EMBEDDED SYSTEMS
RESEARCH
GROUP

University of Minho
School of Engineering

Integrated Master in Industrial Electronics and Computers Engineering

ARGOS

Threat Detection Watchdog Device

Authors:
Hugo Carvalho A85156
José Mendes A85951

Professor:
Dr. Adriano Tavares

February, 2021

Acronyms

ACK Acknowledge. i, 30

ADC Analog to Digital Converter. i, 41, 42

API Application Programming Interface. i, 39, 40, 56, 59, 78, 118, 122

APS Active-Pixel Sensor. i, 31

ARP Address Resolution Protocol. i, 56, 68

BLE Bluetooth Low Energy. i, 32

BOM Bill Of Materials. i

CAD Computer Aided Design. i, 39

COCO Common Objects in Context. i, vii, 44, 45

COTS Commercial Off-The-Shelf. i, vii, 39, 40

CSI Camera Serial Interface. i, 31, 32, 38, 47

DDL Data Definition Language. i, 119

DHCP Dynamic Host Configuration Protocol. i, 56, 68

DMA Direct memory access. i, 85

DML Data Manipulation Language. i, 119

DQL Data Query Language. i, 119

ERD Entity Relationship Diagram. i, vi, 27, 59, 60

GPIO General-Purpose Input/Output. i, vii, 32, 33, 70, 125, 131

GPU Graphics Processing Unit. i, vii, 40, 45, 115

GUI Graphical User Interface. i, vii, 39, 56, 57

HW Hardware. i

I^{2}C Inter-IC Communication. i, vi, vii, 29, 30, 34, 47, 67

IC Integrated Circuit. i, 35

IDE Integrated Development Environment. i, 39

IP Internet Protocol. i, 68

IPC Inter-Process Communication. i

LDO Low-Dropout Regulator. i

LSB Least Significant Bit. i

MAC Media Access Control. i, 69

MCU Microcontroller Unit. i, 34

MIPI Mobile Industry Processor Interface. i, 31, 32, 38

ML Machine Learning. i, 9, 39, 44, 47, 115, 118

MOSFET Metal-Oxyde Semiconductor Field Effect Transistor. i

MPU Microprocessor Unit. i, 32, 70

MSB Most Significant Bit. i, 30

NN Neural Network. i, 115

ODC Object Detection Classifier. i, 39, 44, 115, 116

PGID Process Group ID. i, 75

PHY Physical Layer. i, 69

PID Process ID. i, 75, 76

POSIX Portable Operating System Interface. i

PPID Parent Process ID. i

RDBMS Relational Database Management System. i, 39

SCL Serial Clock. i, 29, 30, 35, 36

SDA Serial Data. i, 29, 30, 35, 36

SID Session ID. i, 76

SQL Structured Query Language. i, 56

SSD Single Shot Detector. i, 44

SSH Secure Shell. i, 56, 68

SW Software. i

TOCO TensorFlow Lite Converter. i, 118

UID User ID. i

URL Uniform Resource Locator. i, 56

V4L Video4Linux. i

V4L2 Video4Linux2. i, viii, 56, 66, 78, 82–84

WLAN Wireless Local Area Network. i, 69, 112

WPA Wi-Fi Protected Access. i, 56, 112

WPAN Wireless Personal Area Network. i, 69

Contents

List of Figures

Page

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter A

Chapter D

Chapter F

List of Tables

Page

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 1

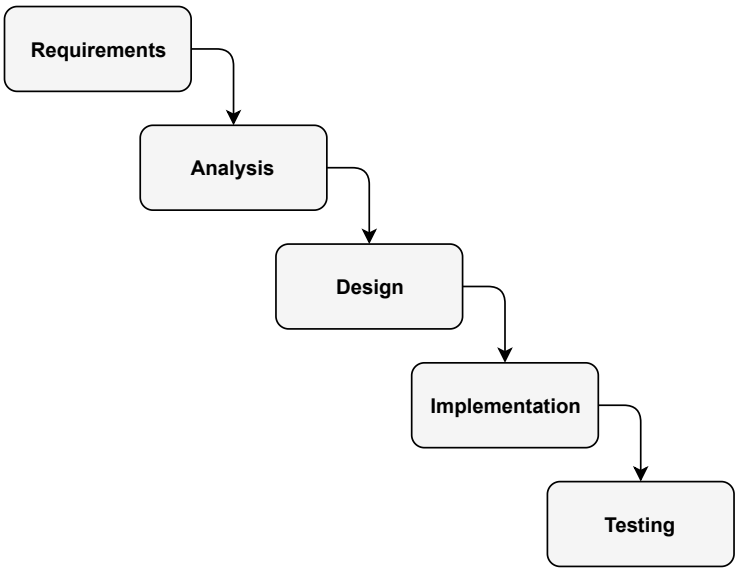
Introduction

This document consists of an analysis and presentation of the work done in the development of ARGOS.

In the first chapters, the product will be put into the proper context globally to understand the need for its existence and where it fits in the market. A high-level overview of the project will also be provided in these chapters. This is meant for future reference when designing the product as well as analyzing the report.

Further in the document, one will be able to find documentation detailing the design and implementation processes, which will hopefully be simple to understand and reproduce.

The report will culminate in the test and verification of the results achieved in the development stages, to understand the impact of the decisions that were made and where to make improvements in future work.



1.1 Problem Statement

Armed robberies and assaults are a menace to the lives and assets of the victims and require swift action to be resolved.

In those circumstances, the identification and reporting of danger are left, most of the time, at the hands of the endangered. This causes a delay between the threat recognition moment and the moment when authorities take action. This means that any technology that can eliminate that delay might help to save people and property.

In response to that matter, this product aims to help identify and report threats resorting to a weapon recognition system powered by computer vision. Upon detecting a dangerous situation, the device will send useful information like surveillance footage frames, the number of weapons identified, and their position in the environment to a human operator. The video surveillance market growth by region [1] is depicted in figure 1.1.1.

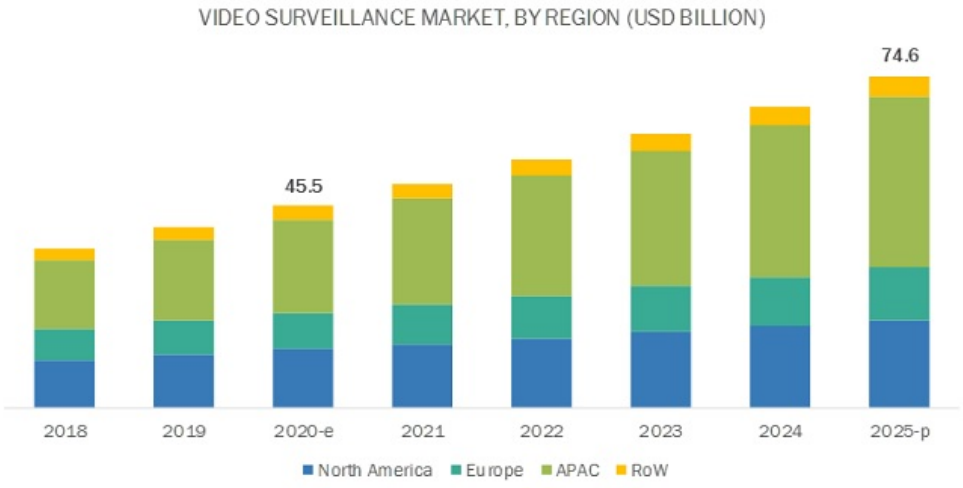


Figure 1.1.1: Video Surveillance Market Growth

1.2 Problem Statement Analysis

From the Problem Statement Analysis, we can extrapolate an overview of main components, features they provide and relationships between them. The schematic in figure 1.2.1 describes a system comprised by:

- A **Camera Module**, which will feed a stream of periodic frames from the environment into the system to be analyzed;
- A **Computational Unit** to which will be assigned the task of analyzing the aforementioned frames in search of events worth signaling and communicating those events to the Remote Client, which will be explained in further detail in the next item. It will also be responsible for storing noteworthy video frames and relevant data in a remote storage that can be accessed by itself and the Remote Client.
- A **Remote Client Back-end**, which will communicate directly with the Computational Unit to receive event alerts and a remote storage service to retrieve information about signaled events and image frames sent there by the Computational Unit.

- A **Remote Client Interface**, a Graphical User Interface that will bridge the gap between the user and the Remote Client, simplifying and streamlining the access to information about important events and video frames, thus allowing for a faster response to be delivered by human operators.
- A **Cloud Storage Service** for hosting the storage of the aforementioned video frames to be placed by the Computational Unit and retrieved by the Remote Client.
- A **Database** for the Computational Unit to store information about signaled events and the Remote Client to retrieve them.

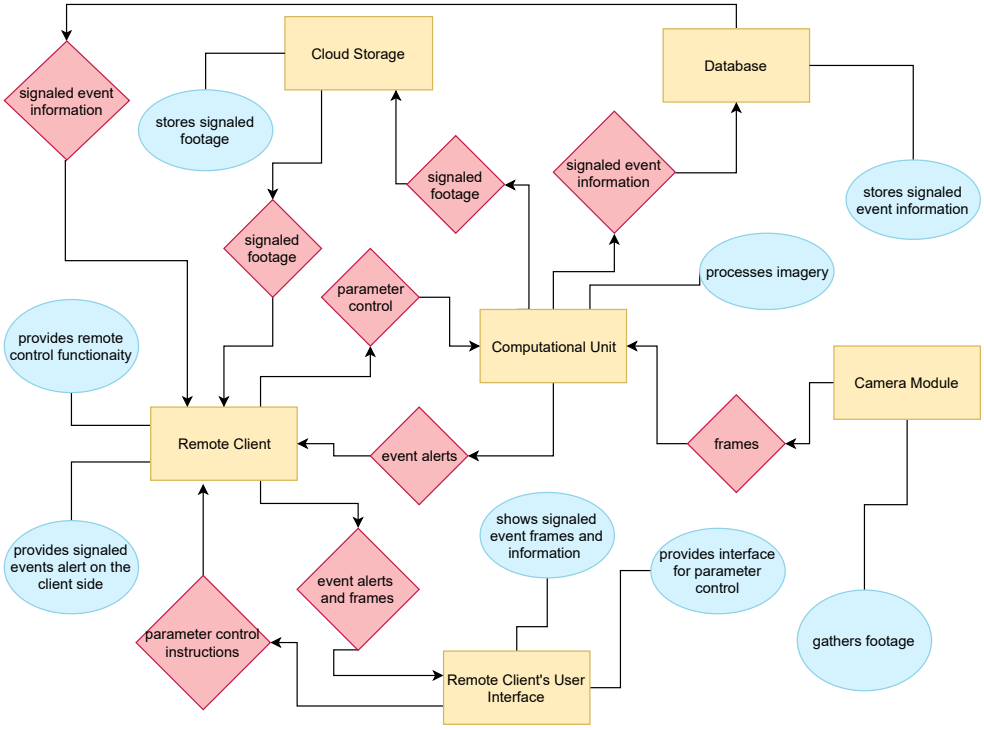


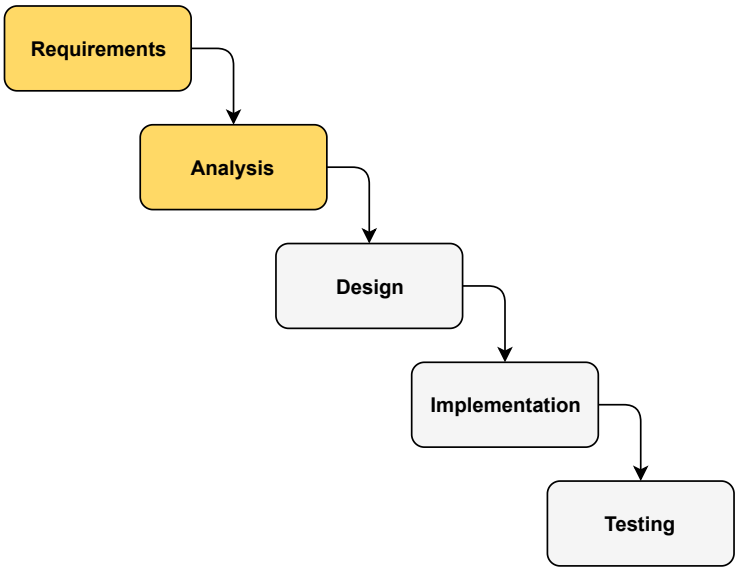
Figure 1.2.1: Problem Statement Diagram

Chapter 2

Analysis

This chapter starts with brief market research about similar products to ultimately distinguish ARGOS from its competitors.

The requirements and constraints of the project are presented as well as a general system overview to better visualize the objectives. Additionally, the system is analyzed in terms of its software and hardware architectures and divided into ARGOS and Operational Infrastructure, proceeding with the presentation of various diagrams that allow further understanding of the overall system. Following, the topic of machine learning will be approached, and the analysis stage ends with the proposed initial budget for the project and its planning through a Gantt diagram.



2.1 Market Study

The market for weapon-detecting security systems has been on the rise since violent attacks involving large groups of people have started happening more often, a fact which has put people and organizations on edge. This section will hopefully help provide a better sense of the existing technology and the features that are expected from such systems.

Evolv Express™

A weapons-detection system built to automatically screen groups of people as they walk through without slowing or stopping, analysing visitors while automatically differentiating weapons from personal items.

Figure 2.1.1: Evolv Express™

Pros

- Ability to identify hidden weapons;
- Comfort and convenience of usage for the individuals being scanned;
- Increased speed relative to a metal detector;
- Easy to setup and move.

Cons

- Limited usage scenarios.

Virtual eForce's SafeSchool™

Virtual eForce's SafeSchool is software that can be run alongside a pre-existing Video Management System to detect weapons in the scenes. It can identify handguns as well as assault riffles.

Figure 2.1.2: Virtual eForce's SafeSchool (usage example)™

Pros

- Versatility;
- Ability to discern between authorized and non-authorized weapon carriers;
- Ability to send mass notifications;
- Ability to analyse the state of important objects in the environment (such as doors being shut or ajar) and notify staff to change it in case of a threat.

Cons

- Complicated to set up.

Trueface Aware

A piece of software that can use foootage from pre-installed cameras to expand on their usefulness, with features like crowd counting, face recognition, age recognition and, more importantly, weapon detection.

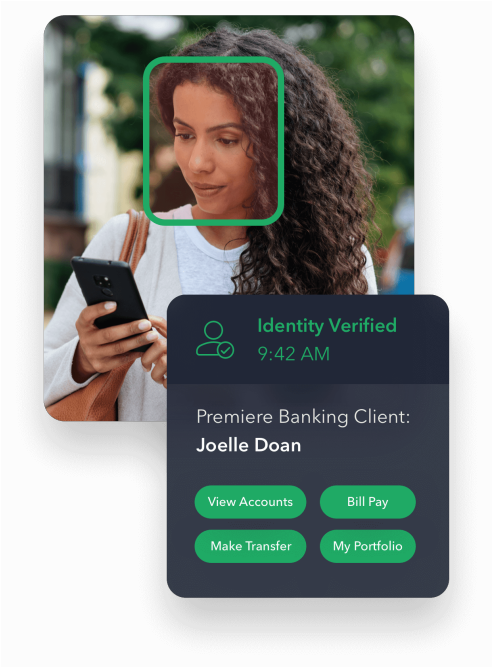


Figure 2.1.3: Trueface Aware

Pros

- Versatility;
- Ability to be programmed to alert security teams;
- Ability to detect firearms as well as knives;
- Vast array of functionalities.

Cons

- Complicated to set up.

2.2 Added Value

Although the market for weapon-detecting systems is highly competitive, the products in it are rarely cheap and simple to set up. ARGOS seeks to match the premium functionalities offered by major players but be more integrated with the hardware and, consequently, more accessible to the average user or small business.

2.3 Requirements and Constraints

The development requirements are divided into functional and non-functional if they are main functionalities or secondary ones, respectively. Additionally, the constraints of the project are classified as technical or non-technical.

2.3.1 Requirements

Functional Requirements

- **See** the predictions about the presence of weapons in the camera frames;
- **Access** the information stored remotely;
- **Receive** notifications from the camera;
- **Control** the connection remotely.

Non-Functional Requirements

- Have low power consumption;
- Have low latency in communication with the remote client;
- Have high throughput in communication with the cloud and database;
- Be upgradable;
- Be discrete and sturdy.

2.3.2 Constraints

Technical Constraints

- Use Raspberry Pi Model 4B as development board;
- Use Buildroot for cross-platform development and system configuration;
- Use POSIX Threads for Real-Time buildout;
- Use Embedded Linux;
- Implement (at least) one device driver;
- Use Object-Oriented Programming Languages.

Non-Technical Constraints

- Project deadline at the end of the semester;
- Pair workflow;
- Limited budget.

2.4 System Overview

The diagram in figure 2.4.1 represents an initial big picture of the project to facilitate objective identification. Concerning the diagram, one can identify four main blocks:

- The **Raspberry Pi Board** block
- The **Camera Module** block
- The **Operational Infrastructure** block
- The **Machine Learning Model** block

The Raspberry Pi block, powered by computer vision provided from the Camera Module will use a Machine Learning (ML) Model to detect weapons in frames. This information will be stored in the cloud and in a database whilst notifying the remote client of possible threats.

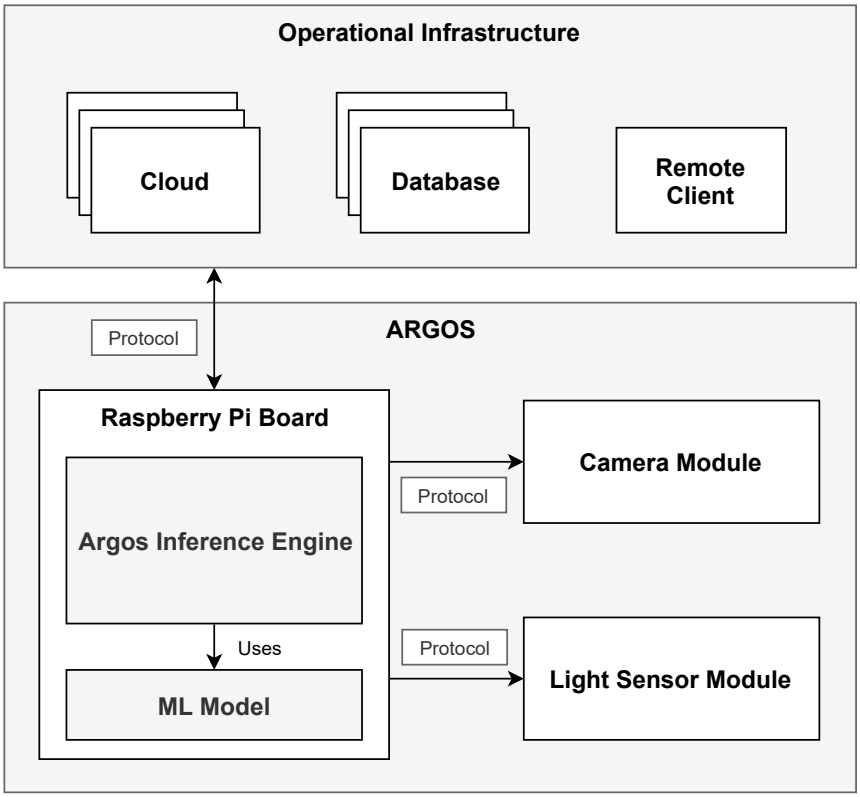


Figure 2.4.1: System Overview Diagram (Aug. in Appendix A.1)

2.5 System Architecture

2.5.1 Hardware Architecture

The block diagram in figure 2.4.1 shows a schematic of the planned hardware architecture, divided into three main blocks:

- The **Raspberry Pi** block, comprised by the Raspberry Pi board and components that will allow it to easily communicate various operational states to the user;
- The **Camera Module** block, with a camera module and a light sensor that will communicate with the Raspberry Pi through serial interfaces to provide video frames and ambient light information in order to adjust camera parameters;
- The **Operational Infrastructure** block, which is made up of all the external components needed to remotely control and manage information extracted from the environment.

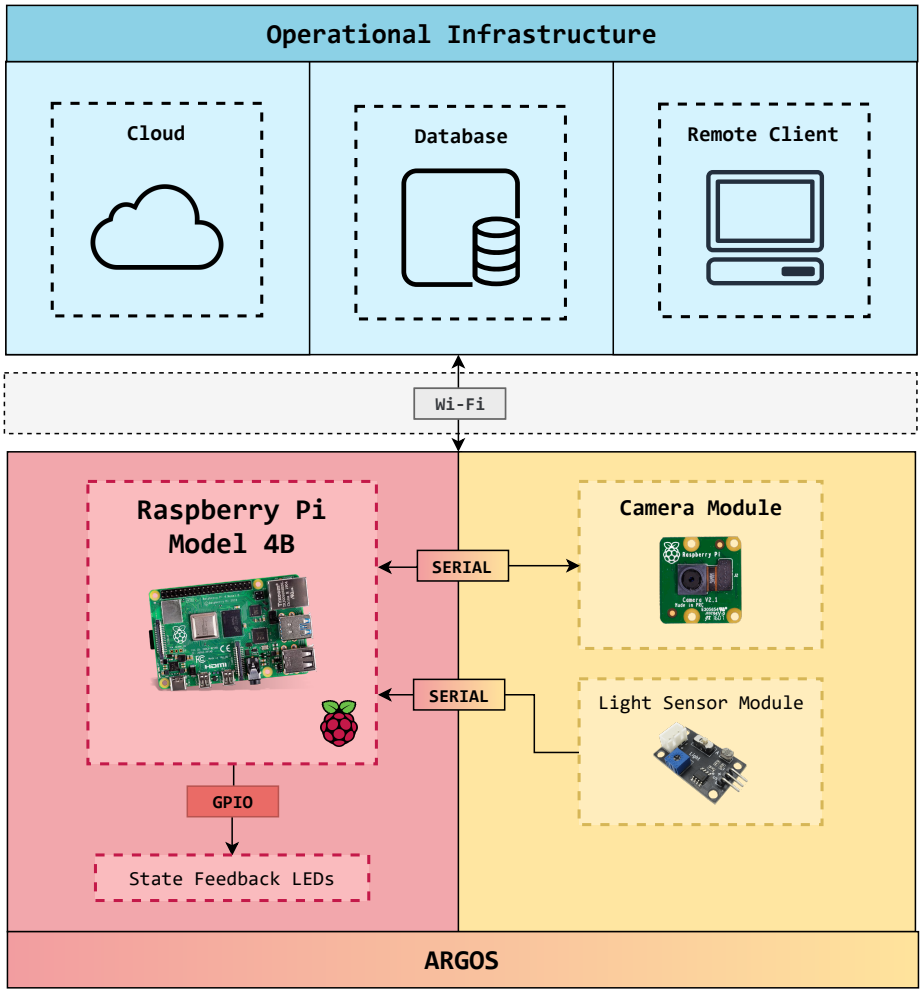


Figure 2.5.1: Hardware Architecture Diagram

2.5.2 Software Architecture

The software architecture for both the Computational Unit and the Remote Client (figures 2.5.2 and 2.5.3, respectively) is divided into three layers, those being:

- The **Operating System** layer, which is comprised by the Operating System drivers and Board Support Packages;
- The **Middleware** layer, which includes software for abstracting the lower level packages and streamlining the development of complex algorithms;
- The **Application** layer, where the core functionality of the program is built, with resource to the API's in the lower level layers.

