends ACK back to laptop ([0b'0][001][010][000][0b'011]), laptop recognises ACK and sends ACK back to the Beetle ([0b'0][010][001][000][0b'011]), signalling the devices are ready to send actual data to each other.

5.3.5 Additional Details

To access the characteristics in the BLE device, use the *get_services()* on the Arduino, which returns a data container for storing the peripheral's service complement. Then use either the *get_characteristic()* or *characteristics()* functions to read data in the characteristic.

On the Arduino, use *Serial.write()* to send new data to the laptop. Use *Serial.available()* and *Serial.read()* to receive data from the laptop [47].

For the Beetles connected to the IR sensor and gun trigger sensor, we would use periodic push of data by Arduino to the laptop, since those signals would not be continuous and are required to be quickly sent to the Ultra96, such that gunshots can be quickly registered. As for the Beetle connected to the IMU, we would use periodic polling by the laptop to get the data, as the Beetle would first be doing feature extraction, and only the resultant features would be sent over to the laptop. Thus, periodic push would be unsuitable.

The detailed protocol for sending data to the laptop would be similar to the implementation of the finite state machine shown in the lecture slides. Features such as checksum, ACK, sequence number and timeout/re-transmission would be implemented to prevent data bit error, ACK/NAK bit error, and packet loss.

5.4 Handling Reliability Issues

5.4.1 Packet Fragmentation

As stated in Section 5.3.2, the data being sent would not exceed the maximum data payload. However, if in actual tests the data sent needs to be fragmented across more than one packet, start and stop delimiters could be added into the packet format to denote the start and end of a string of data. In addition, the sequence number would also have to take up more bits. Upon seeing a start delimiter, the laptop would buffer the fragmented data as they come in, adding further blocks of fragmented data in the order of the sequence number they are given, until the stop delimiter is seen by the laptop.

5.4.2 Connection drops

The BLE connection might drop due to physical obstructions. The Beetle would send out advertising packets outwards, and if the laptop is able to scan it, the BLE connection will be re-established. After performing handshakes, the Beetles are ready to resume sending sensor data to the laptop.

Section 6 External Communications [Danzel Ong Jing Hern]

This section will explain in further detail how external communications between different subsystems are set up, the protocols used, software interfaces, how secure communications are ensured, as well their justifications.

6.1 MQTT (MQ Telemetry Transport)

MQTT was the main chosen communications protocol for external communications. MQTT is a lightweight protocol, using small fixed size headers, suitable for resource constrained, minimal bandwidth, or unreliable connection environments. [2] The publish/subscribe architecture of MQTT consists of 2 parts - the first, clients, which would consist of our relay node laptops, Ultra96 and software visualiser, and the second - brokers, which acts as a server and “post-office” for our clients to connect to, to receive messages which they are subscribed to, called topics. [1] This architecture makes message management simple across multiple devices, and streamlines complex processes making debugging easier.

Bi-directional communications where clients can both subscribe, and publish to certain topics, as well as being able to ensure secure, reliable message delivery, are features which MQTT provide, which are vital in a laser-game system. These features maximise the user experience in game by ensuring game synchronisation through preventing error snowballing, as well as preventing packets being dropped which may invalidate actions performed by the player such as shooting, resulting in the feeling of a buggy, unpolished game system for the player. Finally, the lightweightsness and efficiency for running MQTT clients [1] ensure that more computing resources can be reserved to run computationally expensive tasks, such as AR visuals, enabling inter-device communications without affecting gameplay performance. Thus, MQTT was the main communications protocol chosen for the project.

For the project, **HiveMQ** was chosen as the main MQTT broker [3]. While other MQTT brokers are available, HiveMQ provides a free, cloud-based MQTT broker which is able to connect up to 100 devices. Furthermore, the HiveMQ broker is able to communicate effectively with a large number of client libraries spanning multiple languages such as Python, C++ and Arduino, which is required to provide reliable software interfaces between our devices, and the broker. [3] The MQTT client libraries supported by HiveMQ which will be used are: **Arduino PubSubClient** [4] for MQTT communications between the relay node and broker, and **Paho Python**, the Python MQTT client library [5].

Being a free service also reduces cost for the project which can be allocated elsewhere, while being a cloud-based broker makes setting up connections simple, and accessible, and makes real-time data monitoring and debugging simpler. As such, HiveMQ was chosen as the primary broker for our external communications, between components.

Further details on the message protocols and MQTT message topics between components such as Ultra96, relay node, and visualiser will be explained in the sections below.

6.2 SSH Tunnelling

As the Ultra96 sits on a separate network behind the NUS SoC Firewall, the device will have to be port-forwarded to enable devices on other networks to communicate with the Ultra96. SSH tunnelling was chosen as it allows secure, remote web access for devices without exposing local ports onto the internet [6].

To achieve this, a SSH Tunnel will be created on the Ultra96 to enable connections from other devices. After logging into the Ultra96 via SoC Sunfire SSH client, a bash script on the terminal can be run on the Ultra96 to port-forward, and expose a port for external data to reach the Ultra96 as follows:

```
ssh -L < CONNECTING IP: PORT >:< ULTRA96 IP: PORT TO EXPOSE > < SUNFIRE CREDENTIALS >
```

Figure 6.1: Bash Script to Enable Port Forwarding

Running the script will enable tunnelling into the Ultra96 until logged out. For the implementation of the project, the automatic setting up of an SSH tunnel via Python, using the `sshtunnel` library [7], running on the Ultra96 will also be explored, to reduce the need for manual scripting, and improve the system initialisation flow.

6.3 Concurrency Handling

Socket communications are typically blocking, “holding” up the system until a message is received. In our system, multiple communication connections may have to be made. For example, our game server will have to subscribe to multiple MQTT topics, the evaluation client will have to communicate with the MQTT broker as well as set up a normal TCP web socket connection, and finally, the 2 relay laptops would have to connect with up to 3 beetles each, transmitting data over bluetooth, as well as communicate to the Ultra96 simultaneously.

In order to allow concurrent communications with multiple connected sockets, multithreading can be implemented, where each connected web socket has its own dedicated thread spawned to handle sending/receiving of data. Multithreading will be implemented via the default Python `threading` library [8].

6.4 Communications Between Sub-Systems

This section of the report will go into detail on how each sub-system (Evaluation server, Relay Nodes, Software Visualiser and Ultra96), communicate with each other. The diagram below illustrates high-level communications between these sub-systems.

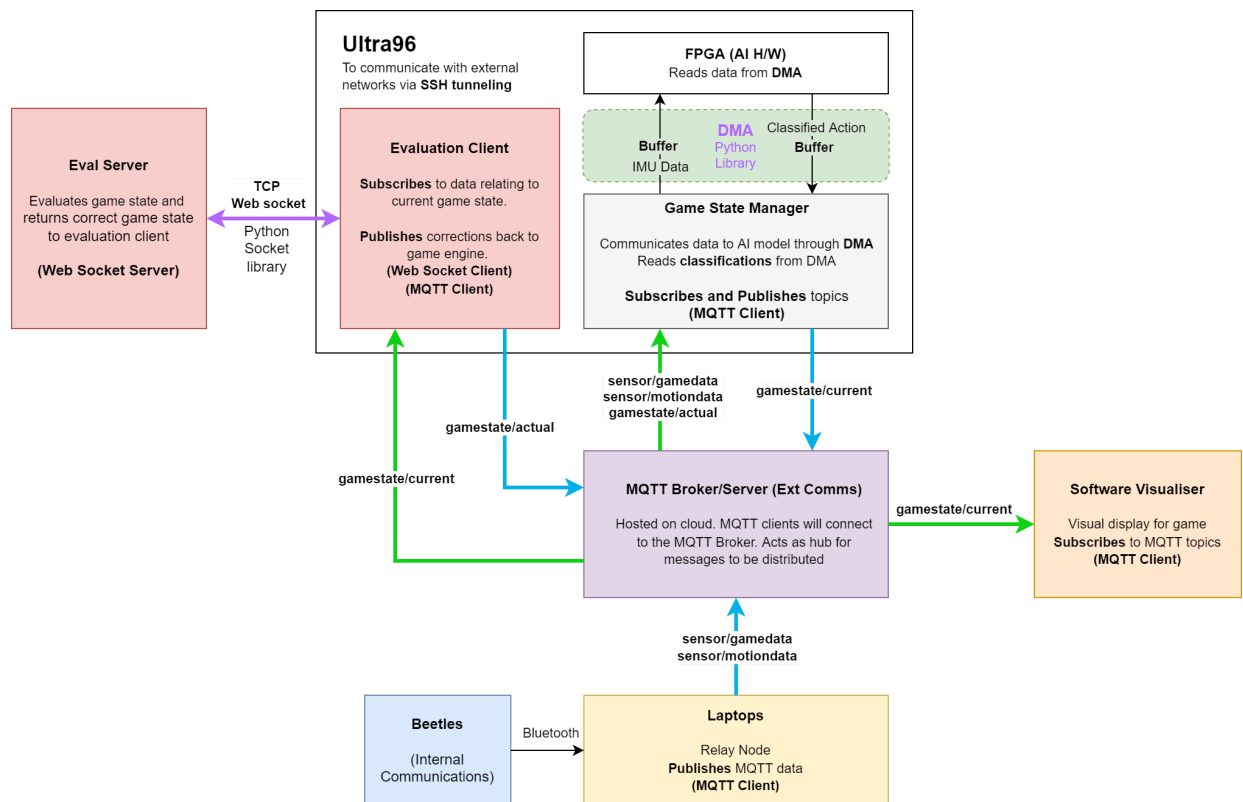


Figure 6.2: Communications Overview

MQTT communications are labelled using green and blue arrows, where green arrows represents **subscription** to messages, and blue arrows represent **publishing** of messages. The

proposed **topics** which each sub-system publishes or subscribes to is listed on the arrows. In MQTT, a topic is a UTF-8 encoded string which the broker uses to filter messages to the respective clients which have subscribed to the corresponding topic [9].