Although the Computational Unit is the heart of the system, the Remote Client plays an important role in connecting it with the user with as little latency as possible. As such, it is essential that both prioritize good networking in their software stacks.

The Computational Unit, specifically, will need to directly interact with the physical environment, so it should have dedicated drivers for interfacing with the sensors. It will also learn to analyse footage and interpret patterns in it, so it will need to have a way to accelerate the development of Computer Vision and Machine Learning algorithms.

The Remote Client, on the other hand, is more focused on user interactions, so it will have modules for accelerating the development of a Graphical User Interface.

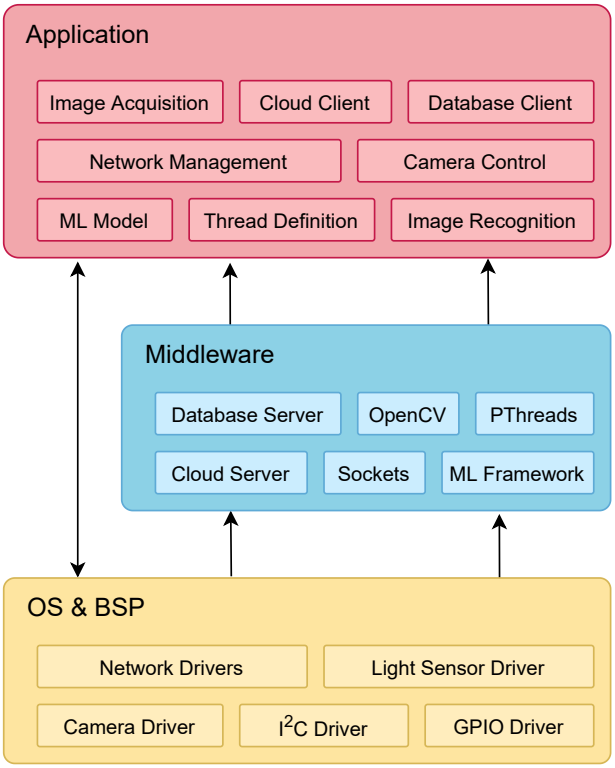


Figure 2.5.2: Software Architecture Diagram - Computational Unit

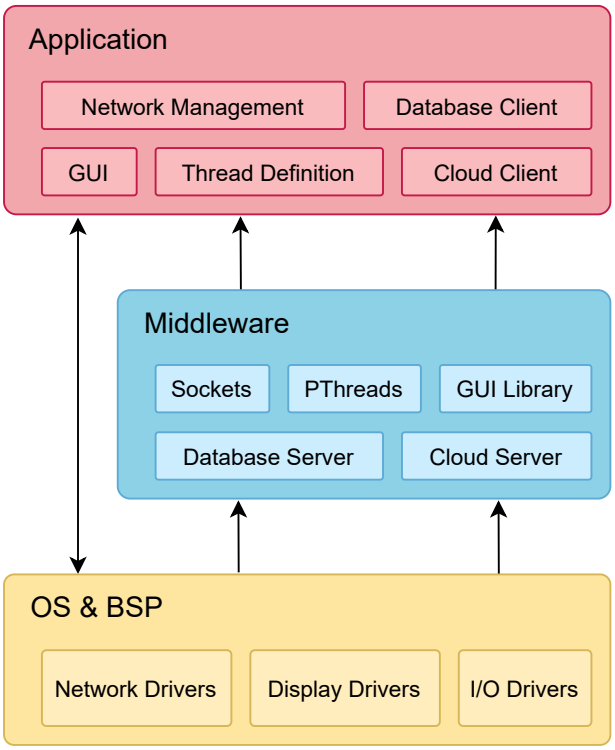


Figure 2.5.3: Software Architecture Diagram - Remote Client

2.6 Computational Unit

The Computational Unit, as discussed previously, will be in charge of gathering and analysing critical footage, uploading the information fast and notifying the client within a pre-determined deadline, so its functionality relies heavily on establishing safe, low-latency and fast connections to ensure that all deadlines are met. As such, the system should be rich in connection management and synchronous events. The **Computational Unit** is highlighted in the system overview of figure 2.6.1.

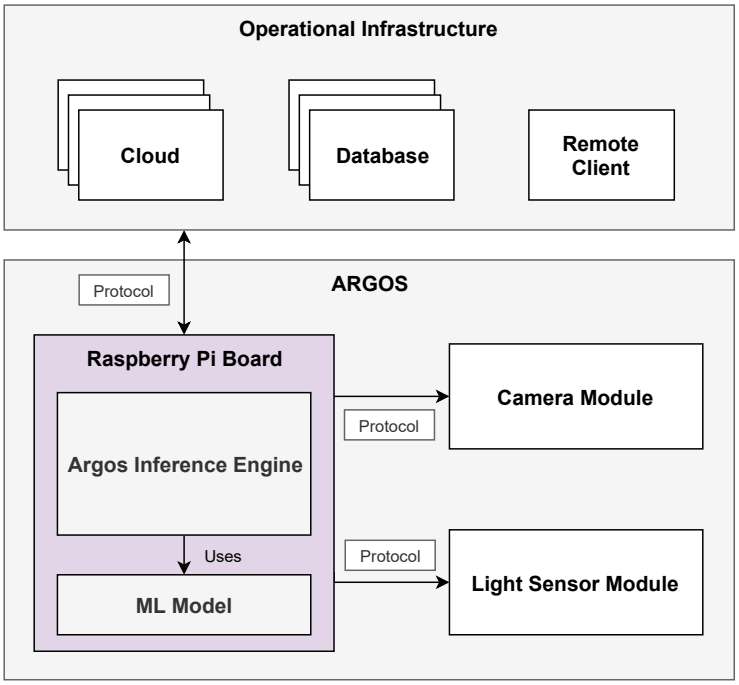


Figure 2.6.1: System Overview Diagram - Computational Unit

2.6.1 Events

In order to understand how the system works, it is important to know what are the most important events that can take place in it and how it will respond to each one of them. Table 2.1 highlights these events from the perspective of the Computational Unit, categorising them by their source and synchronism and linking it to the system's intended response.

Event	System Response	Source	Type
Power on	Configure sensors, listen for a connection from the Remote Client, attempt to log into the database and connect to the cloud service	User	Asynchronous
Connection Request	Verify credentials and respond to the request	Remote Client	Asynchronous
Connection Accepted	Upload buffered frames, if existent	Cloud Server	Asynchronous
Login Accepted	Upload buffered information, if existent	Database Server	Asynchronous
Sampling Period Elapsed	Request sample from sensors	Computational Unit	Synchronous
Camera Frame Available	Analyse the frame and run inference	Camera Driver	Synchronous
Weapon Identified	Save last x frames and start storing information in the cloud and database	ML Inference Engine	Asynchronous

Table 2.1: Computational Unit's Events

2.6.2 Use Cases

The computational unit is the main entity of the system as it holds the machine learning model responsible for threat detection. Additionally, it interfaces with the cloud, a database, and a remote client, as depicted in figure 2.6.2.

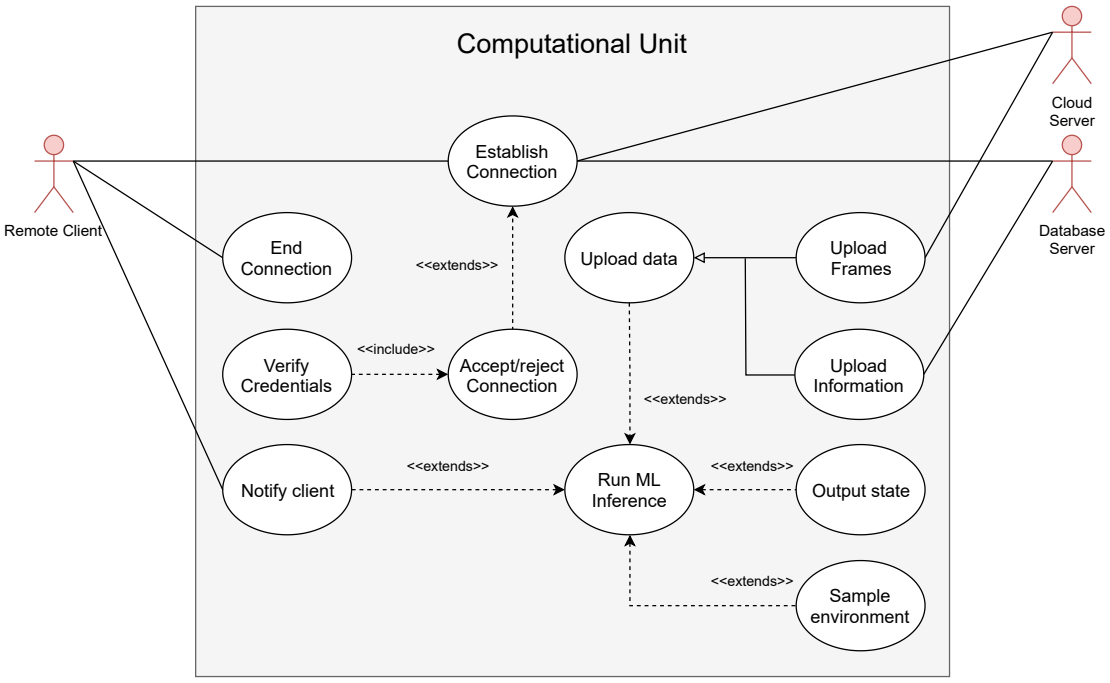


Figure 2.6.2: Local System's Use Case Diagram

2.6.3 State Chart

The two main functional states of the Computational Unit - *Startup* and *Execution* - and their division into more complex state machines corresponding each of its subsystems, are depicted in figure 2.6.3.

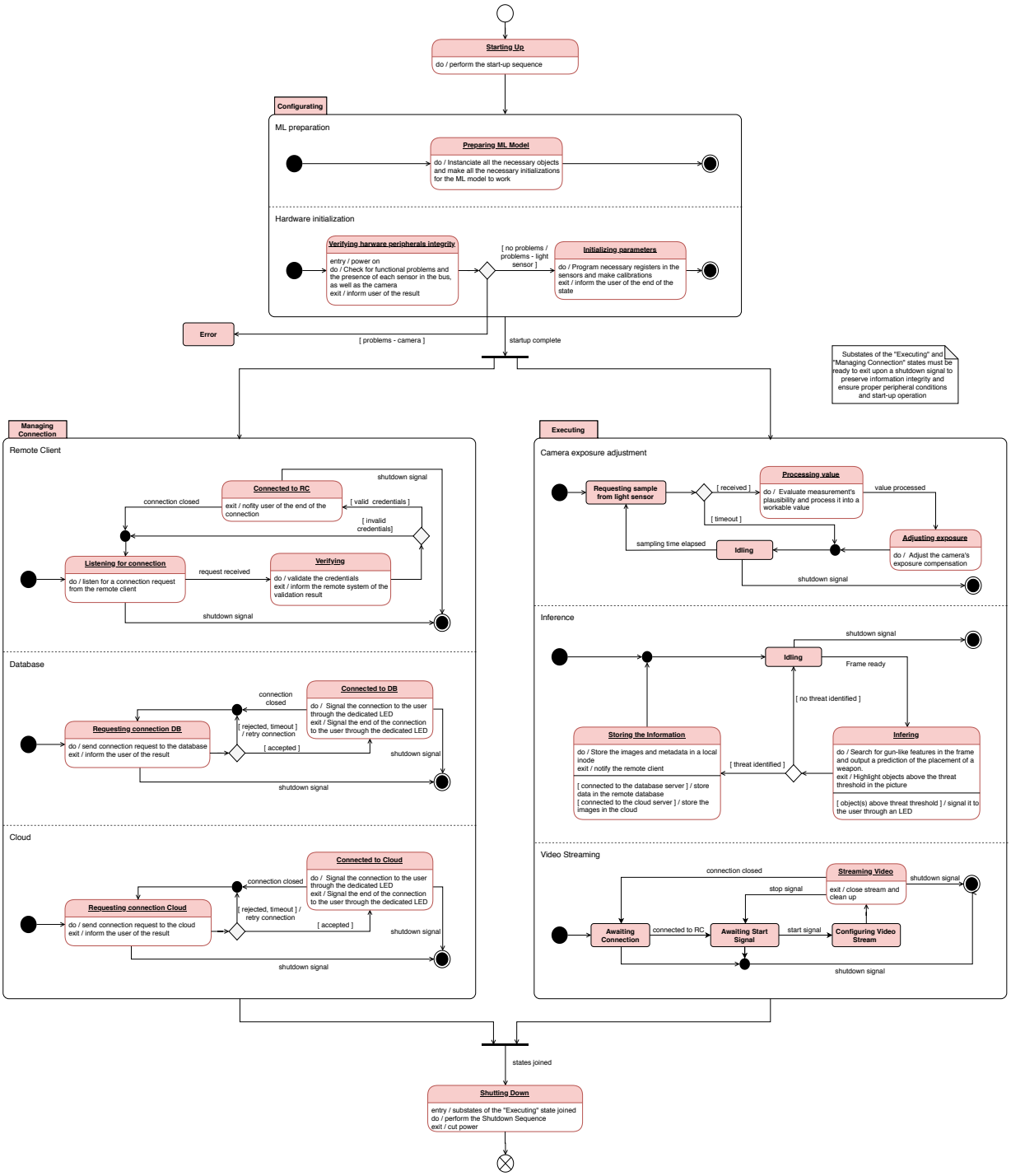


Figure 2.6.3: State Machine Diagram - Comp. Unit (Aug. in Appendix A.2)

2.6.4 Sequence Diagram

The sequence diagrams in figures 2.6.4 to 2.6.6 show the Computational Unit's projected sequences of steps in response to the most important internal and external events.

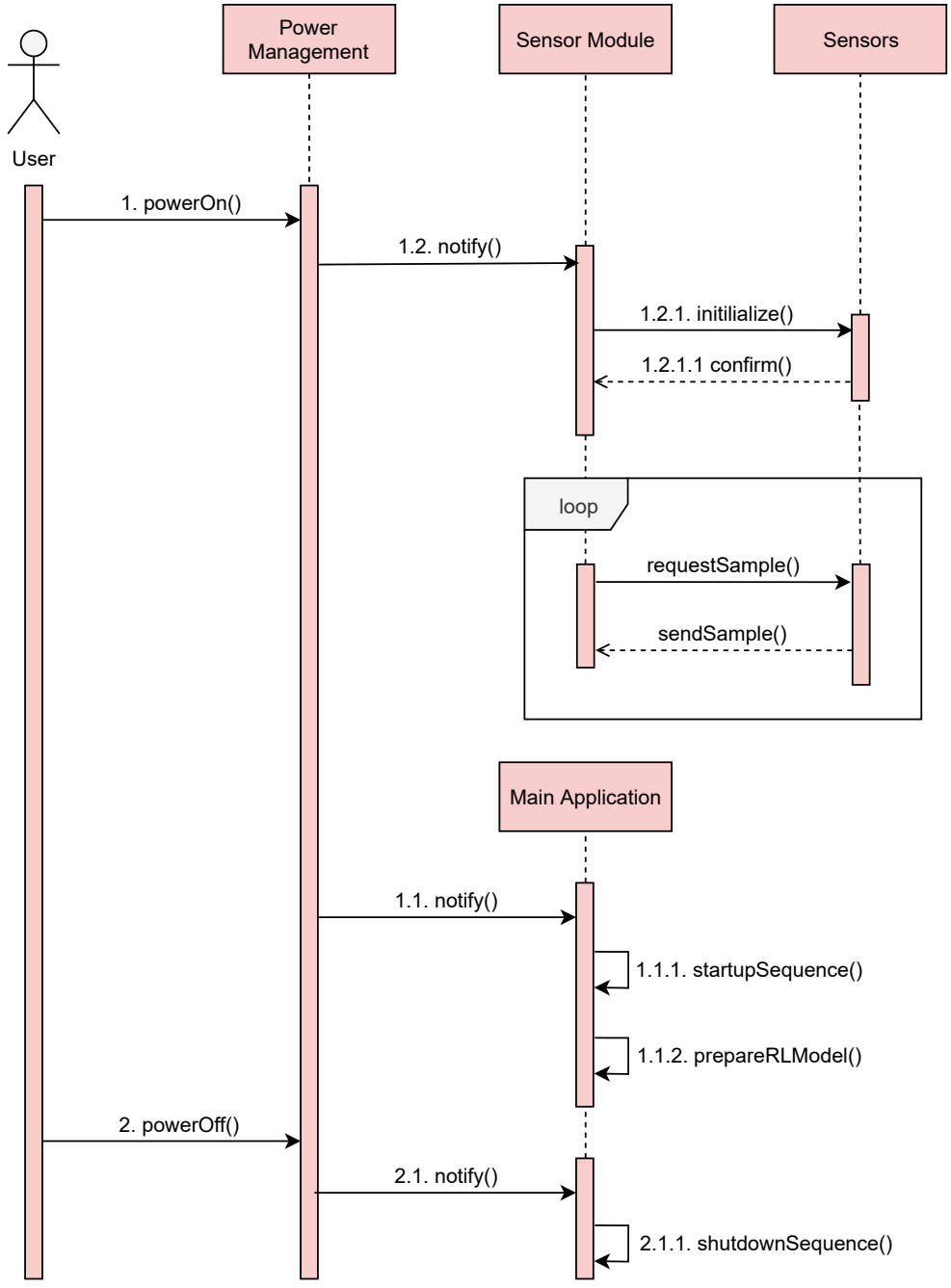


Figure 2.6.4: Computational Unit's Sequence Diagram - Startup

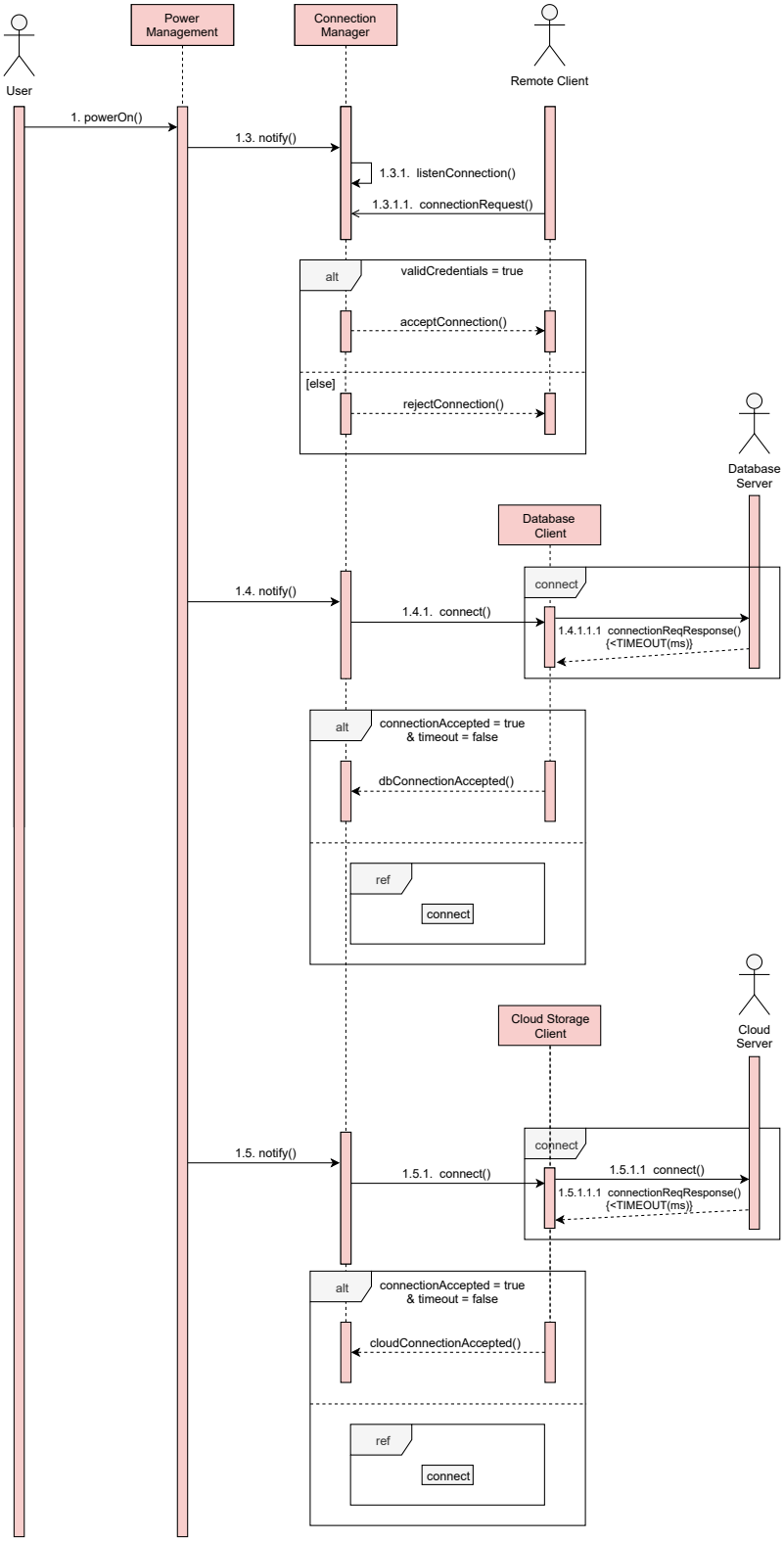


Figure 2.6.5: Computational Unit's Sequence Diagram - Connection Management

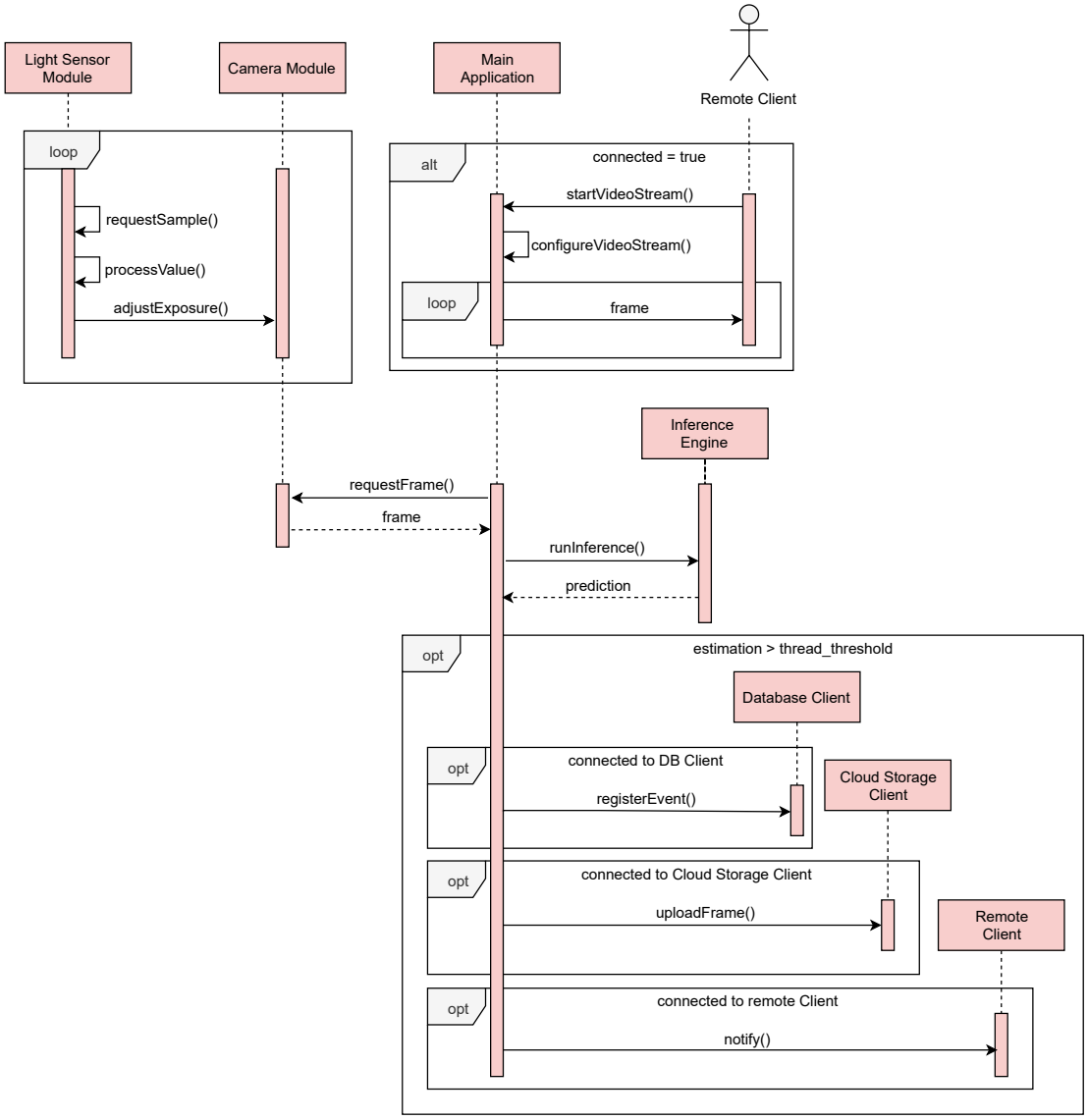


Figure 2.6.6: Computational Unit's Sequence Diagram - Execution

2.7 Machine Learning

The **Machine Learning Module** is highlighted in the system overview of figure 2.7.1.

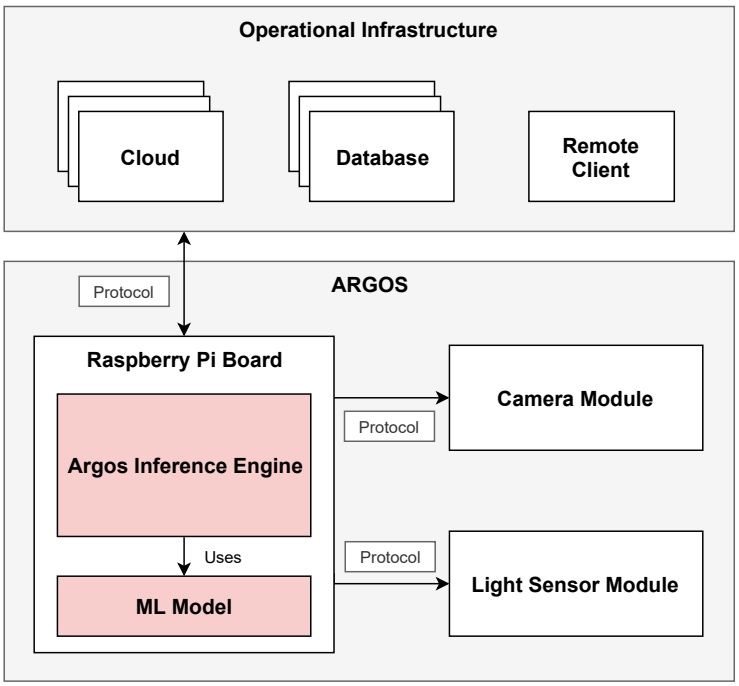


Figure 2.7.1: System Overview Diagram - Machine Learning Model

The process for obtaining a good model in an end-to-end Machine Learning project goes through some key concepts [2]:

1. Look at the Big Picture;
2. Get the Data;
3. Discover and visualise the data to gain insights;
4. Prepare the data for the ML algorithms;
5. Select, train and fine-tune the model;
6. Present a solution;
7. Deploy the model;

The analysis stage will focus on the big picture, the dataset research, and the deployment of the model on the edge device [4].

Analyse the Big Picture

The **Big Picture** of **ARGOS Machine Learning Model** is represented in figure 2.7.2. The model follows a supervised learning approach (classification), and it's expected to learn how to identify handguns.

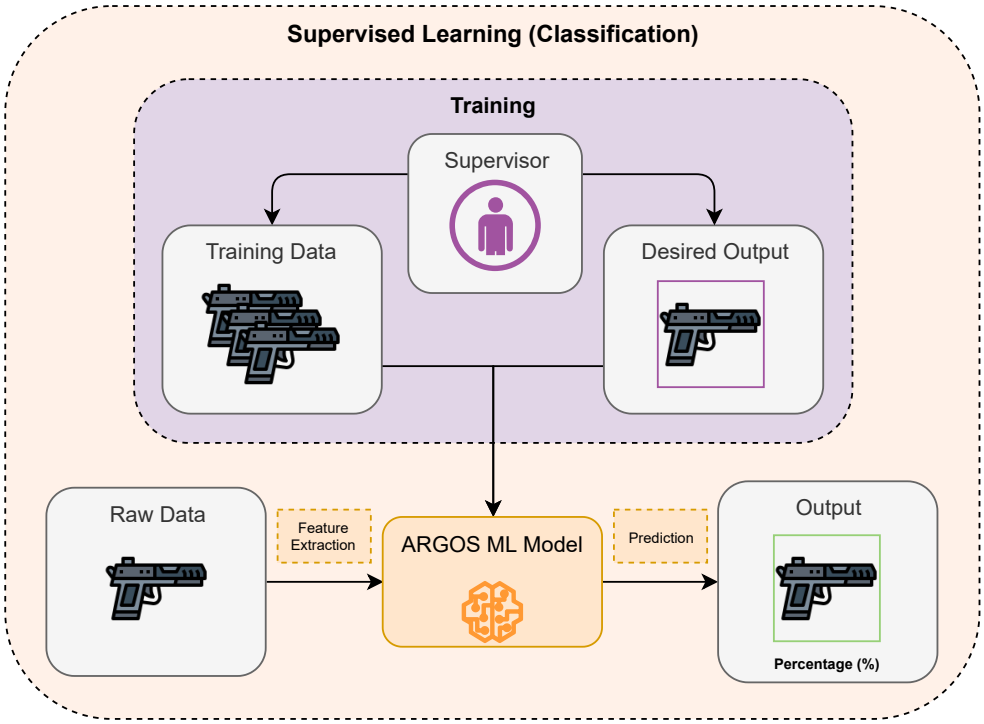


Figure 2.7.2: Machine Learning Big Picture

Dataset

The dataset plays an important part as it is used for training purposes during the learning stage, evaluation of the model in the testing stage, and sometimes in validation, as portrayed in figure 2.7.3. After some research, one found an initial dataset [3] that should suffice the project needs.

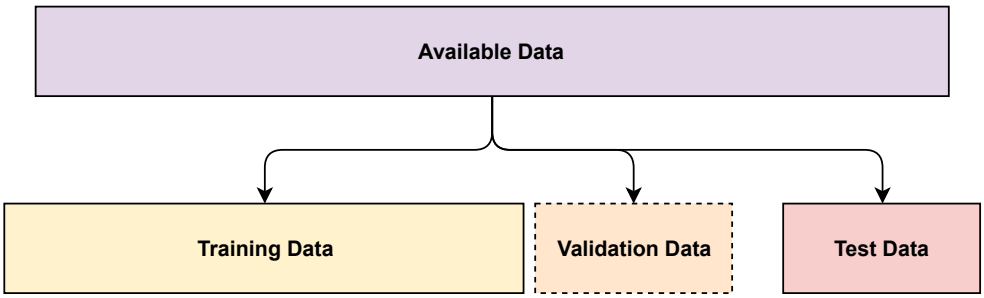


Figure 2.7.3: Dataset Division

Model Conversion & Deployment on Embedded Target

The final step of the mentioned process is the model conversion and deployment on the target device. In this case, the system is a resource-scarce embedded system [4], which demands a more careful deployment considering the compatibility between the model developing tool and the embedded target.

The tool selected for performing the model creation and training was **Tensorflow** since it has been extensively tested with many processors based on the **Arm Cortex-A Series** architecture [6] and has been ported to several other architectures.

The proposed **Edge Machine Learning Framework** [4][7] for ARGOS is presented in figure 2.7.4.

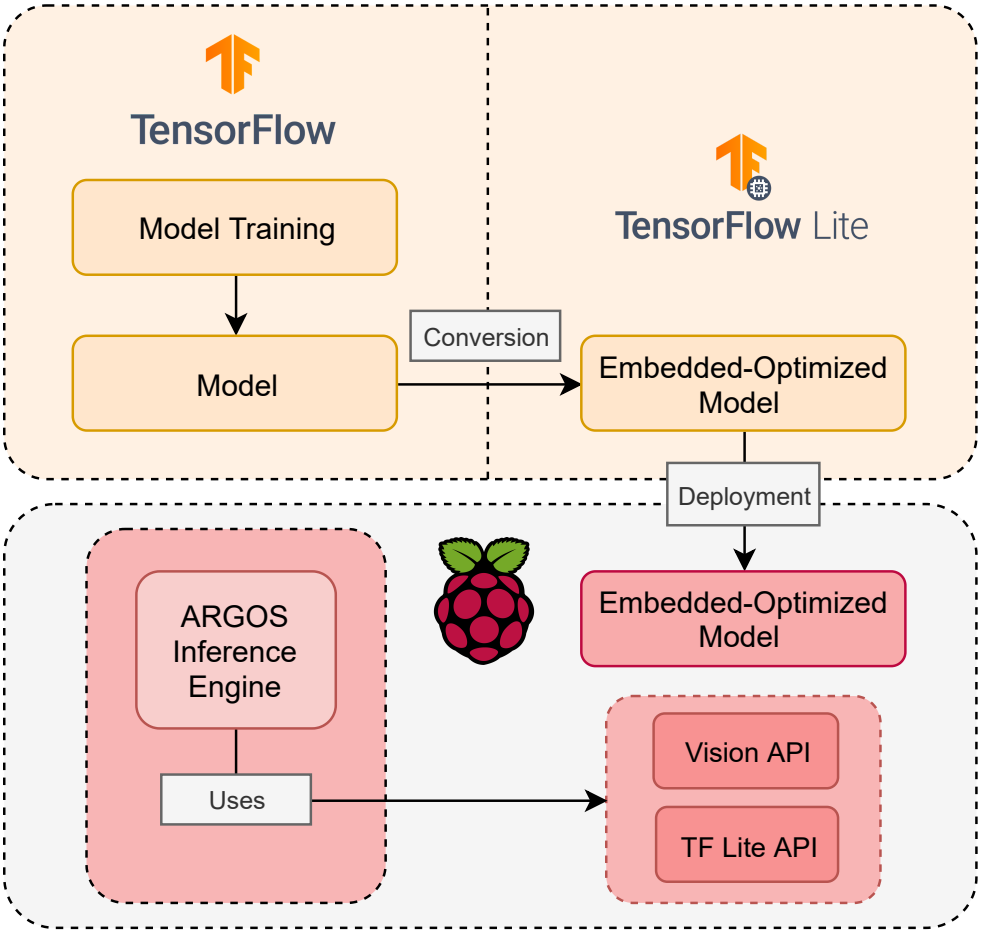


Figure 2.7.4: ARGOS Edge Machine Learning Framework

2.8 Remote Client

The **Remote Client** is highlighted in the system overview of figure 2.8.1.

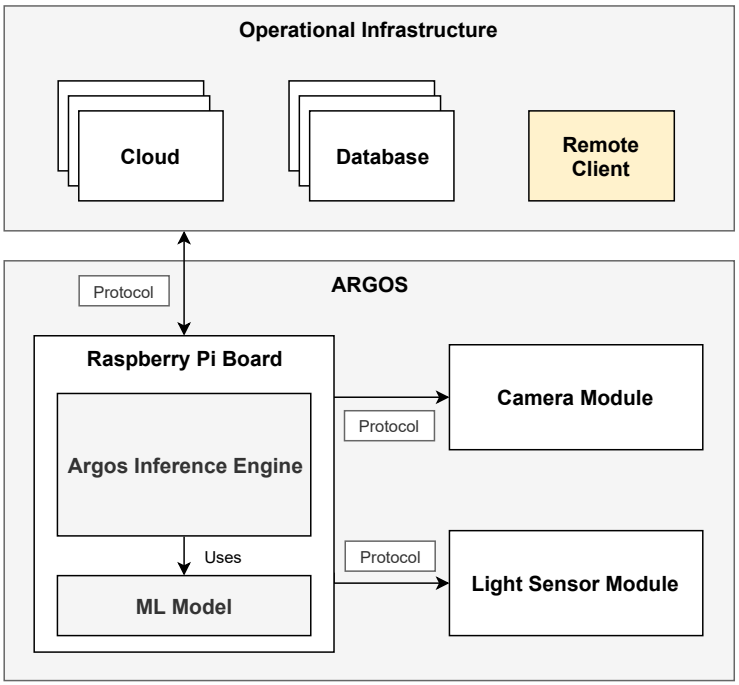


Figure 2.8.1: System Overview Diagram - Remote Client

2.8.1 Events

Table 2.2 presents the relationship between the most relevant events and their respective responses for the Remote Client.

Event	System Response	Source	Type
Power on	Attempt to connect to the Computational Unit, attempt to log into the database and connect to the cloud service	User	Asynchronous
Connection Accepted	Signal to user	Computational Unit	Asynchronous
Connection Accepted	Check for the occurrence of signalled events	Cloud Server	Asynchronous
Login Accepted	Check for the occurrence of signalled events	Database Server	Asynchronous
Verification Period Elapsed	Analyse the database and cloud to identify signalled events	Remote Client	Synchronous
Signalled Event Identified	Notify user and provide the proper interface for viewing footage and other information	Remote Client	Asynchronous

Table 2.2: Remote Client's Events

2.8.2 Use Cases

The remote client is responsible for establishing the connection with the user through a GUI, in which one can access the data previously stored in the cloud and a database, as represented in figure 2.8.2.

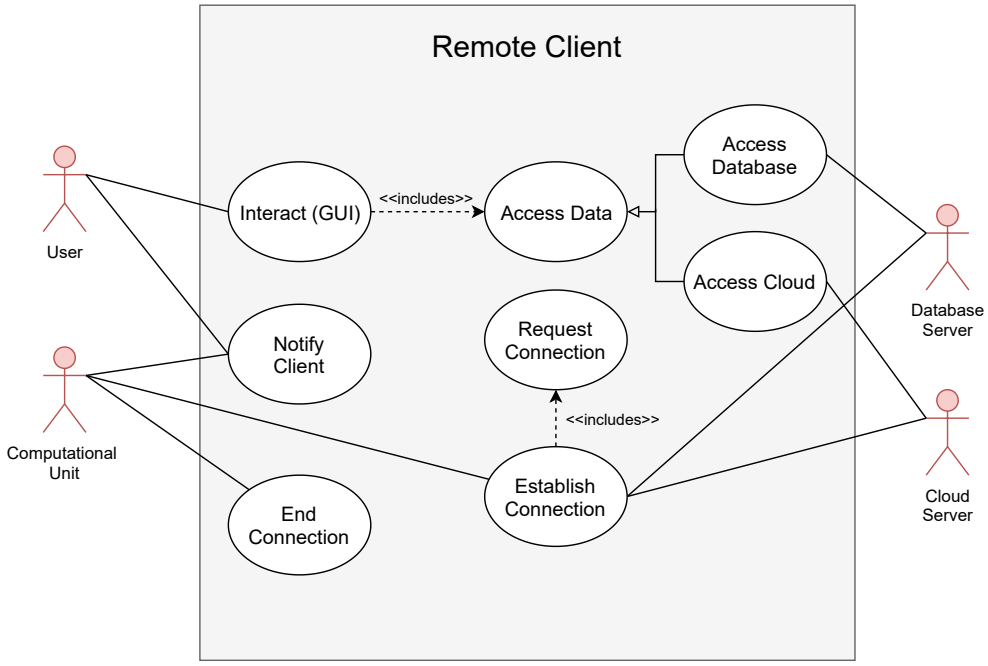


Figure 2.8.2: Remote System's Use Case Diagram

2.8.3 State Chart

The two main functional states of the Remote Client - *Startup* and *Execution* - and their division into more complex state machines corresponding each of its subsystems, are depicted in figure 2.8.3.

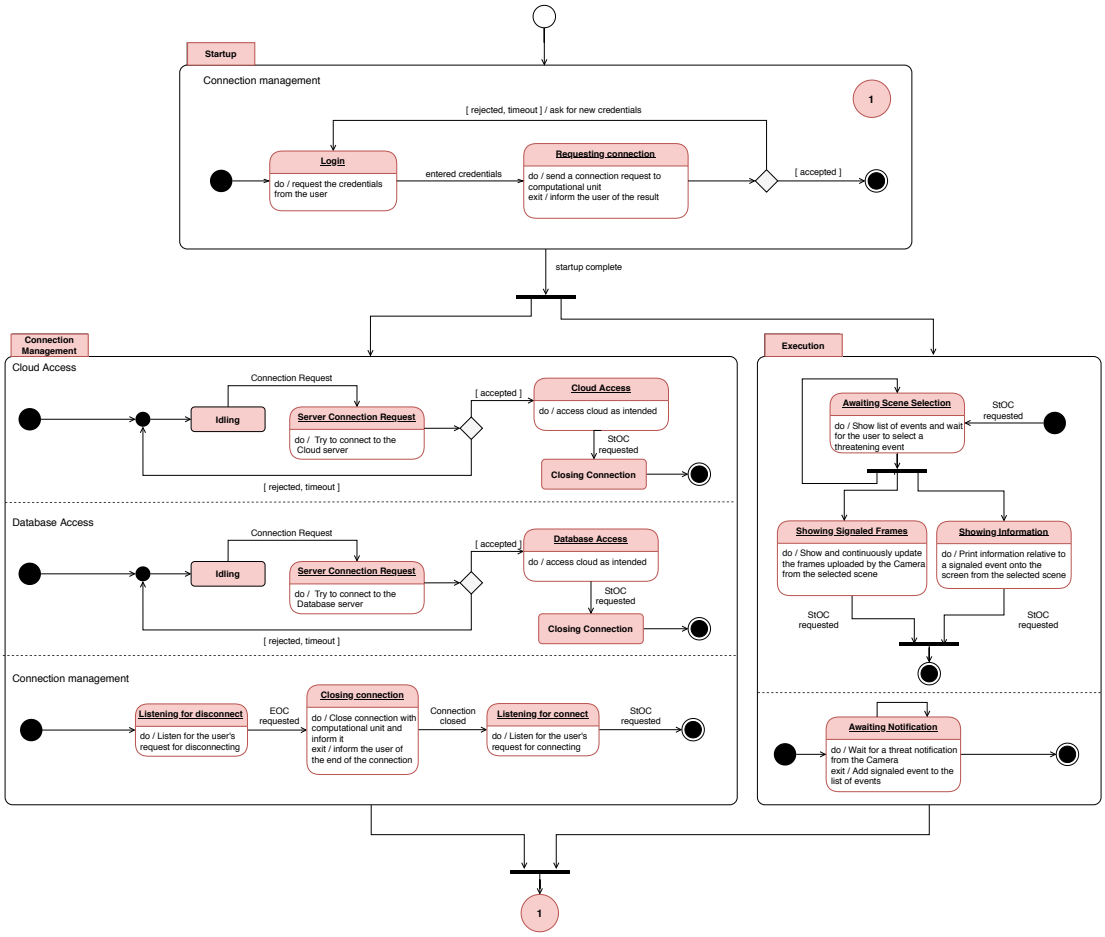


Figure 2.8.3: State Machine Diagram - Remote Client (Aug. in Appendix A.3)

2.8.4 Sequence Diagram

The sequence diagrams in figures 2.8.4 and 2.8.5 show how the Remote Client responds to the user's inputs, bridging the gap between them and the camera and and transforming their requests into commands and data queries.