6.4.1 Secure Communications

Data packets from our application will be sent over the network, and as such, it is important to encrypt our data packets to prevent information hijacking, and act as a way to authenticate if received data packets do belong to us.

As such, data will not be sent over in clear text, but instead, encrypted with the AES (Advanced Encryption Standard), symmetric block cipher, in the CBC (Ciphertext Block Chaining) mode, where a plaintext block gets XOR-ed with the previous ciphertext block before encryption.

Encryption of clear text data will be done following the steps as shown in the diagram below.

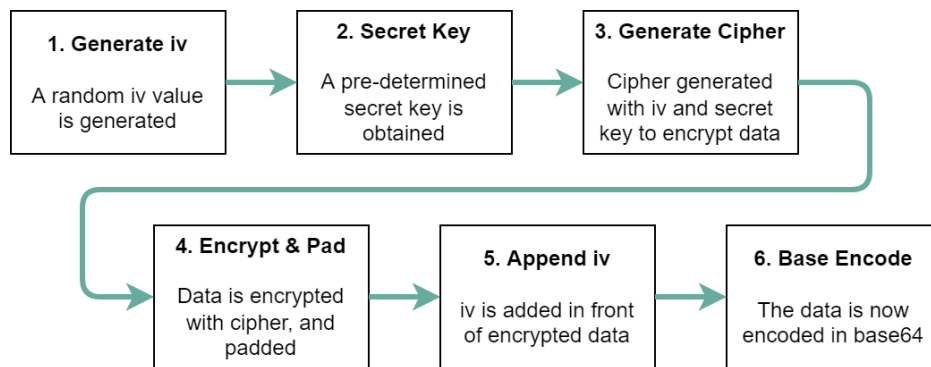


Figure 6.3: Data encryption procedure

The iv (Initialization Vector) will be of block size 16 bytes as default for AES, and the predetermined secret key will be of either 16, 24 or 32 bytes, where 1 byte = 1 character. For AES encryption, the **pycryptodome** library [10] will be utilised. Base64 encoding will be done using the default **base64** Python library [11].

As for decryption, the raw cipher text will first be base64 decoded. The iv value will then be extracted from the first 16 bytes of the decoded string. Similar to encryption, the AES cipher will then be created using the secret key, and iv. Finally, the encrypted data will be decrypted, and unpadded to return the clear text data for use.

This encryption and decryption process will be used in the TCP web socket communications between the evaluation client running on the Ultra96 to the evaluation server, and extended to the messages being sent via MQTT, between the game state manager, software visualiser, and evaluation client, to protect our data over the network.

While AES is a moderately resource intensive encryption algorithm [12], the data packets going from the relay node to the Ultra96 will not be encrypted, due to the expected high volume of data being transferred between the 2 devices. Having to decrypt and encrypt the raw data from the sensors may introduce unnecessary overheads in packet size due to padding, as well affect performance, increasing latency.

6.4.2 TCP

TCP (Transmission Control Protocol), was the main protocol chosen for communications between all our devices. While UDP (User Datagram Protocol) is simpler and faster due to smaller overheads in header size, which benefits streaming large amounts of data, it is an

unreliable service with no guaranteed delivery, when compared to TCP, which is connection oriented, and is able to guarantee ordered, data delivery [13].

In the context of a game, while speed is important to reduce latency, the importance of ensuring no packet loss is far greater, as having lost packets could mean player shots and actions not being registered, leading to a worse user experience. Thus, the benefits of guaranteed delivery provided by TCP outweigh that of UDP, and as such, TCP was chosen. For external communications using MQTT, TCP is the default, and the usage of UDP is unavailable.

6.4.3 Message Format

Messages and data will be set as stringified JSON objects, which will then be encrypted in the format as mentioned in *Section 6.4.1 Secure Communications*.

Apart from the payload data, the data packet will be prefixed with a 'header', which will contain the size of the expected payload data in bytes. A single '_' underscore denotes the end of the header, and the start of the encrypted payload of size as stated in the header.

The entire data packet will then be encoded in base64, in a utf-8 format before being sent.

6.4.4 Ultra96 and Eval Server Communications

Communications between the Ultra96 and the evaluation server will be done via TCP web socket connection. To achieve this, the Ultra96 board will run a Python program called the "Evaluation Client". The evaluation server and client will run the Python **socket** library [14]. Concurrent communications between the evaluation server, evaluation client on Ultra96, and from evaluation client to game state manager will be achieved via Python multithreading, as explained in section *6.3 Concurrency Handling*.

The evaluation client will help consolidate game state information, and prepare the packets for evaluation by the evaluation server. These preparations include encryption of the packets, according to the steps mentioned in *6.4.1 Secure Communications*, as well as formatting the message in the correct message format as mentioned in *6.4.3 Message Format*.