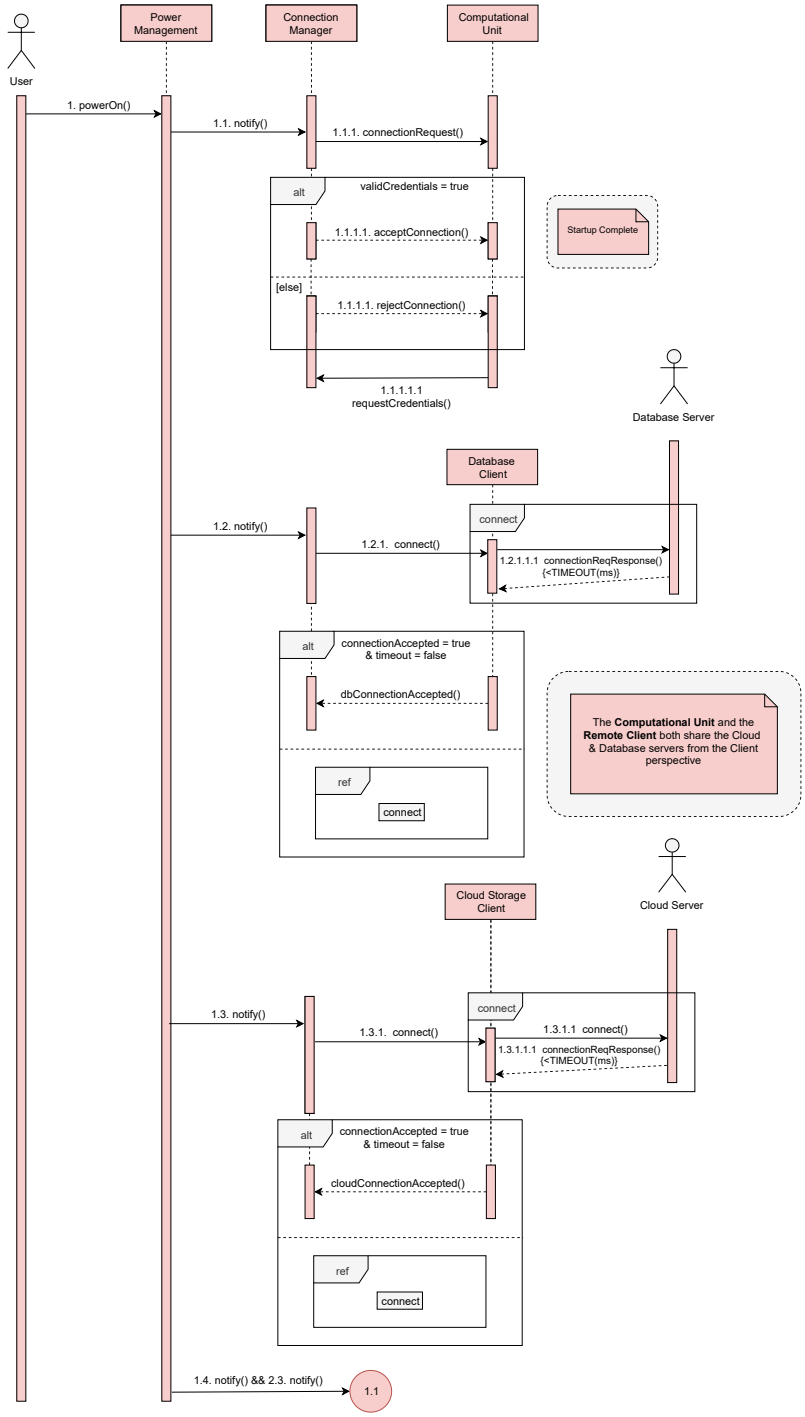


Figure 2.8.4: Remote Client's Sequence Diagram - Startup

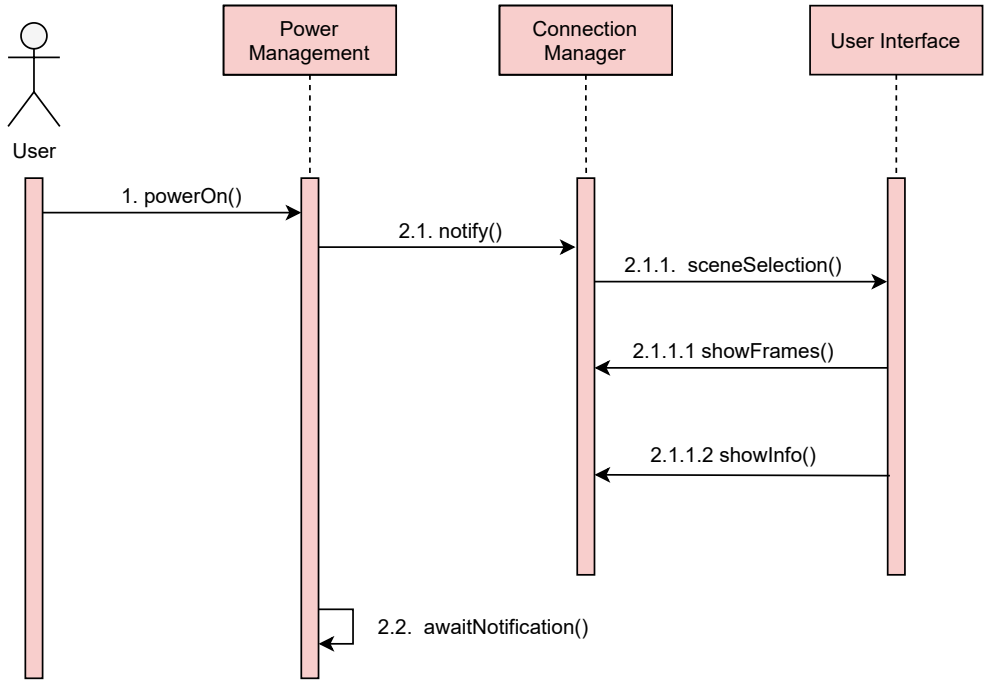


Figure 2.8.5: Remote Client's Sequence Diagram - Startup

2.9 Database

In figure 2.9.1 is represented the **Entity Relationship Diagram** for the **ARGOS database**. The latter intends to store information relative to the entire process of threat detection. Analysing the schematic, one can notice that the user can have one or multiple cameras but a camera is usually owned by only one user, hence the one-to-many relation between these entities. The camera itself outputs multiple frames, yet each frame is produced only by one camera, hence the one-to-many relation. In a frame that was captured the inference engine can detect from zero up to multiple objects/weapons, so the entities are connected by a one-to-many relation. Additionally, each object detection in a single frame can trigger an alert for the user or no alert at all, depending on which the threat threshold was crossed or not, respectively. Moreover, the user that accesses the remote client can receive zero or multiple threat alerts, hence the one-to-many relation between the user and the alert entity. Finally, one can observe that the alert is related to a threat level but it's not impossible to have alerts with the same level of menace, which implies a one-to-many relation.

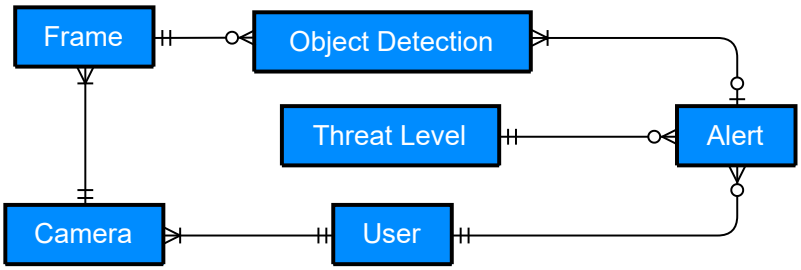


Figure 2.9.1: ARGOS Database Entity Relationship Diagram

2.10 Initial Budget

Table 2.3 represents an initial project budget estimate.

Item	**Cost**
Raspberry Pi Model 4B	42€
Camera Module	15€
Sensor Modules	5€
Power Modules	9€
Structure	10€
Total	81€

Table 2.3: Initial Budget

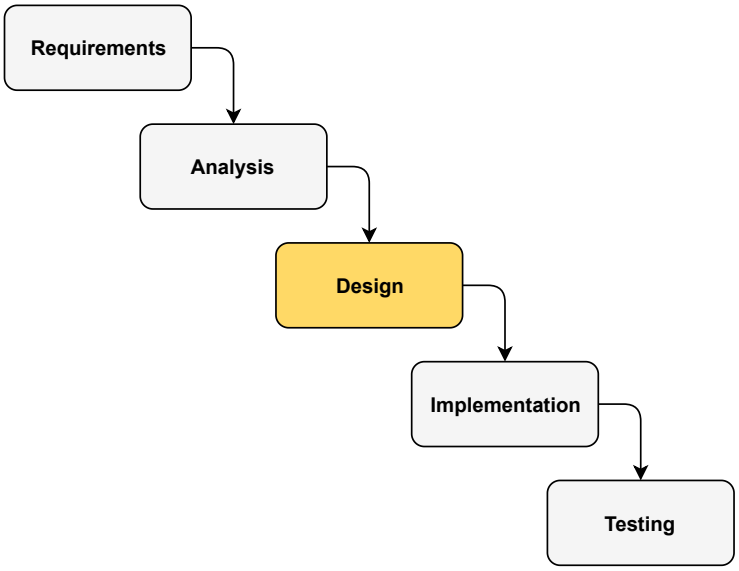
2.11 Gantt Diagram

The Gantt diagram for the project is represented in figure F.1, in the appendix chapter F.

Chapter 3

Design

In this stage, the hardware and software specifications will be approached for each ARGOS subsystem, computational unit, and remote unit. As the design proceeds, one will specify their peripheral interface, main protocols, working sequences, and the tools and COTS leveraged for the design. At the end of this stage, the overall system must be completely clear for stakeholders, and the guidelines for implementation must be explicit.



3.1 Theoretical Foundations

In this section, some main concept background will be provided, namely relating to the communications protocols.

3.1.1 Protocols

In order to establish communication between the peripherals and the Computational Unit, as well as between it and the Remote Client, the pre-existing and standard protocols that are supported by the these devices will be used and their operation will be explained just as deeply as necessary to understand their how they are to be used in the context of this application.

I²C

The Inter-IC Communication (I^2C) bus is a half-duplex bidirectional bus that can connect, in a master-slave hierarchy, one or more master devices to one or more slave devices. Communication is done, logically, using two lines, Serial Data (SDA) and Serial Clock (SCL) and the slaves respond only when interrogated by the master, through their unique address.

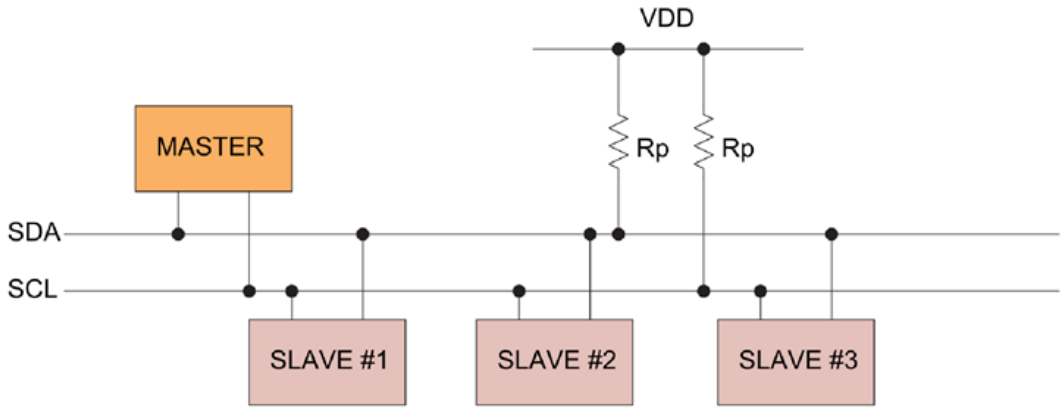


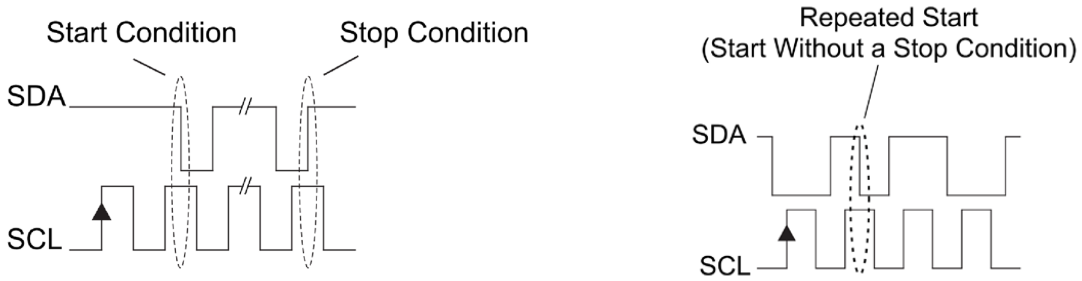
Figure 3.1.1: I^2C connection

In the physical layer, I^2C compatible devices connect to the bus with open collector or open drain pins which pull the line LOW, for writing a LOW bit into the bus, or leave the line HIGH for writing a HIGH bit. Electrically, this is configuration is made possible by pulling the communication lines to the supply voltage using pull up resistors. Bytes are clocked on falling clock edges.

I^2C supports 7, 8 or 10-bit addressing modes, allowing for the connection of up to 1023 devices, being address 0 a general call for all devices in the bus. I2C data packets are arranged in 8-bit bytes comprising slave address, register number, and data to be transferred.

I^2C transmissions always start with a "START Condition", to wake the slaves up from a sleeping state and end with a "STOP Condition", to tell the slaves to resume sleep, as exemplified in figure 3.1.2. These are characterized by

changing the SDA line while SCL is high. When a Start condition is repeated during a transmission without the need for first terminating with a Stop condition, it's known as a "Repeated Start" and it can used to, for example, change data transmission direction or repeat transmission attempts.



(a) Start Condition And Stop Condition

(b) Repeated Start Condition

Figure 3.1.2: I^2C transition delimiter conditions

Data bits encode the actual transmission data and are transmitted in 8-bit byte format, starting with the MSB, and each bit is synchronized with SCL. Each byte must be followed by an Acknowledge (ACK) bit which is generated by the recipient of the data.

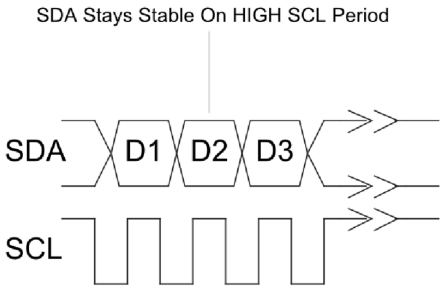


Figure 3.1.3: I^2C Bit Transition of Data Bits

When writing to or reading from a specific register in a SLAVE, the master must first point to the specific register by writing the register address once the SLAVE has been addressed. While the register address can be considered a data byte, to avoid confusion it is often classified as a Command byte.

S	ADDRESS	$\overline{W}$	A	COMMAND	A	S	ADDRESS	R	A	DATA	$\overline{A}$	P
	1 1 0 a3:a0	0	0	X X b5:b0	0		1 1 0 a3:a0	1	0	b7:b0	1	

Figure 3.1.4: I^2C Example Byte Read Transaction

MIPI Camera Serial Interface (CSI)

CSI is a camera interface protocol originally developed by MIPI Alliance that connects image sensors to host processors. It arose with the need for image and display interface solutions that met stringent power and performance requirements [11]. The most common CSI standard is CSI-2. It consists of a unique physical bus (D-PHY) that contains a differential clock and from one to four differential data lanes. A CSI-2 interface between an image sensor and an Active-Pixel Sensor (APS) [12] is illustrated in figure 3.1.5.

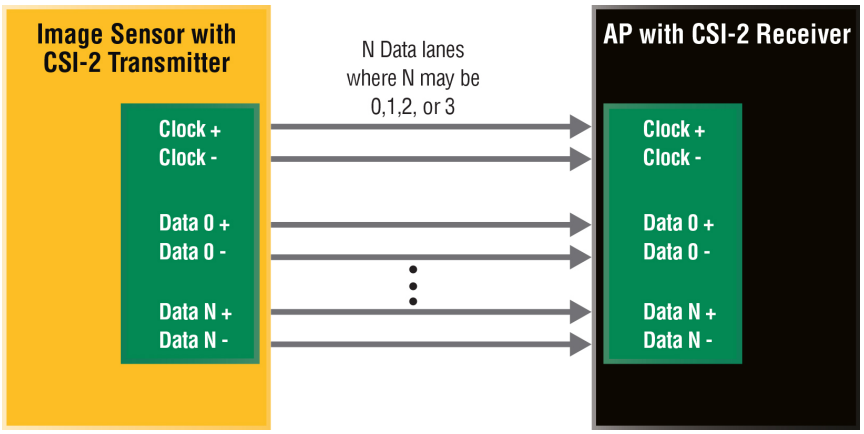


Figure 3.1.5: CSI-2 Sensor Interface Example

3.2 Computational Unit: HW Specification

This section intends to lay out all the necessary hardware for the ARGOS project and clearly define the characteristics which promoted each choice made relative to other viable choices.

3.2.1 Development Board

This project includes a Raspberry Pi 4 Model B (3.2.1) considering it was a technical constraint identified in the analysis phase (section 2.3.2). The board includes a Broadcom BCM2711 MPU and a range of computer-like features that suit several applications.

For the project at stake, the most important features are the following:

- 2GB LPDDR4-3200 SDRAM;
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE;
- Raspberry Pi standard 40 pin GPIO header;
- 2-lane MIPI CSI camera port;
- Micro-SD card slot for loading operating system and data storage;

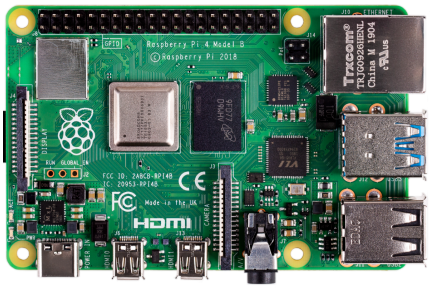


Figure 3.2.1: Raspberry Pi 4 Model B

The Raspberry Pi Model 3 B/B+ would also be a good fit for the project, as it's cheaper than the model 4B without jeopardizing a lot of performance and yet providing the necessary peripherals. Model 4B was chosen simply because it was already in stock for other purposes (time-to-market reduction), but also because its price isn't that far from its prior counterpart. In other words, the project budget could be decreased with the use of an earlier model at the cost of processing power and an increase in power consumption [8].

General-Purpose Input/Output (GPIO)

The interface with external peripherals is possible due to the 40 pin GPIO header the board provides. The latter is illustrated in figure 3.2.2 [9] with the pins' functionalities.

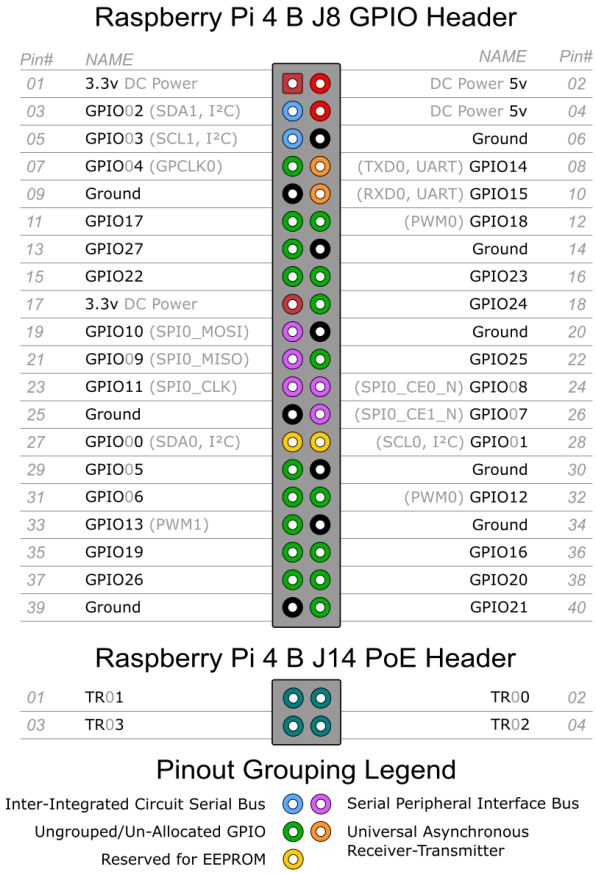


Figure 3.2.2: Raspberry Pi 4 GPIO

3.2.2 Camera Module

In order to acquire the frames that ARGOS' inference engine needs to run inference, a camera (figure 3.2.3) must be added to the project. Considering this, the computational unit will a include a Raspberry Pi Camera Module V1, capable of delivering a clear 5MP resolution image, or 1080p high definition video recording at 30fps.

One opted for a simple yet effective camera model that serves as the bedrock for the ARGOS vision subsystem, whilst reducing project cost.

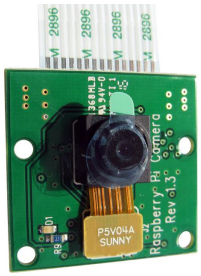


Figure 3.2.3: Raspberry Pi 4 Camera Module

3.2.3 Light Sensor

To later adjust the onboard camera settings based on the current ambient light, a TSL2581 light-to-digital converter (3.2.4) will be used to attain values that can be converted into the desired illuminance (ambient light level) in lux.

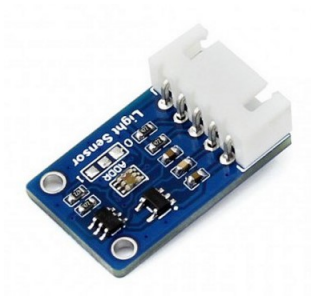


Figure 3.2.4: TSL2581 Light Sensor

Some features that justify the selection made are:

- I^2C interface directly outputs the ambient light intensity value without calibration;
- Low Quiescent Current 3μA in Power Down Mode;
- Approximates Human Eye Response;
- Embedded photodiode with infrared filter, gets high a precision output even in environments with strong infrared noise interference;
- Embedded voltage level translation circuit, allows directly connecting to 3.3V/5V MCU systems;

3.2.4 Power Supply

The project's power supply will be the standard one that usually powers up the model 4B. Its representaion is in figure 3.2.5.

Figure 3.2.5: Raspberry Pi 4B Power Supply

3.2.5 Power Switch

In order to turn the Computational Unit's power on and off, ARGOS implements a push button power switch. This is an alternative to the typical toggle switch and it's meant to give the product a modern appeal, as well as delegate as much power as possible to the processor when turning off, giving it time to terminate tasks and close previously open connections. Another benefit of this is allowing the product to turn itself down. For this, the push button solution needs to have as low $I_{Quiescent}$ as possible.

Although there are a few integrated solutions on the market, none quite satisfy the specific functionality, space, energy and cost-constrained needs of the project, and as such a custom solution is needed. This is indeed the route that was taken, as a simple, cost-efficient and compact solution was found that solved this problem in just the right way.

This solution is based on the Maxim MAX16150 Pushbutton On/Off Controller and Battery Freshness Seal IC and follows the recommendation in the "Typical Application Circuit" section of the product's own datasheet [23], found in figure 3.2.6.

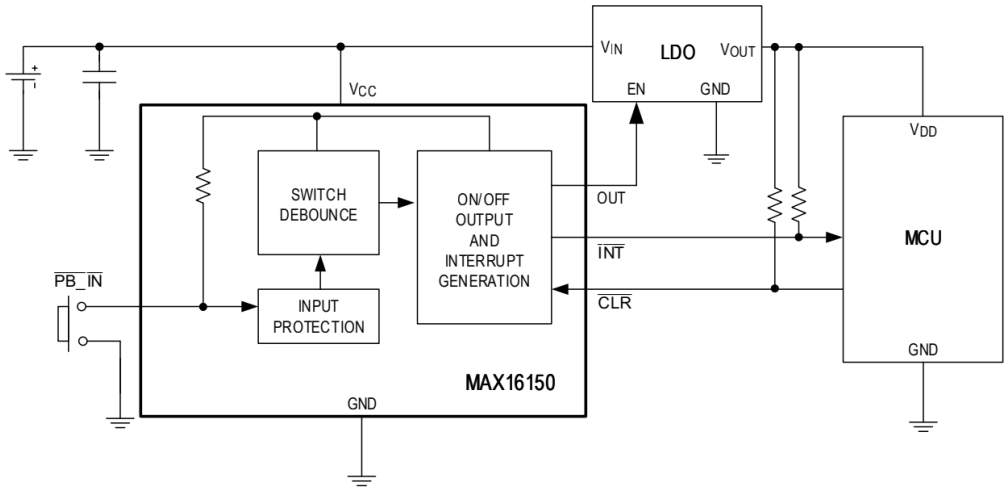


Figure 3.2.6: Typical Application Circuit for MAX16150

3.3 Computational Unit: Peripheral Interface

In this segment, the connections between the hardware components will be considered, mainly addressing the peripheral connections with the development board.