In order for the evaluation client to receive game state data, and send back the actual game state returned from the evaluation server to prevent cascading errors, the evaluation client will communicate with the MQTT broker. To receive current gamestate data, the evaluation client will subscribe to the "gamestate/current" topic, published by the game state manager.

The game state manager's role is to maintain, update and act as a source of truth for the current game state, which includes keeping track of shield timings. This will be implemented as a Python program, also running on the Ultra96. Other than publishing the current game state data to the MQTT broker, the game state manager will also subscribe to the "gamestate/actual" topic, published by the evaluation client program, to ensure game state is synchronised with the evaluation server.

Payload data published by the evaluation client, as well as game state manager onto the MQTT broker as the "gamestate/current" and "gamestate/actual" topics will follow the same data packet JSON format as that which is expected to be received by the evaluation server, sent from the evaluation client via TCP connection. The data format illustrated in figure 6.4 below. The message formats and protocols explained here also extend to the software visualiser

```
{'hp': 4,
  'action': 'shoot',
  'bullets': 3,
  'grenades': 1,
  'shield_time': 3,
  'shield_health': 10,
  'num_deaths': 2,
  'num_shield': 1}
```

Figure 6.4: Gamestate Data JSON Format

6.4.5 Ultra96 and Relay Node Communications

In total, 2 relay node laptops will be used, each connected to 3 separate beetles via bluetooth. Multithreading will be used to achieve this.

External communications from the relay node laptops to the Ultra96 however, will be done through the use of MQTT protocol, to the MQTT broker. To allow for better data management on the game state manager, data being published from the relay nodes to the Ultra96 will be separated 2 different topics, “sensor/gamedata” and “sensor/motiondata”. The “sensor/gamedata” topic will contain data from the players gun and vest, while “sensor/motiondata” will contain all data collected from the IMU for motion controls.

The messages sent from the relay node onto the Ultra96 game state manager will not be encrypted due to the high expected volume of data, thus reducing latency incurred from the encryption and decryption process. Data from being sent from the relay nodes to the Ultra96 game state manager will follow the format as shown in figure 6.5 below. The ‘player’ field will change according to whether the data came from player 1 or 2. For the “sensor/gamedata” topic, the ‘data’ field in the JSON object will be either ‘shoot’ to reflect a shot being fired, or ‘hit’ which denotes a player getting hit.

For IMU data, the respective XYZ data will be appended to the ‘data’ field on the JSON object, and sent to the Ultra96 over MQTT topic “sensor/motiondata”. On message arrival to the game state manager on Ultra96, the motion data will be passed directly to the AI classifier, via DMA (Direct Memory Access), using the Python Pynq library for DMA [15]. The classified action will also be returned to the game state manager, via DMA.

```
{ 'player': 1, 'data': 'shoot' }
```

Figure 6.5: Relay Node to Ultra96 Data Format

6.4.6 Ultra96 and Visualiser Communications

The software visualiser will provide the player with important gameplay information, visually. The data relating to the current game state will be communicated to the visualiser from the game state manager running on the Ultra96, via MQTT.

A MQTT client will run on the software visualiser, which subscribes to the “gamestate/current” topic published by the game state manager. Subscribing to the topic would allow the software visualiser to continuously listen for game state updates, and update the UI elements on the fly quickly, when the state is altered.

Game state data would be encrypted as specified in section 6.4.1, and would follow the message format as mentioned in 6.4.3. Game state data would follow the same JSON object form as shown in figure 6.4 above.

Section 7 Software Visualizer [Hu Xuefei]

An important part of a game is how the game play is visually displayed to the players. In our project, the laser tag game will be visualised on the player's phone to give the players an image of the game and hence a more immersive experience.

7.1 Visualizer Design

Before designing the visualizer, we conducted a survey with 5 NUS students from different genders who either have rich experience in phone shooting games or virtual reality shooting games. Each person was asked to share their experiences about the game they like the most and the game they dislike the most.

From the discussion, all 5 participants mentioned they prefer all relative data or state of their character is clearly displayed or easy to pull out as they need to refer to these information before they make a decision or make a movement, to quote one of the participant: 'If I still have a lot of blood, there is no point to waste my shield ... I can't always find a shelter just to pull out my state to see the most basic things like how many bullets I have, it's definitely better if I can see my basic states all the time.' Another participant mentioned 'If I can't see what others do when I am supposed to see it, then this game is not fair to me', therefore another important thing mentioned is that all the actions both from themselves or their opponents should be displayed on the screen. The players need to see others' actions when they are not sheltered to anticipate what their opponents intend to do so they can make quick movement, they also need to see their own actions so they know they have made a clear move and their intention is clearly reflected in the game. A participant shared his experience with VR shooting games with headsets, 'In these games, I won't physically move around or turn around, it's very insecure to move around when I can't actually see the environment, so if I am wearing a headset, I always just sit there.'

Therefore, after getting some information about players' requirements about the game visualizer, we came up with the following rough design.

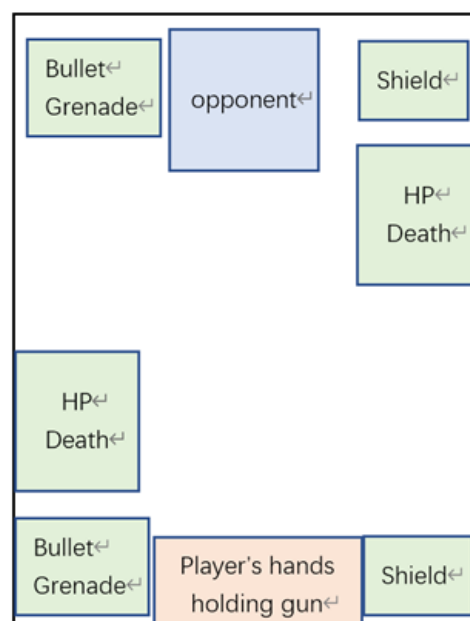


Figure 7.1: rough design of visualizer UI

The player's hands with the gun will be at the bottom centre of the screen to provide a more immersive first-person view. The following information of each player will be displayed: Character HP, number of deaths, number of bullets left, number of grenades left, number of shields left, shield HP and time to wait before the shield is ready again. All will be displayed at the corner or side to prevent this information from blocking the player's vision.

The actions of opponents will be similar to their actual actions as the full body of the opponent, however since only hands are displayed for the player, some actions are difficult to be visible to the player himself. Therefore, to make the actions more distinguished, when the player reloads, both hands and gun will disappear and then come up to imitate putting down the gun and holding up again, and when the player uses a grenade, both hands will hold a grenade instead of a gun followed by the grenade flying away to imitate throwing a grenade. When a character uses a shield, their left hand will be holding a shield instead of both hands on the gun.

Then, the player will use their phone as the visualizer, the phone will be vertically mounted to the side of their gun. In our game, the players need to physically move around in a space and turn around to change their vision angle, therefore although a headset can be more immersive, it is safer to choose not to completely deprive players' vision in the real world. Without a headset, the player can easily make fast movements without being concerned about safety issues.

7.2 Visualizer Software Architecture

The main development engine used will be Unity. As of September 2019, 52% of the top 1,000 mobile games were powered by Unity, as well as 60% of all AR/VR content [36]. As a free engine for personal game development, Unity supports both iOS and Android platforms which can reduce our cost and at the same time incorporate the requirement of multi-platform devices.

The following is the basic design of the visualizer.

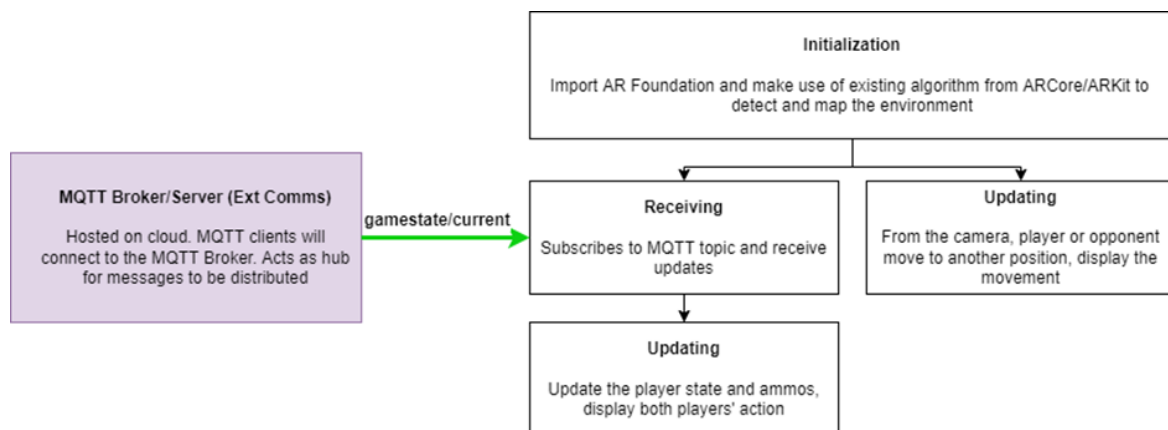


Figure 7.2: basic architectural design of visualizer

The camera of the visualizer phone will be used through AR Foundation which supports ARCore and ARKit plug-in so it can be used in both Android and IOS platform. The camera together with the phone LiDAR is used to detect the opponent and map the environment to see if there are any walls or blocks and map them out. The camera together with the algorithms provided by ARCore/ARKit will also be used to relate the player with other essential objects like the opponent and other blocks, so that the angle and distance between player and the environment can be better reflected on the screen. This is also a reason why the player's view on the visualizer is designed as a first-person view, as the camera is mounted to the gun, which is a position close to the player's eyes, so the calculated vision angle is closer to the actual one. The states and ammos will be added as UI elements and anchored to their respective positions.