Light Sensor

The interface with this peripheral is pretty straight forward since one only has to connect the peripheral to the board's V_{DD}, GND, SDA, and SCL. The INT pin is used in case of interrupt output (optional) (3.3.1) [19].

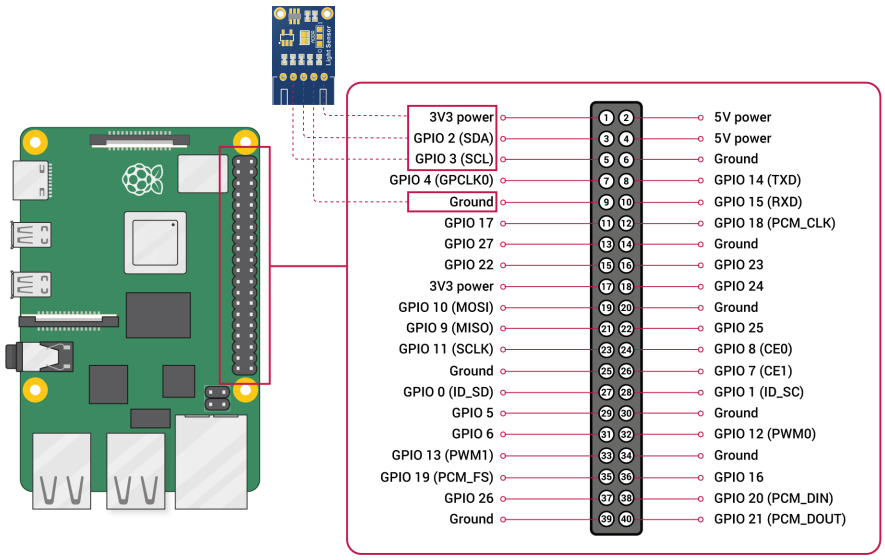


Figure 3.3.1: Light Sensor Interface

The general application circuit is depicted in figure 3.3.2. Firstly, the power supply lines must be decoupled with a $0.1\mu F$ capacitor placed as close to the device package as possible. The bypass capacitor must be of the ceramic type, so it provides a low impedance path to the ground at higher frequencies, handling the transient currents generated by the sensor's internal logic switching. Additionally, some pull-up resistors must be added to ensure the SDA and SCL lines are at a high level when the bus is available.

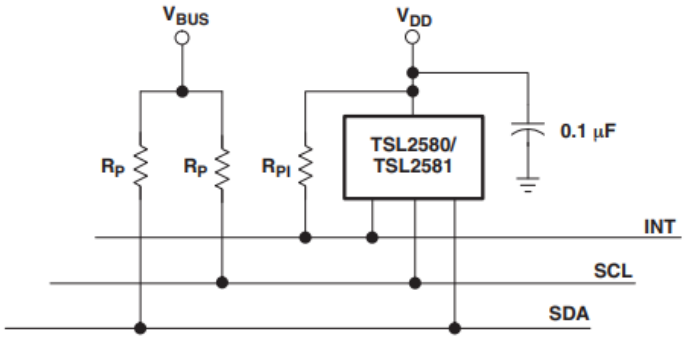


Figure 3.3.2: Power Supply Decoupling and Application Hardware Circuit

The value for R_p resistors can be calculated based on the table presented in in figure 3.3.3 [21]. The V_{OLMAX} value can be seen in figure 3.3.4 [20].

A pull-up resistor (R_{PI}) is also required for the interrupt (INT), which functions as a wired-AND signal similarly to the SCL and SDA lines. For R_{PI} an impedance value between $10k\Omega$ and $100k\Omega$ can be used [19].

1) $R_{MIN} < R_{PU} < R_{MAX}$

2) $R_{MIN} = (V_{DDMAX} - V_{OLMAX}) / I_{OLMAX}$

V_{DDMAX}	V_{OLMAX}	R_{MIN}			
		I_{OLMAX} = 3mA	I_{OLMAX} = 6mA*	I_{OLMAX} = 12mA**	I_{OLMAX} = 30mA***
2.7 V	0.6 V	**700 Ω**	**350 Ω**	**175 Ω**	**70 Ω**
3.6 V	0.6 V	**1.0 kΩ**	**500 Ω**	**250 Ω**	**100 Ω**

* I²C Bus with a buffer
** EXPxxxx bus
*** I²C Fast-mode Plus Bus

3) $R_{MAX} * C_{MAX} = 1.18 * t_r$

MODE	Frequency	t_r	C_{MAX}	R_{MAX}
Standard	100 kHz	1000 ns	400 pF	**2.96 kΩ**
Fast Mode	400 kHz	300 ns	400 pF	**885 Ω**
Fast Mode Plus	1000 kHz	120 ns	560 pF	**252 Ω**

Figure 3.3.3: Pull-up Resistor Calculation

Symbol	Parameter	Limits		Units	Comments
		Min	Max		
V_{DD}	Nominal bus voltage	1.8	5.0	V	
	Operating bus voltage	1.62	5.5	V	Nominal ±10%
V_{IH}	HIGH level input voltage for SMBCLK and SMBDAT	1.35	V_{DD}	V	
V_{IL}	LOW level input voltage for SMBCLK and SMBDAT	–	0.8	V	
V_{OL}	Low level output voltage for SMBCLK and SMBDAT	–	0.4	V	I_{OL} = -350 µA
I_{LEAK_PIN}	Input leakage current per device pin	–5	5	µA	Note 1
I_{PULLUP}	Current through pull-up resistor or from pull-up current source	100	350	µA	Note 2

Figure 3.3.4: Low Power SMBus DC Specification

Since this was already implemented inside the integrated sensor module (3.3.5) [22], one doesn't need to worry about those details.

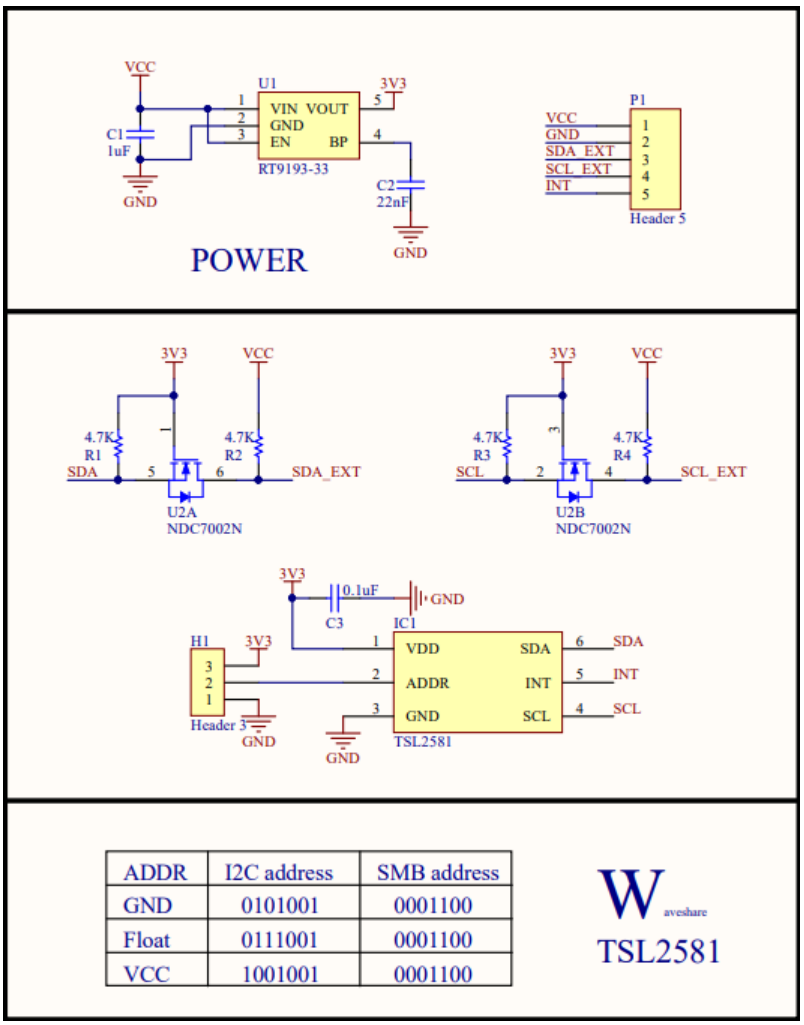


Figure 3.3.5: Light Sensor Schematic

Camera Module

The camera module will interface with the Raspberry Pi 4B through the 2-lane MIPI CSI port (3.3.6).

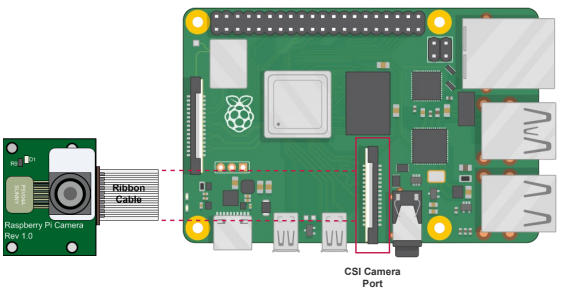


Figure 3.3.6: Camera Interface

LEDs

The project will have three LEDs to display the current state of the device:

- A **green/red** one to signal power/fault;
- A **white** one to signal the connection with the remote client;
- A **blue** one to signal the Wi-Fi connection;

The processing state of any of the above will be signalled in a blinking manner.

3.4 Tools and COTS

The software tools and COTS used for the project are described in this section.

3.4.1 Tools Summary

- **Buildroot** - Tool to configure and generate the Raspberry Pi Kernel image;
- **QtCreator** - Cross-platform IDE used for the GUI development;
- **C++** - Programming language chosen for the Computational Unit;
- **Python** - Programming language chosen for the Remote Client GUI and the ML model training stage;
- **MySQL** - Relational Database Management System (RDBMS) used for the ARGOS database;
- **Visual Studio Code** - Open-source editor used for developing the system's software;
- **TensorFlow Lite** - Tensorflow for microcontrollers to deploy the model onto the computational unit;
- **ThinkerCAD** - CAD tool used for the initial camera 3D model design;
- **Autodesk Fusion 360** - CAD tool used for designing the product's technical drawing;
- **Anaconda** - Tool utilised for creating virtual environments;

3.4.2 COTS Summary

- **PThreads (POSIX Threads) API** - Used for thread creation and management;
- **Qt API** - Used for the Graphical User Interface;
- **OpenCV API** - Used for image capture and processing;
- **TensorFlow** - Machine Learning API used for designing ARGOS Object Detection Classifier (ODC);

- **CUDA and cuDNN** - Nvidia API used for running Tensorflow on a Graphics Processing Unit (GPU);
- **Cloudant API** - IBM Cloud service that executes transactions in databases;
- **Cloud Storage API** - Google Cloud service that allows for immutable data storage and retrieval;
- **Light Sensor (TSL258x) Device Driver [13]** - open-source device driver used for interfacing with the light sensor;

Most of the **Software Tools and COTS** used for the project are ilustrated in figure 3.4.1.

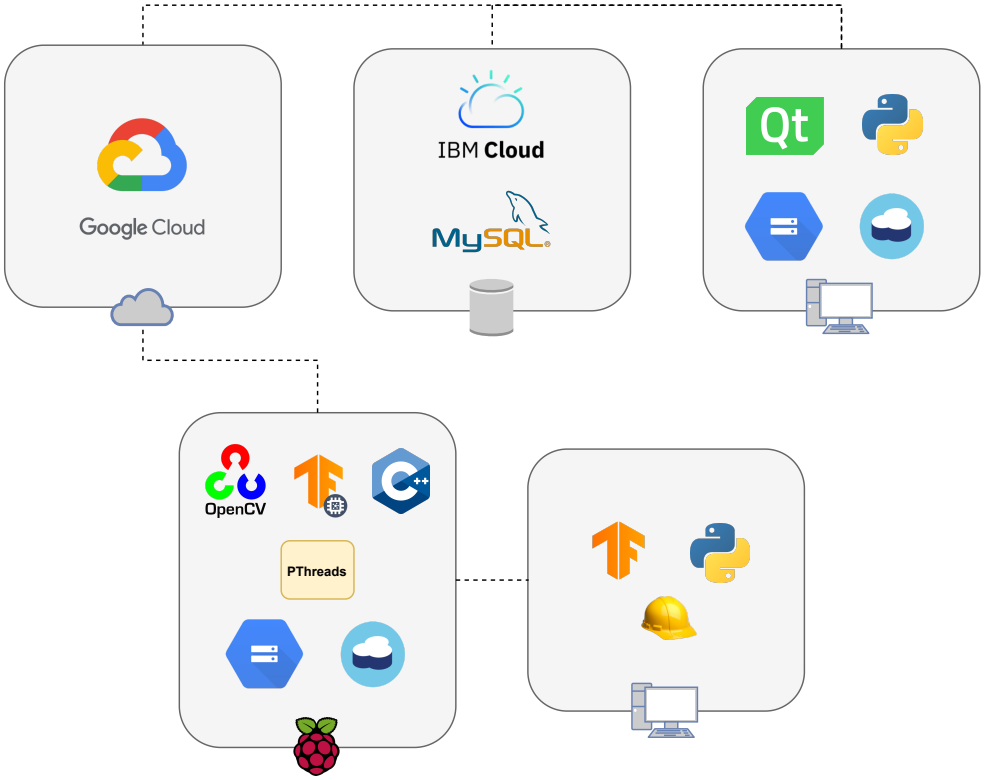


Figure 3.4.1: Software Tools and COTS Overview

3.5 Computational Unit: SW Specification

This section aims to describe all the software relating to the ARGOS' Computational Unit and explicitly define the intended workflow of the system unit.

3.5.1 Device Drivers

Light Sensor

In this section, the software for the light sensor module will be specified. For the matter, one will define the basic operation of the sensor and the device driver used.

Basic operation

The minimum configuration that is necessary to power on TSL2581 and have it sampling is shown in the schematic in figure 3.5.1 taken from the product's own datasheet:

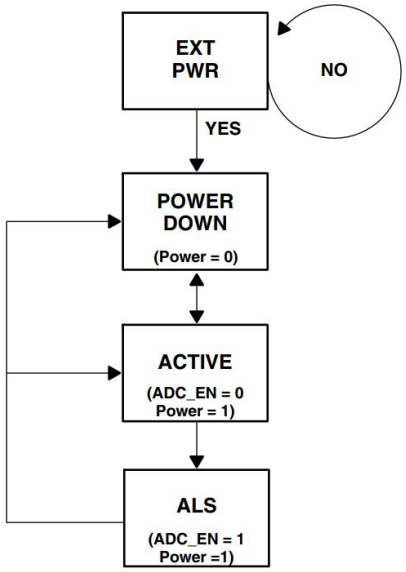


Figure 3.5.1: TSL2581 Basic Operation Diagram

After applying external power, the device will initially be in the power-down state. To operate the device, the first step is to issue a power-up command. After that, a command should be issued to enable the ADCs and start sampling. From then on, all that is needed is to periodically poll the sensor.

Registers

For this application, the most relevant registers in TSL2581 are:

The CONTROL register, which includes:

1. 0 POWER bit, for making the device into sleep mode and waking it up fom sleep;
2. 1 ADC_EN bit, for activating and deactivating the complementary ADCs.

Bit:	7	6	5	4	3	2	1	0	
Address 00h	Reserved		ADC_INTR	ADC_VALID	Reserved		ADC_EN	POWER	Reset 00h

FIELD	BIT	DESCRIPTION
Reserved	7:6	Reserved. Write as 0.
ADC_INTR	5	ADC Interrupt. Read only. Indicates that the device is asserting an interrupt.
ADC_VALID	4	ADC Valid. Read only. Indicates that the ADC channel has completed an integration cycle.
Reserved	3:2	Reserved. Write as 0.
ADC_EN	1	ADC Enable. This field enables the two ADC channels to begin integration. Writing a 1 activates the ADC channels, and writing a 0 disables the ADCs.
POWER	0	Power On. Writing a 1 powers on the device, and writing a 0 turns it off.

Figure 3.5.2: TSL2581 Registers: CONTROL

The TIMING register, for indicating the intended sampling period for the ADCs.

Bit:	7	6	5	4	3	2	1	0	
Address 01h	ATIME								Reset 00h

FIELD	BIT	DESCRIPTION		
ATIME	7:0	Integration Cycles. Specifies the integration time in 2.7-ms intervals. Time is expressed as a 2's complement number. So, to quickly work out the correct value to write: (1) determine the number of 2.7-ms intervals required, and (2) then take the 2's complement. For example, for a 1 × 2.7-ms interval, 0xFF should be written. For 2 × 2.7-ms intervals, 0xFE should be written. The maximum integration time is 688.5 ms (00000001b). Writing a 0x00 to this register is a special case and indicates manual timing mode. See CONTROL and MANUAL INTEGRATION TIMER Registers for other device options related to manual integration.		
		INTEG_CYCLES	TIME	VALUE
		–	Manual integration	00000000
		1	2.7 ms	11111111
		2	5.4 ms	11111110
		19	51.3 ms	11101101
		37	99.9 ms	11011011
		74	199.8 ms	10110110
		148	399.6 ms	01101100
		255	688.5 ms	00000001

Figure 3.5.3: TSL2581 Registers: TIMING

The ANALOG register, for selecting the intended ADC.

Bit:	7	6	5	4	3	2	1	0	
Address 07h	RESV					GAIN			Reset 00h

FIELD	BITS	DESCRIPTION	
Resv	7:3	Reserved. Write as 0.	
Gain	2:0	Gain Control. Sets the analog gain of the device according to the following table.	
		FIELD VALUE	GAIN VALUE
		x00	1×
		x01	8×
		x10	16×
		x11	111×

Figure 3.5.4: TSL2581 Registers: ANALOG

The CHANNEL DATA registers, for reading the sampled ADC register values.

REGISTER	ADDRESS	BITS	DESCRIPTION
DATA0LOW	14h	7:0	ADC channel 0 lower byte
DATA0HIGH	15h	7:0	ADC channel 0 upper byte
DATA1LOW	16h	7:0	ADC channel 1 lower byte
DATA1HIGH	17h	7:0	ADC channel 1 upper byte

Figure 3.5.5: TSL2581 Registers: CHANNEL DATA

Module

In order to guarantee good product reliability and shorten time-to-market, the module that will be used to interact with the light sensor is the built-in driver for the TSL258x family of devices [13], published by the manufacturer.

3.5.2 Camera

For configuring and interfacing with the Raspberry Pi Camera module, the application needs a fitting abstraction layer that can ensure a reliable interface with the camera, respecting the layered architecture of the system.

In order to ensure the best operation feasible and speed up development, the best fit for the task should be OpenCV, as it will already be in use. OpenCV provides a very simple interface for requesting a video stream, taking video samples and making configurations required by the project, such as frame width and height, and image exposure.

For this project, a couple of functionalities from the OpenCV library will be of interest in this case, all from the `VideoCapture` class:

- `open(filename, apiPreference)`, for opening a capturing device. `filename` will have the value `/dev/videoX`, `X` being the minor number of the video capture device;
- `set (propId, value)`, for setting the `value` of various `propId` properties;
- `read(image)`, for capturing an image for later treatment and analysis.
- `release()`, for safely closing the capturing device file.

3.5.3 LEDs

The control of the LEDs, according to the needs of the project will require a device driver with additional writing capabilities, such as defining the intended duty-cycle. With that in mind, a generic definition of the write function will be as follows:

- `ledWrite(file, buffer, length, duty_cycle)`

3.5.4 Machine Learning

The ARGOS Object Detection Classifier (ODC) development will follow the guidelines presented in the flowchart of figure 3.5.6.

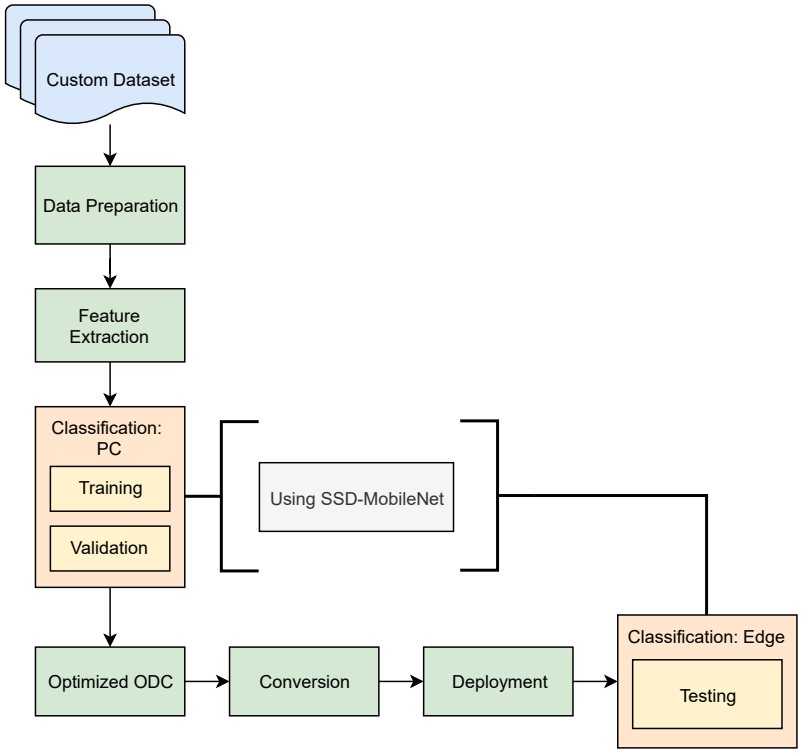


Figure 3.5.6: Machine Learning Flowchart

Beginning with a custom dataset, one will have to ensure that the images don't take up too much space or have too much resolution so that the training phase can be done as quickly as possible. Additionally, note that one shouldn't trade-off dataset length (number of images) for resolution, meaning that it's best to have a bigger image pool with a well-rounded resolution/size ratio than fewer images of higher resolution.

The intent is to utilize a pre-trained ML model (with COCO dataset [16] in this case) and train it to detect a custom set of objects it has never seen [17][18]. There were some options between different Tensorflow models (3.5.7), all varying in terms of speed and accuracy, but the ones that stood out were from the SSD-MobileNet line since they had fast inference times without a significant loss in accuracy, considering they were made to run on edge devices. Of course, these characteristics can only be achieved with longer training periods.

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_mobilenet_v1_0.75_depth_coco ☆	26	18	Boxes
ssd_mobilenet_v1_quantized_coco ☆	29	18	Boxes
ssd_mobilenet_v1_0.75_depth_quantized_coco ☆	29	16	Boxes
ssd_mobilenet_v1_ppn_coco ☆	26	20	Boxes
ssd_mobilenet_v1_fpn_coco ☆	56	32	Boxes
ssd_resnet_50_fpn_coco ☆	76	35	Boxes
ssd_mobilenet_v2_coco	31	22	Boxes
ssd_mobilenet_v2_quantized_coco	29	22	Boxes
ssdlite_mobilenet_v2_coco	27	22	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes
faster_rcnn_nas_lowproposals_coco	540		Boxes
mask_rcnn_inception_resnet_v2_atrous_coco	771	36	Masks
mask_rcnn_inception_v2_coco	79	25	Masks
mask_rcnn_resnet101_atrous_coco	470	33	Masks
mask_rcnn_resnet50_atrous_coco	343	29	Masks

Figure 3.5.7: TensorFlow Detection Model Zoo (COCO Trained Models)

On the one hand, the training and validation stages will be done on a laptop or desktop PC to offload the end-point device. Besides, by doing these on a PC rather than on the board, one has access to more resources and can even run TensorFlow with GPU support.

On the other hand, the testing phase will be done on the edge (Raspberry PI 4B), to analyze if the device can handle the computations of the ML model whilst making accurate predictions about the test data.

To develop the ML model, one intends to follow the path highlighted in figure 3.5.8.

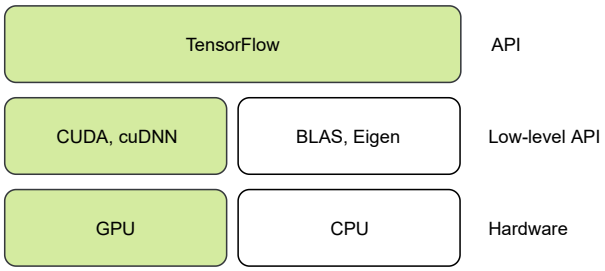


Figure 3.5.8: TensorFlow on the GPU

3.5.5 Processes

This section will describe every system process and, if necessary, complement that description with specific flowcharts.

Processes List

- Head Process;
- Shutdown Daemon;

Head Process

The head process will be comprised of all the threads that will be later discussed in section 3.5.6. In this regard, it will handle all the functionalities related to the camera, the inference engine, and the Wi-Fi, cloud, database, and remote client connections.

Shutdown Daemon

The shutdown daemon will be responsible for checking if the power button was pressed and perform the soft/brute force shutdown depending on the case. After its creation following the usual steps for creating a daemon, the latter will check for button presses in an infinite loop. If the user intends to turn the system off the daemon will proceed to launch the termination signal and verify if the programs closed successfully. If any program doesn't meet the stipulated deadline the system is going to preemptively respond with a brute force shutdown, turning off the device. The flowchart for the daemon can be seen in figure 3.5.9.

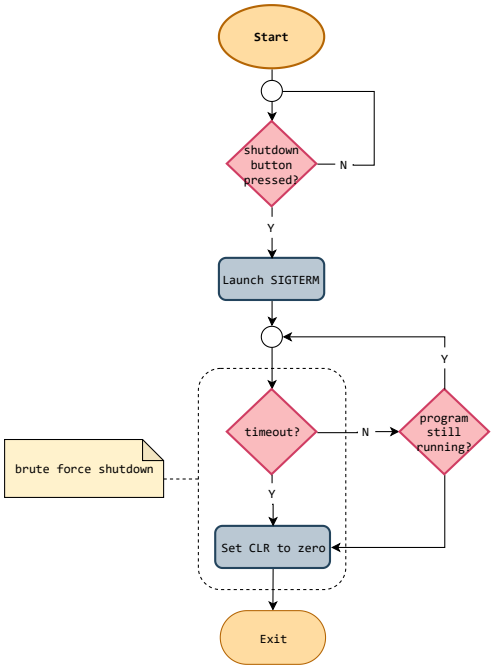


Figure 3.5.9: Shutdown Daemon Flowchart

3.5.6 Threads

In this section, every main system thread will be designed, specifying their priorities, communications, and their intended workflow through the use of flowcharts.

Thread List

For identifying which tasks are suitable for threading and making their assignment and distribution, following criteria may be applied [14]:

1. It is independent of other tasks.
2. It can become blocked in potentially long waits.
3. It can use a lot of CPU cycles.
4. It must respond to asynchronous events.
5. Its work has greater or lesser importance than other work in the application.

The thread list for the **Computational Unit**, which was formulated taking into consideration the previously mentioned criteria, is introduced below:

- **tMain** or main task, which controls the overall flux of the application;
- **tCameraControl**, which initializes the CSI and I^2C interfaces as well as the light sensor, periodically samples the light intensity and changes the camera exposure accordingly;
- **tMLInference**, which, after preparing the ML model for inference, periodically requests a camera frame and makes predictions, signaling threatening events to other tasks;
- **tCloud**, which connects to the Cloud Storage service and uploads relevant media to the Cloud;
- **tDatabase**, which connects to the Remote Database and populates it with information pertaining to relevant events;
- **tRemoteClient**, which accepts a connection from the Remote Client and notifies it of a threat on command;

Thread Priority Level Assignment

As foresaid, one needs to assign each thread a static priority level to indicate their relative urgency. Moreover, the scheduler will always pick the thread that is ready to execute with the highest priority level. Note that an unsuitable assignment of thread priorities might result in system performance loss and unresponsiveness, since the processor could be overtaken by a single high hierarchy thread. Considering this, the thought out priority assignment diagram is represented in figure 3.5.10.

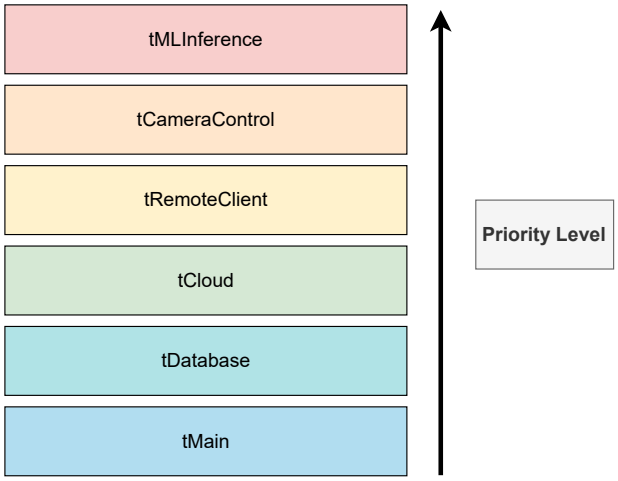


Figure 3.5.10: Priority Assignment Schematic

Thread Timeline

The schematic in figure 3.5.11 shows the main flow of the application, the thread hierarchy, thread life cycle and the main synchronization points.

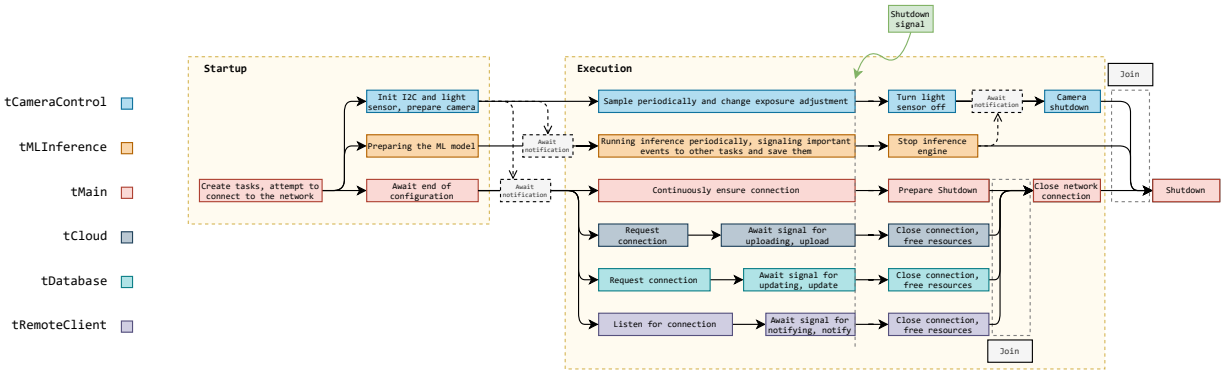


Figure 3.5.11: Thread Timeline (Aug. in Appendix D.1)

Thread Communication

The communication and access control of the threads will be performed by the following objects:

Condition Variables

- **cTerminate**, soft shutdown condition, all threads check it to guarantee a smoother system power off;
- **cCameraInit**, condition necessary to pass the information that the camera's setup was executed;
- **cInferenceStopped**, condition which indicates the cease of inference so one can close the camera on a soft shutdown;
- **cThreatFound**, condition that notifies that a threat had been found;

Mutexes

- **mCamera**, mutex that controls the access to the camera;
- **mNotifyCache**, mutex that controls the access to the notification cache on the local system;
- **mFrameCache**, mutex that controls the access to the image/frame cache on the local system;
- **mMetaCache**, mutex that controls the access to the event/metadata cache on the local system;

The schematic of figure 3.5.12 represents an overview on how the aforementioned threads will communicate taking the objects into account.

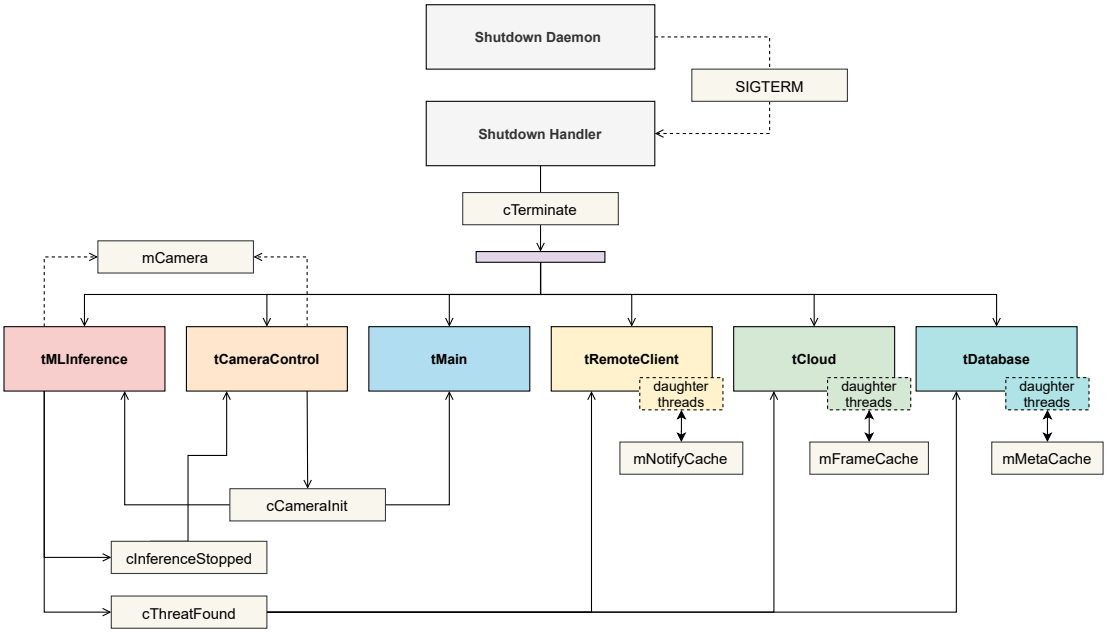


Figure 3.5.12: Thread Communications Overview

Thread Algorithms

All the thread workflow algorithms will be presented in this section.