During the gameplay, there will be no information transport from the visualizer to the MQTT server and there will be no information exchange between both players' visualizer devices. Each visualizer will connect to the MQTT Server and subscribe to the 'gamestate/current' channel to receive updates. The data received will include: HP, action made by the player such as reloading or action resulted by another player such as get shot, number of grenades, shields, bullets left. After receiving the data, on the visualizer device, the program will first compare the received game state against the current game state to see which part should be updated. Then only the changed part will be updated in the program and reflected on the screen, the parts not affected will not be changed. For example, when a player shoots a bullet, only the number of bullets left will be updated and the rest of the parts in charge of changing other states like HP are not affected. In this way, less reaction time is required so the program can run faster and be more sensitive to changes.

7.3 Detailed Design

Firstly, in the actual game play, the visualizer will need to receive information from the MQTT server, which requires an MQTT client to be set up in unity. Therefore, to implement this feature, the M2MQTT library[37] will be used. This library contains an MQTT client which is developed in C#, since unity also uses C# language, this is a suitable library for the visualizer programming.

Secondly, since the player needs to see the opponent in the visualizer accurately, the program needs to detect the opponent and most importantly distinguish this human body from the surrounding environment.

You can refer to this table to understand which parts of AR Foundation ai

	ARCore	ARKit
Device tracking	✓	✓
Plane tracking	✓	✓
Point clouds	✓	✓
Anchors	✓	✓
Light estimation	✓	✓
Environment probes	✓	✓
Face tracking	✓	✓
2D Image tracking	✓	✓
3D Object tracking		✓
Meshing		✓
2D & 3D body tracking		✓

Figure 7.3: parts of features supported by AR Foundation for different platform[38]

As shown in figure 7.3, among the multi-platform API defined by AR Foundation, 3D body tracking is supported for ARKit platform. The `HumanBodyTracking.cs` class [39] can be used for detecting the opponent, however, this is not suitable for all platforms, so the MediaPipe Plugin [41] which supports motion tracking will be adopted to help with human body tracking and detecting.

Thirdly, since the opponents need to make movements according to their action, an AR object needs to be placed and manipulated instead of directly using what is shown in the camera. Therefore, the human layer will be separated from the background and an AR human object will be placed at the same place. In this way, the opponent character in the visualizer can be manipulated by the programmer to reflect their actions in designed ways.

Section 8 Project Management Plan

The following table illustrates the group plan to implement the laser-tag game system.

Wk	Team target	Hardware Sensor	Hardware AI	Internal Comms	External Comms	Software Visualizer
3	Initial Design Submission	Research on suitable devices/components	Figure out data I/O with respect to model on FPGA using DMA	Research on BLE stack and multithreading	Research on MQTT and web socket	Learn Unity and set up AR Foundation Complete user survey
4	Trial Implementation	Obtain components	Determine deployment workflow by deploying a dummy model and using test data for inference	Test out connecting to Beetle, and sending packets between the Beetles and laptop	First connection between MQTT clients, and TCP web socket	Basic game UI design complete and test app on phones
5	Progress Check	IMU's and tested working Finalize shooting and target mechanisms, start implementation	Complete Neural Network deployment to FPGA	Implement the FSM into the code, and multithreads for Beetle to laptop connections	Working code in place, comms with eval server, visualiser, relay node	Complete score system and test with inputs
6	Complete Subsystems	Complete Subsystem	Analyse power usage and test power management techniques	Debug any issues, calibrate values.	Ext comms established between all subsystems	Complete and test AR effect of components Test the complete app on phone
RW	Have fun :D + Continue updating and improving each individual system to prepare for integration					
7	Improve individual components	Improve range and calibrate sensors for improved accuracy	Co-ordinate with team on Feature Extraction implementation on Beetle	Improve on FSM, use actual data from sensors.	Improve based on feedback. Start integrating comms	Fix problems from previous tests and incorporate survey feedback Test with Ultra96
8	Start system integration	Test system for Individual Progress Check and prepare for 1-player game	Train model with our own IMU data	Co-ordinate with external comms – implement last thread	Work out issues that may arise for comms	Test app and check code for individual progress check Prepare for actual game play
9	System ready for 1-player game	Prepare for tests by performing system checks, perform maintenance and debugging when necessary	Maintain model accuracy for further tests by obtaining more training data or extracting different features if needed.	Fix bugs, ensure reliability of connections, and reduce latency of the system.	Continue fixing bugs arising from communications, while improving the reliability of the system	Fix all bugs rising from testing with other parts and improve on visual and accuracy
10	Prepare and test the system for 2 player games					
11						
12						
13	Start final report					
Celebrate!						