tMain

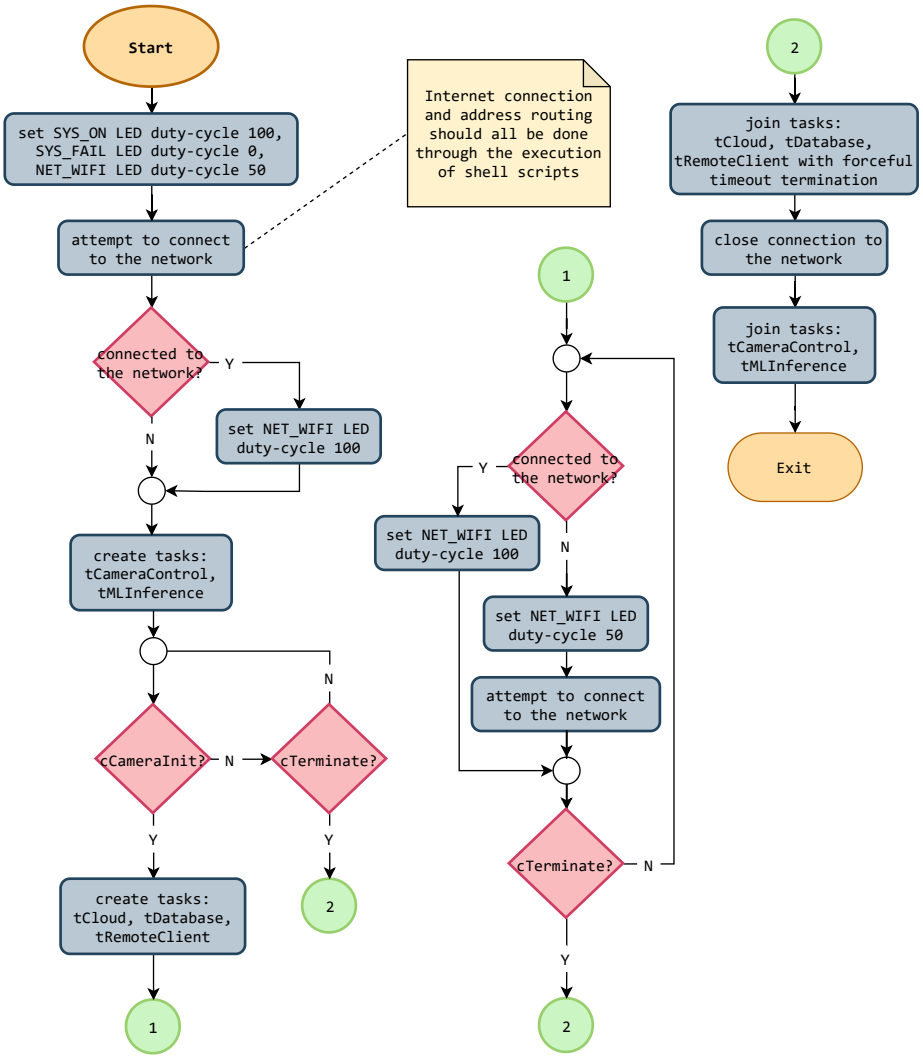


Figure 3.5.13: tMain Flowchart

tCameraControl

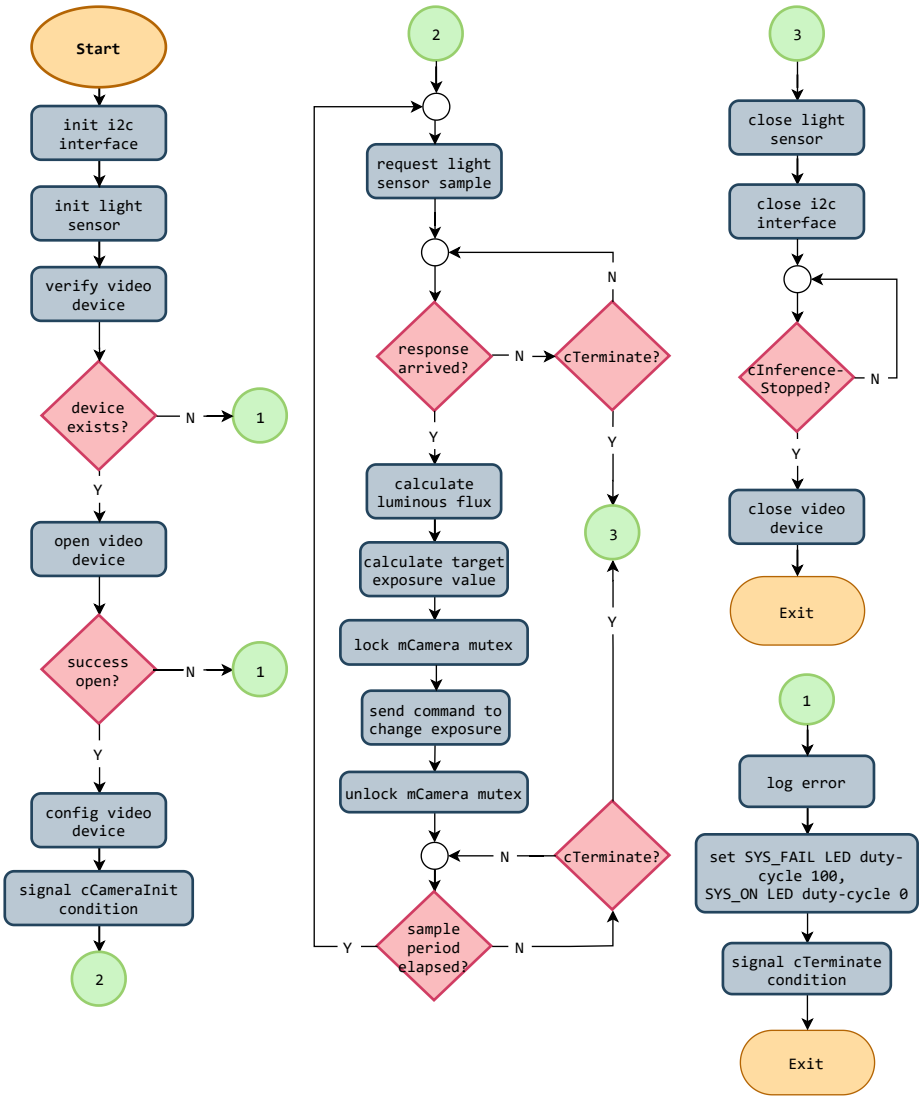


Figure 3.5.14: tCameraControl Flowchart

tMLInference

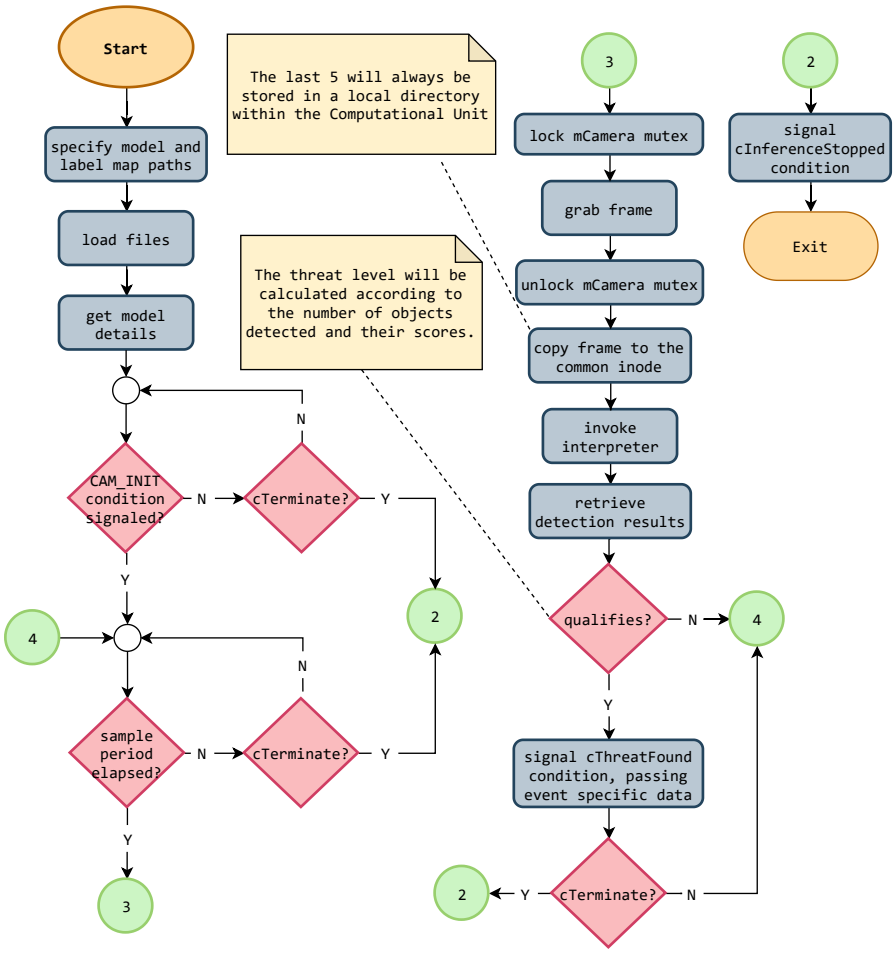


Figure 3.5.15: tMLInference Flowchart

tCloud

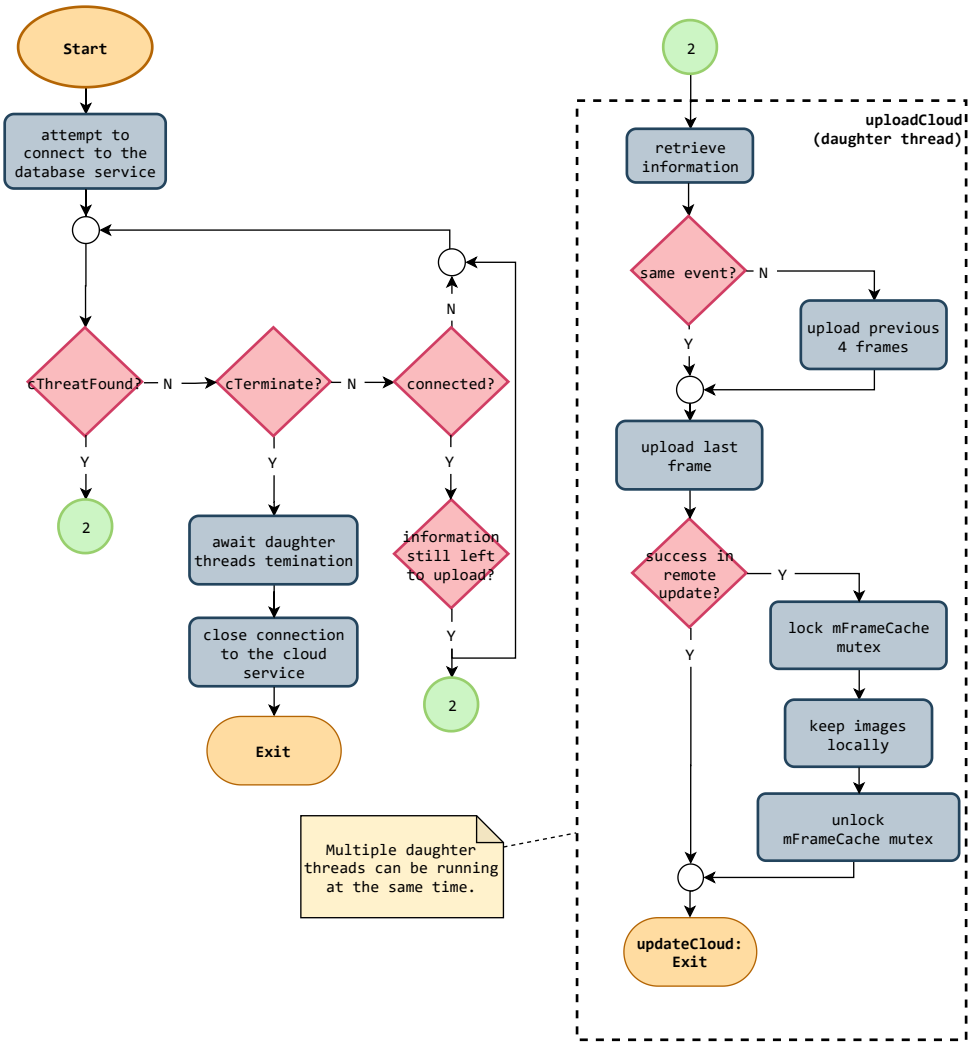


Figure 3.5.16: tCloud Flowchart

tDatabase

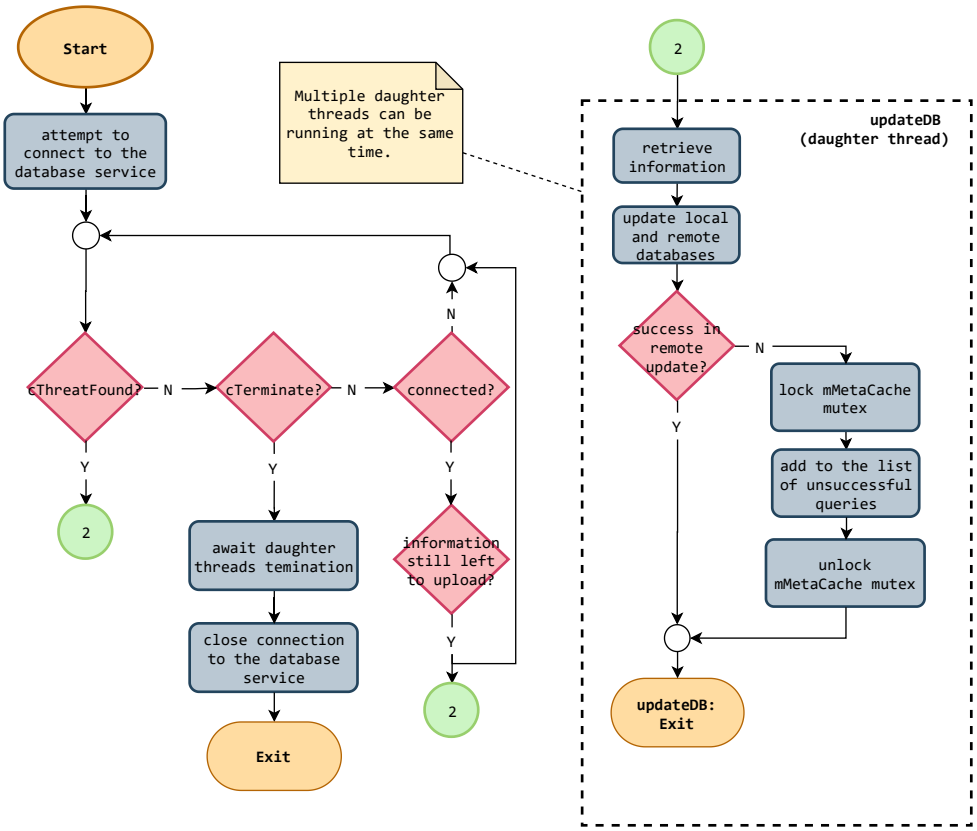


Figure 3.5.17: tDatabase Flowchart

tRemoteClient

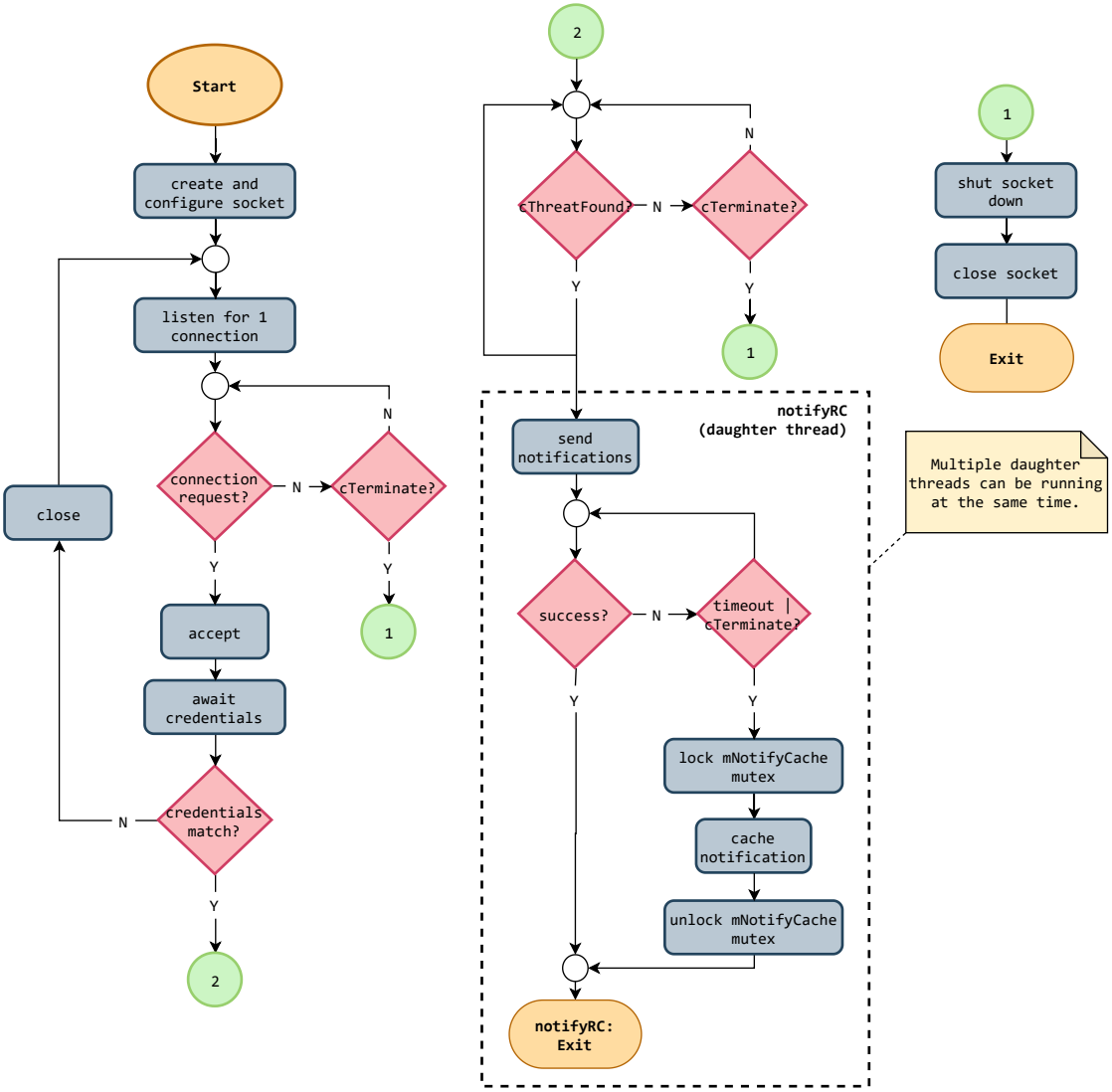


Figure 3.5.18: tRemoteClient Flowchart

3.5.7 Kernel Image Dependencies

Buildroot will be used to generate a custom Linux Kernel image with only the necessary packages for the project. This is a good way to reduce the image size and yet have it to be target-specific. The tool has a simple interface, accessed in the buildroot directory only by tipping on the terminal `$ make menuconfig`. The default configuration offered for the version of the project's Raspberry Pi must be run before entering the GUI with the command `$ make raspberrypi4_defconfig`. From then on, buildroot presents a myriad of packages and configuration options to choose from. Take note that unlike other image-generating tools, buildroot doesn't support on-target package management, meaning that images are rather static with a fixed size. Although applications can change the filesystem, one cannot install new packages on a working system, which means that the image must be thought out at this stage (design) to avoid future setbacks.

Regarding the base packages necessary for nearly every system, the project will have the following:

- **`gdb`**, standard GNU debugger;
- **`gdbserver`**, for debugging purposes;
- **`dropbear`**, lightweight SSH client and server system;
- **`bzip2`**, data compression program;

Concerning the packages/configurations necessary for peripheral interface and some other useful functionalities, one will also include the following:

- **`v4l2` (Video4Linux2)**, collection of device drivers and an API for real-time video capture support on Linux systems;
- **`opencv`**, API for image processing;
- **`dhcp`**, Dynamic Host Configuration Protocol server;
- **`mysql`**, SQL database system;
- **`i2c-tools`**, heterogeneous set of I^2C tools for Linux;
- **`libcurl`**, command line tool for transferring files with URL syntax;
- **`arping`**, ARP level ping utility;
- **`arpwatch`**, ethernet monitor program for keeping track of ethernet/ip address pairings;
- **`wpa-supplicant`**, support for WPA and WPA2;

3.6 Remote Client: SW Specification

This section aims to describe all the software relating to the ARGOS' Remote Client and explicitly define the intended workflow of the system unit.

3.6.1 Front-end: Graphical User Interface (GUI)

This section will present the initial mockups for the Graphical User Interface of the remote client, as well as a brief description of the functionalities it offers. Firstly, the user will be presented with a window where the login will be made, introducing the machine credentials. If the authentication was successful, the current window will shift to one where it is possible to:

- **Left section**, see a list of recent events;
- **Middle section**, see important information about the event/frame selected;
- **Right section**, see the frames/images captured by the camera with gun predictions and respective scores (certainty associated with the predictions);

Mockups

The aforementioned mockups are ilustrated in the figures 3.6.1 and 3.6.2.

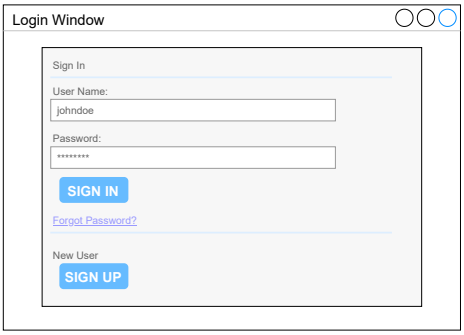


Figure 3.6.1: GUI - Login Window

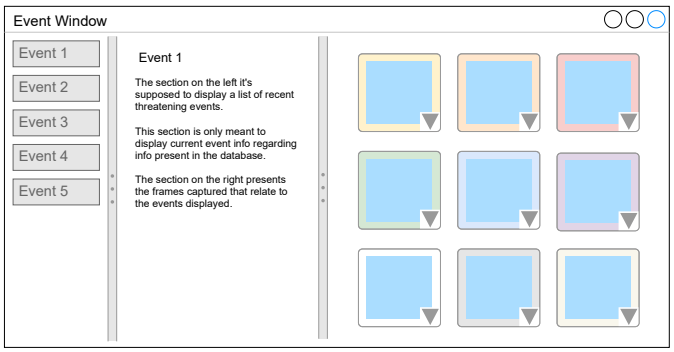


Figure 3.6.2: GUI - Event Window

3.6.2 Back-end

Initialization Sequence

The initialization process in the Remote Client starts with the application asking the user for the machine credentials and logging into the database and cloud services. Without all these elements, the application cannot start up as the information gathered from each of these sources makes little sense on its own.

Right upon startup, it should download all available information from all sources and readily present it to the user.

Execution

During execution, the application should continuously poll for new events from the database and cloud services and propagate every bit of information onto the GUI.

Exit Sequence

When a soft shutdown is requested, the application should finish all ongoing transactions and log out of the cloud and database services. Most importantly, though, it should explicitly disconnect from the computational unit.

3.7 Remote Storage

The **Remote Storage** is highlighted in the system overview of figure 3.7.1.

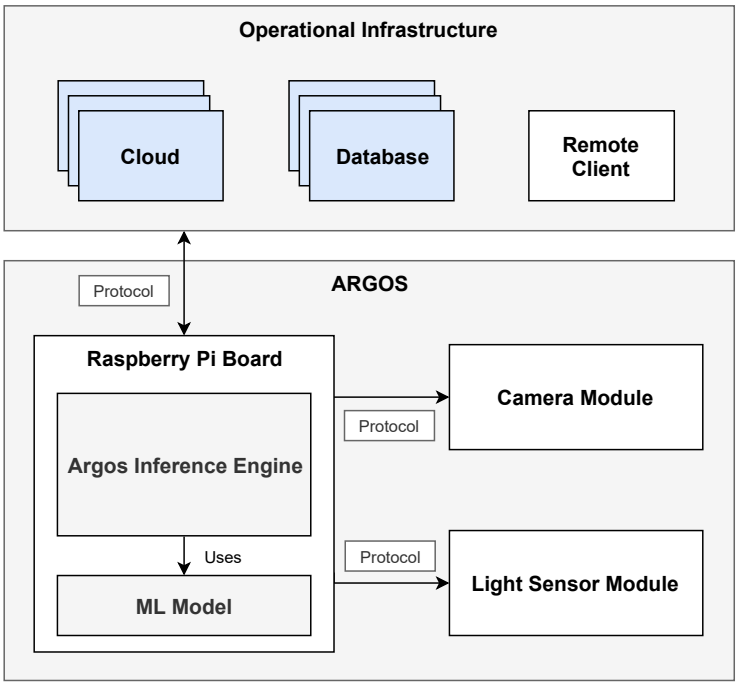


Figure 3.7.1: System Overview Diagram - Remote Storage

This section is dedicated to the database design, specifically, of the complete Entity Relationship Diagram (ERD), logic model, and the overall interactions with the database and cloud.The management of the latter will essentially rely on the two API referred to in section 3.4.2.

3.7.1 Storage Framework

The general overview of the interactions between ARGOS, the user, and the environment is depicted in figure 3.7.2. Following this diagram, one intends to implement a local redundancy framework to assure data and metadata (*"data about data"*) delivery in case of communication failure. The system will be able to continuously store and update data on a fixed raspberry inode, whilst uploading it to the online database and cloud to be accessed by the user.

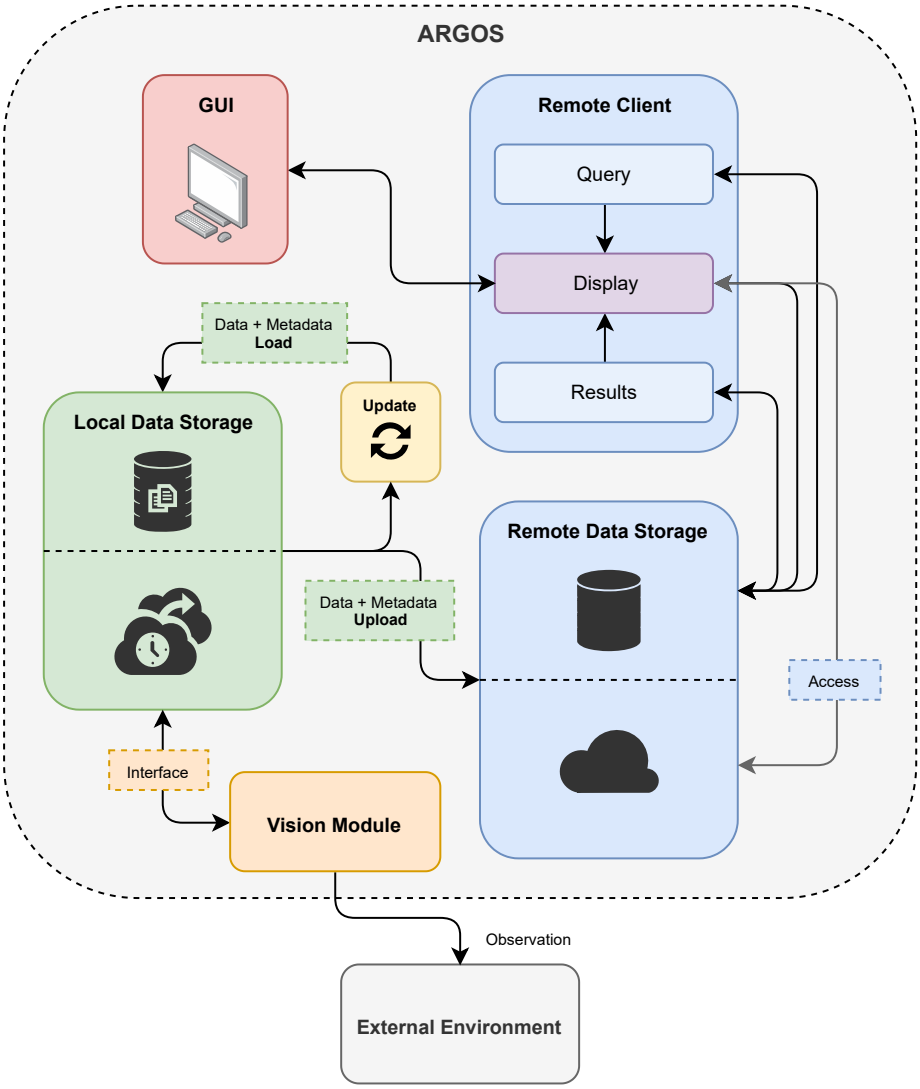


Figure 3.7.2: Remote Storage Overview

3.7.2 Database

In this section, one aims to continue with the database design based on the database analysis performed in section 2.9, with the first Entity Relationship Diagram (ERD).

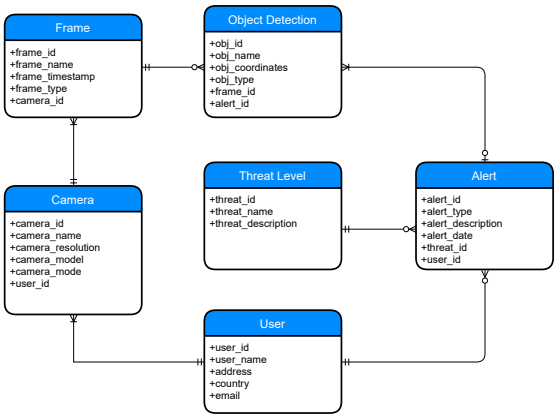


Figure 3.7.3: ARGOS Database ER Diagram (Aug. in Appendix A.4)

Logic Model

Following, one must identify the primary and foreign keys for each entity. Note that, as in the previous diagram, entities that are on the "many" side of a one-to-many relationship must heir the primary key of the other entity as a foreign key. Additionally, those foreign key names were kept at this stage to make the diagram easier to interpret, but they can be switched into more meaningful ones at the implementation phase. The database's logic diagram is depicted in figure 3.7.4.

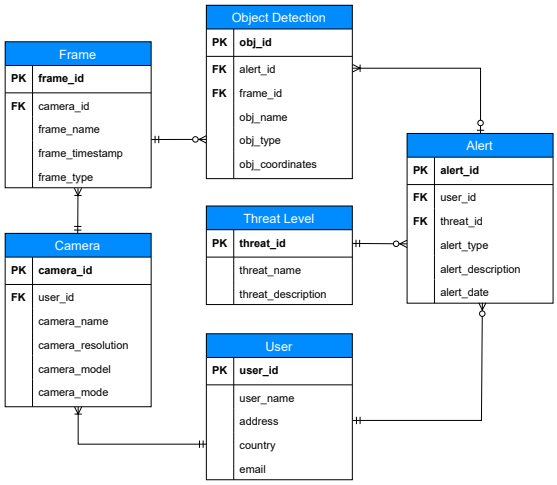


Figure 3.7.4: ARGOS Database ER Logic Diagram (Aug. in Appendix A.5)

3.8 Structure Design

As with every part of the system, the physical structure also undergoes a design phase. One can observe the camera's technical drawing in figure 3.8.1, and the 3D model with a Raspberry Pi representation in figure 3.8.3.