References

1. “MQTT: The Standard for IOT Messaging”, mqtt.org, <https://mqtt.org/> [Accessed: 18 August 2022]
2. Corinne Bernstein, Kate Brush, Alexander S. Gillis, “MQTT (MQ Telemetry Transport)”, TechTarget <https://www.techtarget.com/iotagenda/definition/MQTT-MQ-Telemetry-Transport> [Accessed: 18 August 2022]
3. “Reliable Data Movement For Connected Devices”, HiveMQ. <https://www.hivemq.com/> [Accessed: 21 August 2022]
4. Nick O’Leary, “Arduino PubSubClient - MQTT Client Library Encyclopedia”, HiveMQ. <https://www.hivemq.com/blog/mqtt-client-library-encyclopedia-arduino-pubsubclient/> [Accessed: 18 August 2022]
5. Roger Light, “Paho Python - MQTT Client Library Encyclopedia”, HiveMq. <https://www.hivemq.com/blog/mqtt-client-library-paho-python/> [Accessed: 18 August 2022]
6. Sakshyam Shah, “SSH Tunneling Explained”, Teleport. <https://goteleport.com/blog/ssh-tunneling-explained/> [Accessed: 18 August 2022]
7. Pahaz, “sshtunnel 0.4.0”, GitHub <https://github.com/pahaz/sshtunnel/> [Accessed: 18 August 2022]
8. “Threading – Thread-based parallelism”, Python <https://docs.python.org/3/library/threading.html> [Accessed: 18 August 2022]
9. HiveMQ Team, “MQTT Topics, Wildcards, & best Practices - MQTT Essentials: Part 5”, HiveMQ. <https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/> [Accessed: 21 August 2022]
10. “AES”, PyCryptodome. <https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html> [Accessed: 18 August 2022]
11. “Base64 - Base16, Base32, Base64, Base85 Data Encodings”, Python. <https://docs.python.org/3/library/base64.html> [Accessed: 18 August 2022]
12. Ron Franklin, “AES vs RSA Encryption: What Are the Differences?”, precisely. <https://www.precisely.com/blog/data-security/aes-vs-rsa-encryption-differences> [Accessed: 21 August 2022]
13. Matt Cook, “TCP vs. UDP: What’s the Difference?”, Lifesize. <https://www.lifesize.com/en/blog/tcp-vs-udp/> [Accessed: 18 August 2022]
14. “Socket – Low-Level networking interface”, Python. <https://docs.python.org/3/library/socket.html> [Accessed: 21 August 2022]
15. “DMA”, Python productivity for Qynz (Pynq). https://pynq.readthedocs.io/en/v2.1/pynq_libraries/dma.html#:~:text=In%20the%20Python%20code%2C%20two,other%20for%20output%20are%20allocated.&text=Transfer%20the%20input_buffer%20to%20the,DMA%20to%20the%20output%20buffer. [Accessed: 21 August 2022]
16. S. Sasidhar, “CG4002: Computer Engineering Capstone Project Hardware AI”, NUS [Online] [Accessed: 19 August 2022]
17. Dasmehdixtr, “Human action recognition dataset,” *Kaggle*, 25-Dec-2019. [Online]. Available: <https://www.kaggle.com/datasets/dasmehdixtr/human-action-recognition-dataset> [Accessed: 19 August 2022]
18. P. P. Ippolito, “Feature extraction techniques,” *Medium*, 11-Oct-2019. [Online]. Available: [https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be.](https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be) [Accessed: 19 August 2022]

19. "hls4ml Quick start," *Quick Start · GitBook*. [Online]. Available: <https://fastmachinelearning.org/hls4ml/setup/QUICKSTART.html>. [Accessed: 19 August 2022]
20. *Continuous motion recognition - edge impulse documentation*. [Online]. Available: <https://docs.edgeimpulse.com/docs/tutorials/continuous-motion-recognition>. [Accessed: 19 August 2022]
21. "Welcome to HLS4ML's documentation!," *Welcome to hls4ml's documentation! - hls4ml 0.5.1 documentation*. [Online]. Available: <https://fastmachinelearning.org/hls4ml/index.html> [Accessed: 19 August 2022]
22. "Hls4ml tutorial," *Google Slides*. [Online]. Available: https://docs.google.com/presentation/d/1c4LvEc6yMByx2HJs8zUP5oxLtY6ACSizQdKvw5cg5Ck/edit#slide=id.ge9c66f87b4_6_1. [Accessed: 19 August 2022]
23. A. Taylor, "Accelerating your ultra96 developments!," *Hackster.io*, 01-Oct-2018. [Online]. Available: <https://www.hackster.io/adam-taylor/accelerating-your-ultra96-developments-806a72>. [Accessed: 19 August 2022]
24. *Spectral features - edge impulse documentation*. [Online]. Available: <https://docs.edgeimpulse.com/docs/edge-impulse-studio/processing-blocks/spectral-features>. [Accessed: 19 August 2022]
25. S. Turney, "Skewness: Definition, Examples & Formula," *Scribbr*, 12-Jul-2022. [Online]. Available: <https://www.scribbr.com/statistics/skewness/>. [Accessed: 19 August 2022]
26. S. Turney, "What is kurtosis?: Definition, Examples & Formula," *Scribbr*, 18-Jul-2022. [Online]. Available: <https://www.scribbr.com/statistics/kurtosis/>. [Accessed: 19 August 2022]
27. "Neural network settings," *Classification (Keras) - Edge Impulse Documentation*. [Online]. Available: <https://docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/classification>. [Accessed: 19 August 2022]
28. T. Yiu, "Understanding random forest," *Medium*, 29-Sep-2021. [Online]. Available: <https://towardsdatascience.com/understanding-random-forest-58381e0602d2>. [Accessed: 19 August 2022]
29. "DMA," *DMA - Python productivity for Zynq (Pynq) v1.0*. [Online]. Available: https://pynq.readthedocs.io/en/v2.4/pynq_libraries/dma.html. [Accessed: 19 August 2022]
30. Cathalmccabe and Cathalmccabe Split This Topic, "Tutorial: Using a HLS stream IP with DMA (Part 1: HLS design)," *PYNQ*, 25-Nov-2021. [Online]. Available: <https://discuss.pynq.io/t/tutorial-using-a-hls-stream-ip-with-dma-part-1-hls-design/3344>. [Accessed: 19 August 2022]
31. Cathalmccabe, "Tutorial: Using a HLS stream IP with DMA (Part 3: Using the HLS IP from PYNQ)," *PYNQ*, 25-Nov-2021. [Online]. Available: <https://discuss.pynq.io/t/tutorial-using-a-hls-stream-ip-with-dma-part-3-using-the-hls-ip-from-pynq/3346>. [Accessed: 19 August 2022]
32. "Fast machine learning," *Indico*. [Online]. Available: <https://indico.cern.ch/event/822126/contributions/3482454/>. [Accessed: 19 August 2022]
33. "Xilinx documentation portal," *Documentation Portal*. [Online]. Available: <https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Area-Information>. [Accessed: 19 August 2022]
34. "Xilinx documentation portal," *Documentation Portal*. [Online]. Available: <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Using-the-Flow-Navigator>. [Accessed: 19 August 2022]

35. “Xilinx documentation portal,” *Documentation Portal*. [Online]. Available: <https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Generating-and-Opening-the-HLS-Report>. [Accessed: 19 August 2022]
36. M. Dealessandri, “What is the Best Game Engine: Is unity right for you?,” *GamesIndustry.biz*, 16-Jan-2020. [Online]. Available: <https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you>. [Accessed: 21-Aug-2022]
37. GitHub. 2022. GitHub - eclipse/paho.mqtt.m2mqtt. [online] Available at: <https://github.com/eclipse/paho.mqtt.m2mqtt.git> > [Accessed 21 August 2022].
38. Docs.unity3d.com. 2022. About AR Foundation | AR Foundation | 4.2.3. [online] Available at: <https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.2/manual/index.html> > [Accessed 21 August 2022].
39. GitHub. 2022. Body-Tracking-ARKit/HumanBodyTracking.cs at master · LightBuzz/Body-Tracking-ARKit. [online] Available at: <https://github.com/LightBuzz/Body-Tracking-ARKit/blob/master/body-tracking-arkit/Assets/Scripts/HumanBodyTracking.cs> > [Accessed 21 August 2022].
40. K. Townsend, “Introduction to bluetooth low energy,” *Adafruit Learning System*. [Online]. Available: <https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap>. [Accessed: 21-Aug-2022].
41. GitHub. 2022. GitHub - homuler/MediaPipeUnityPlugin: Unity plugin to run MediaPipe graphs. [online] Available at: <https://github.com/homuler/MediaPipeUnityPlugin.git> > [Accessed 21 August 2022].
42. “CC2540,” CC2540 data sheet, product information and support | TI.com. [Online]. Available: <https://www.ti.com/product/CC2540#features>. [Accessed: 21-Aug-2022].
43. K. Townsend, “Introduction to bluetooth low energy,” *Adafruit Learning System*. [Online]. Available: <https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt>. [Accessed: 21-Aug-2022].
44. “Bluno_sku_dfr0267,” *DFRobot*. [Online]. Available: https://wiki.dfrobot.com/Bluno_SKU_DFR0267. [Accessed: 21-Aug-2022].
45. “Interfaces, exceptions and utils,” *Interfaces, exceptions and utils - bleak 0.16.0a1 documentation*. [Online]. Available: <https://bleak.readthedocs.io/en/latest/api.html?highlight=mac+address#module-bleak.backends.bluezdbus.client>. [Accessed: 21-Aug-2022].
46. “Developer help,” *Bluetooth® Low Energy Packet Types - Developer Help*. [Online]. Available: <https://microchipdeveloper.com/wireless:ble-link-layer-packet-types>. [Accessed: 21-Aug-2022].
47. Cevinus, “Serial communication with the Bluno using Bluetooth Le,” *cevinus*, 17-Aug-2016. [Online]. Available: <https://www.cevinus.com/2016/08/17/serial-communication-with-the-bluno-using-bluetooth-le/>. [Accessed: 21-Aug-2022].