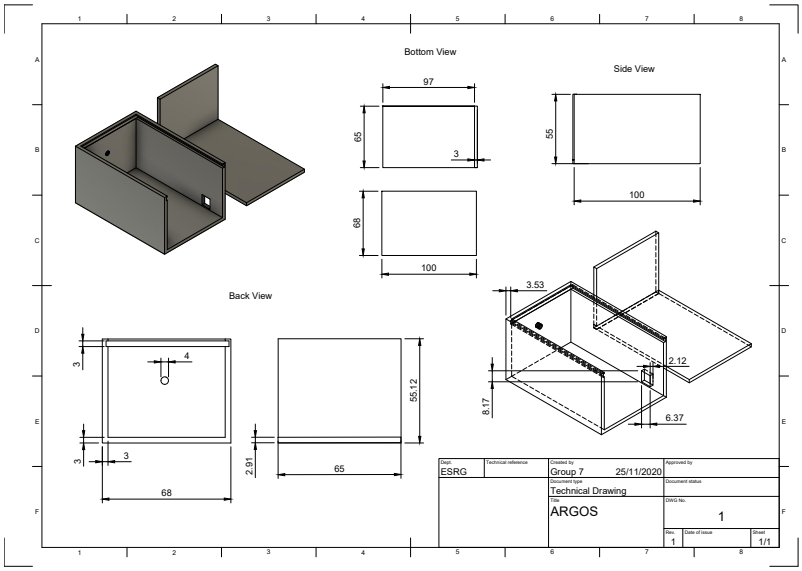
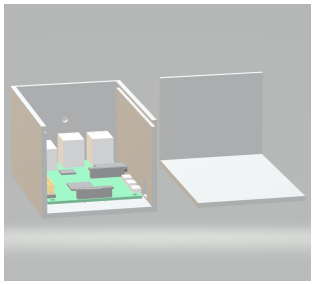
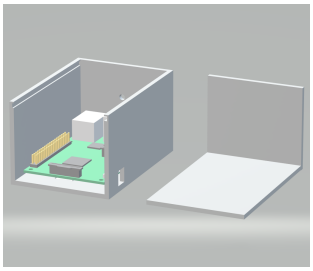


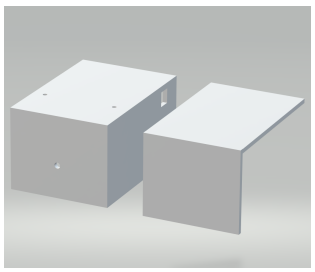
Figure 3.8.1: ARGOS Technical Drawing (Aug. in Appendix E)



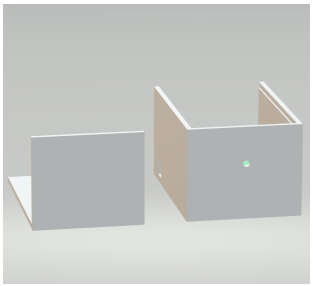
(a) Left Side view



(b) Right Side view



(a) Bottom View



(b) Front View

Figure 3.8.3: ARGOS 3D Model w/ Raspberry Pi

3.9 Computational Unit Test Cases

3.9.1 Computational Unit: Unit Tests

Computational Unit: Unit Tests	Expected Results	Real Results
Press the power on button while the device is off	Successful power on sequence and connection with the Remote Client, Database and Cloud	
Press the power on button while the device is on	Disconnection from all remote systems and successful power off sequence without timeout	
Display a prototype weapon in front of the camera	The device identifies the threat	
Trigger weapon detection without connection to database, cloud or Remote Client and then connect	The device stores the data locally and, upon connection, sends the data, guaranteeing no repetitions and deleting unnecessary local data	
Change amount of light in the area around the camera	Light sensor readings change accordingly	
Connect to the Wi-Fi	Connection successful	
Control LEDs	LEDs change accordingly	
Read button state	Button state read correctly	

Table 3.1: Computational Unit Test Cases

3.10 Remote Client Test Cases

3.10.1 Remote Client: Unit Tests

Remote Client: Unit Tests	Expected Results	Real Results
Open app	App opens and displays login screen	
Insert credentials	App attempts to establish a connection with the Computational Unit	
Connection rejected	Show warning and allow for credentials reinsertion	
Connection accepted	Go into the control panel, immediately fetching pertinent information from the cloud and database	
Click specific event button	Show frames and metadata about that event	

Table 3.2: Remote Client: Unit Tests

3.11 Integration Tests

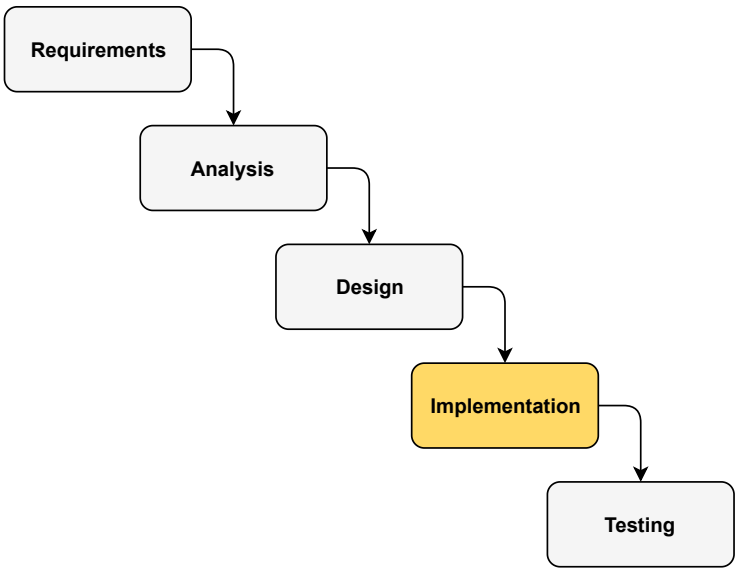
Integration Tests	Expected Results	Real Results
Display prototype weapon in front of the camera	Weapon is identified and frames and data start appearing in the Remote Client's GUI	
Turn off camera	Warning shows up in Remote Client and device turns off within the specified time	

Table 3.3: Integration Tests

Chapter 4

Implementation

This chapter will present the implementation guidelines one must follow to achieve the intended prototype attending to what was stipulated in the design phase. All the steps will be described in detail to ensure rich product documentation. This matters, as the document might be used for future reference.



4.1 Kernel Image Generation

As mentioned in section 3.5.7, there are several package dependencies related to the application. Considering this, the latter need to be included in the Linux kernel image to deploy in the Raspberry Pi 4B alongside with some extra system-specific configurations through the Buildroot tool. By issuing some commands in the buildroot's directory that allow for the Raspberry 4 default configuration, grant access the configuration menu, one can perform the aforementioned task and successfully generate (make) the image to introduce in the SD card. To ensure that the image structure is understood, this section will lay out the most important packages and configurations for the ARGOS's Linux system. In table 4.1 it is possible to observe an overview of the packages leveraged for the application.

Target Package	Included	External	Target Package	Included	External
BusyBox	✓	—	dropbear	✓	—
ffmpeg	✓	—	ifupdown	✓	—
v4l2grab	✓	—	ifupdown scripts	✓	—
v4l2loopback	✓	—	iputils	✓	—
v4l-utils tools	✓	—	arping	✓	—
nano	✓	—	clockdiff	✓	—
util-linux	✓	—	ping	✓	—
bzip2	✓	—	rdisc	✓	—
imagemagick	✓	—	tracepath	✓	—
i2c-tools	✓	—	traceroute6	✓	—
python	✓	—	ninfod	✓	—
gnutls	✓	—	iw	✓	—
cppdb	✓	—	wireless tools	✓	—
mysql support	✓	—	wpa_supplicant	✓	—
libcurl	✓	—	wpan-tools	✓	—
arp-scan	✓	—	kill	✓	—
crda	✓	—	rfkill	✓	—
dhcpcd	✓	—	Tensorflow	—	✓
opencv3	✓	—			

Table 4.1: Packages Overview

4.1.1 Target Packages

BusyBox combines small versions of many common UNIX utilities into a single and light executable file. This package is crucial since it provides replacements for most of the utilities one usually finds in GNU fileutils, shellutils, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU counterparts, however, the options that are included behave similarly and provide the expected functionalities. BusyBox was thought out as an embedded-targeted package and, as such, it has been written with size-optimisation and limited resources taken into consideration. Additionally, it is also extremely modular so you can easily include or exclude commands (or features)

at compile time, making it pretty customizable. In figure 4.1.1 it is possible to see the selection of the aforementioned package under the Target Packages section.

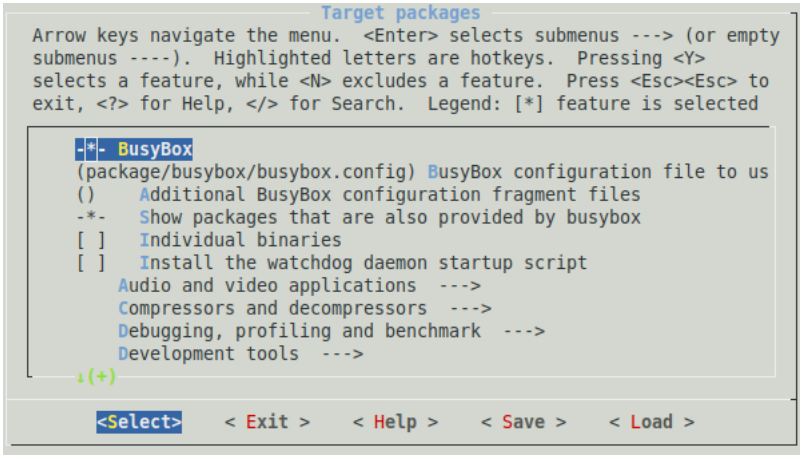
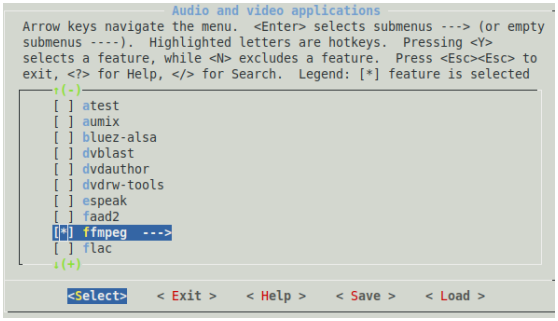


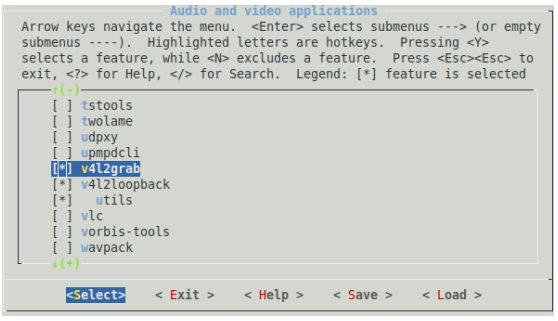
Figure 4.1.1: Buildroot Target Packages: BusyBox

The package **ffmpeg** is a cross-platform audio/video conversion tool. In the case of ARGOS, the package will be used as a frame grabber to capture frames from the camera's video stream.

The latter is often associated with **Video4Linux2 (V4L2)** packages to leverage multimedia devices. While the v4l2grab allows for frame capture from V4L2 devices, being similar to the one provided by the libv4l package, the v4l2loopback allows one to create virtual video devices. So that, applications that rely on v4l2 can interact with such devices as if they were real, whilst the video comes from other applications. These packages can be seen in figure 4.1.2 under the Target Packages → Audio and video applications.



(a) Buildroot Target Packages: ffmpeg



(b) Buildroot Target Packages: V4L2

Figure 4.1.2: Buildroot Target Pakages: Vision System

The package **nano** is just a simple text editor that was enabled to be easier to edit some files from within the embedded environment. On the one hand, **util-linux** yields a diverse collection of Linux kernel utilities. It provides dmesg and includes tools for working with file systems, block devices, UUIDs, TTYs, and many other tools, including **kill** (kill a process via signal) and **rfkill** which is a tool for enabling and disabling wireless devices. On the other

hand, **gnu-utils** adds some basic commands like `ls` or `rm`. The package **bzip2** was added because a compression program might be handy in some situations.

Moreover, **imagemagick** provides some image processing utilities to create, edit, compose, or convert digital images. It can read and write images in a variety of formats (e.g., PNG, JPEG, GIF, HEIC, TIFF, DPX, EXR, WebP, PDF, and SVG). ImageMagick can resize, flip, mirror, rotate, distort, shear and transform images, adjust image colours, apply various special effects, or draw text, lines, and shapes. In this case, that will be useful to draw the detection boxes and labels based on the inference results. Figure 4.1.3 shows the selection of the imagemagick package under Target Packages → Graphic libraries and applications (graphic/text).

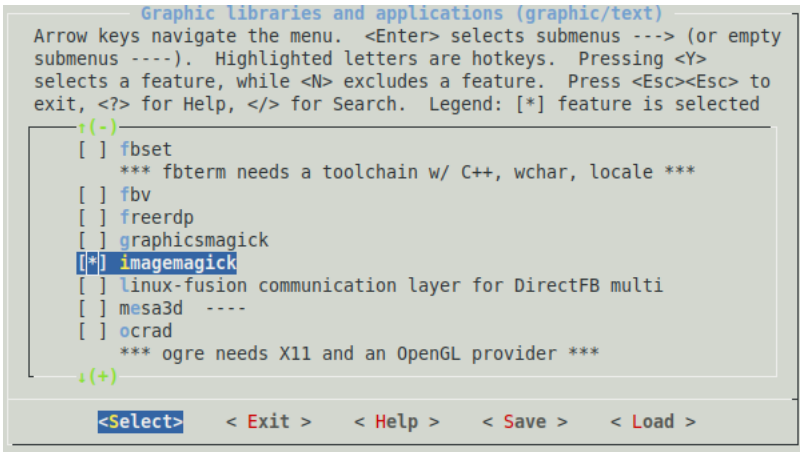


Figure 4.1.3: Buildroot Target Packages: imagemagick

The package **i2c-tools** adds a set of I^2C tools to interface with specific hardware. In this case, one intends to interface with the light sensor via I^2C. The figure 4.1.4 depicts the selection of this package under the section Target Packages → Hardware handling.

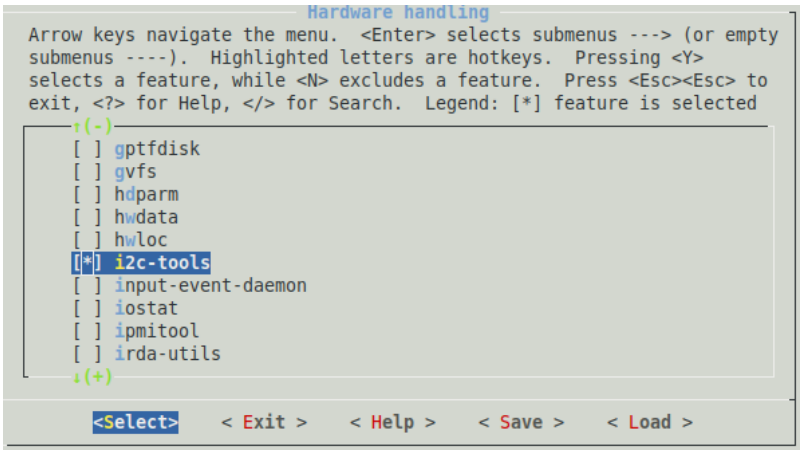


Figure 4.1.4: Buildroot Target Packages: i2c-tools

The package **python** is self-explanatory, it provides a python language interpreter. This will be necessary since

one intends to run inference with a python script. The aforementioned package can be enabled under the section Target Packages → Interpreter languages and scripting, as represented in figure 4.1.5.

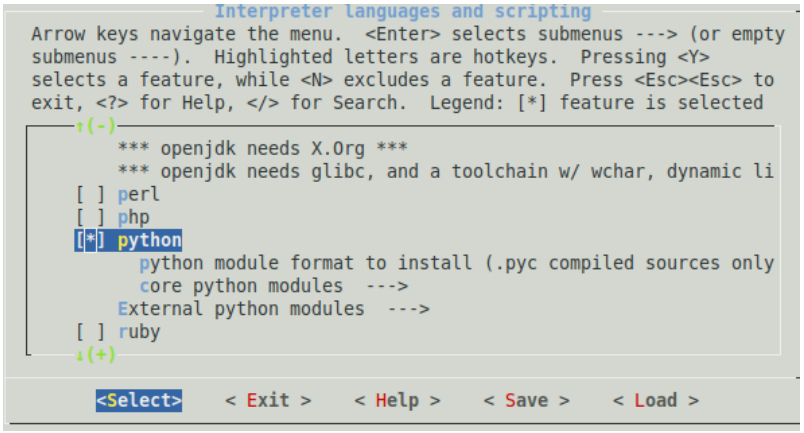
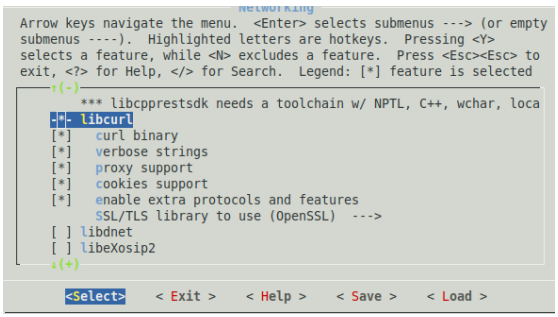


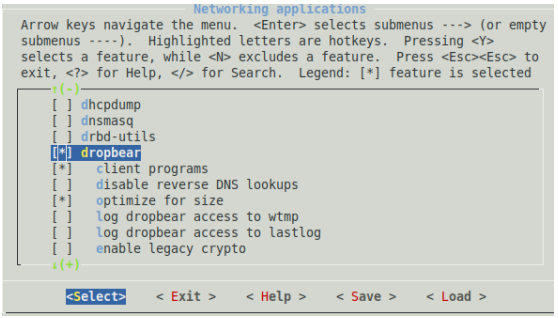
Figure 4.1.5: Buildroot Target Packages: python

The package **libcurl** is a tool for transferring files with URL syntax, supporting FTP, FTPS, HTTP, HTTPS, GOPHER, TELNET, DICT, FILE and LDAP. In ARGOS's case the latter was added to assure one can use curl if necessary. Figure 4.1.6a depicts the selection of the package under Target Packages → Libraries → Networking.

The package **dropbear** is a light SSH server and client system, broadly used in embedded Linux devices. The SSH protocol uses encryption to secure the connection between a client and a server. All user authentication, commands, output, and file transfers are encrypted to protect against attacks in the network. In the case of the application, this tool will be used to link the Raspberry to the PC. Figure 4.1.6b represents the selection of the package under Target Packages → Networking applications.



(a) Buildroot Target Packages: libcurl



(b) Buildroot Target Packages: dropbear

Furthermore, **arp-scan** is a tool that uses the ARP protocol to discover and fingerprint IP hosts on the local network; **crda** is the Linux wireless central regulatory domain agent; **dhcpcd** is an implementation of the DHCP client. It gets the host information (IP address, routes, etc) from a DHCP server and configures the network interface of the machine on which it is running; **ifupdown** and **ifupdown scripts** correspond to high-level tools to configure network interfaces based on interface definitions in the file /etc/network/interfaces; **iputils** is a set of small utilities for Linux networking (e.g., ping, traceroute, arping); **iw** is a tool to show/manipulate wireless devices and their configuration.

Its used in devices with the mac80211 kernel stack (Linux Wi-Fi drivers); **wireless tools** is a package to configure WLAN cards; **wpa_supplicant** is a cross-platform supplicant with support for WEP, WPA and WPA2 (IEEE 802.11i) for secure wireless networks (figure 4.1.7); **wpan-tools** offers a configuration interface for WPAN networks. It allows to configure various PHY and MAC layer parameters and properties.

All of these tools were used to configure the Wi-Fi connection in the edge device (Raspberry) and can be found under section Target Packages → Networking applications.

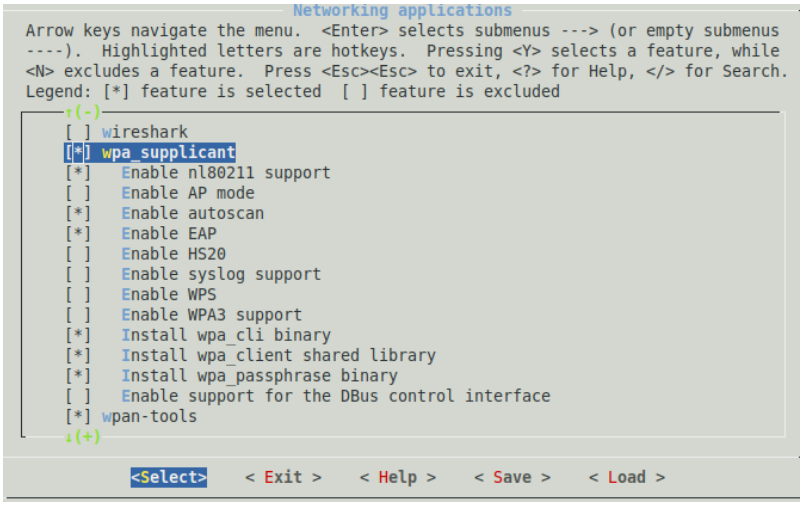
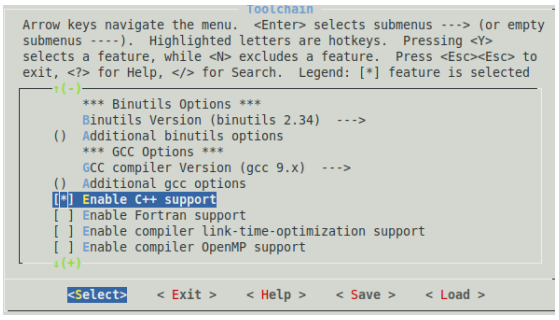


Figure 4.1.7: Buildroot Target Packages: wpa_supplicant

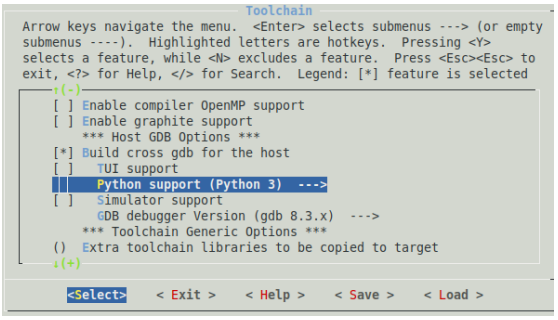
The package **opencv3** was also needed for opening the frames within the python script. The latter was found under the section Target packages → Libraries → Graphics.

Regarding to the **Tensorflow** package, it's not possible to add the latter to the kernel image with Buildroot. Although, one found another way to add the libraries for TensorFlow to the system. This will be discussed later in section 4.8.5.

Concerning the system configurations, is important to enable both python and C++ support for the aforementioned reasons. In figures 4.1.8a and 4.1.8b one can see the enabling of these options under the section Toolchain.



(a) Buildroot Toolchain: C++ Support



(b) Buildroot Toolchain: Python Support

4.2 Device Drivers

4.2.1 General-Purpose Input/Output (GPIO)

The access to fundamental logic input and output in ARGOS is done with resource to a custom character device driver for General-Purpose Input/Output (GPIO). This simple device driver provides the functionality necessary for changing the logic level of the microprocessor's GPIO pins configured as outputs and reading the logic level present in pins configured as inputs. For that, this kernel module implements the classic `read`, `write` and `ioctl` interfaces.

Hardware Interface

In order to understand how the module writes to and reads from the MPU's pins, it is essential to know how it interacts with the hardware. All fundamental operations in the module involve accessing or writing to the registers in the GPIO peripheral of the MPU. These are organised and interpreted as a structure of the type `GPIORegisters_t`, which encapsulates the important registers for this project. While the `GPFSEL` registers allow for selecting the mode of operation for each pin, the `GPFSET` and `GPFCLR` registers allow for setting or clearing a pin, respectively. The `GPFLEV` register allows one to read the digital state of an input pin.

```
typedef struct GPIORegisters_t {

        uint32_t GPFSEL[6];
        uint32_t reserved1;
        uint32_t GPSET[2];
        uint32_t reserved2;
        uint32_t GPCLR[2];
        uint32_t reserved3;
        uint32_t GPLEV[2];

} GPIORegisters_t;
```

Listing 4.1: GPIO device driver's Hardware Interface

Write

Writing to a pin in ARGOS is done through a call to the `write()` function, providing a string in the format `"%s, %s"` containing the number of the pin of which the logic level is to be switched as well as a number indicating the value it is to be switched to, as explicit in listing 4.2.

```
/* gpiomodule.c */

ssize_t gpio_write(struct file *pfile, const char __user *pbuff, size_t len, loff_t *off) {

        int pin_number, pin_value;
        GPIORegisters_t *pdev;

        pr_alert("%s: called (%u)\n",__FUNCTION__,len);

        /* Check for file nullity */
```

```
        if(unlikely(pfile->private_data == NULL))
                return -EFAULT;

        /* Parse input */
        pr_alert("%s:pbuff.str = %s\n", __FUNCTION__, pbuff);
        sscanf(pbuff, "%u,%u", &pin_number, &pin_value);
        printk("%s: parsed values: pin_number = %u, pin_value = %u\n", __FUNCTION__,
            pin_number, pin_value);

        if (configs[pin_number] != OUTPUT) {
                printk("%s: pin %u is not configured as an output\n", __FUNCTION__,
                    pin_number);
                return -1;
        }

        /* Assert pin number's validity */
        if (pin_number<0 || pin_number>=MAX_PIN_NUMBER)
                return -EBADRQC;

        /* Fetch peripheral address */
        pdev = (struct GPIORegisters_t *)pfile->private_data;

        /* Set pin value */
        setGPIOOutputValue(pdev, pin_number, pin_value == 0 ? 0 : 1);

        return len;
}


/* gpio_util.c */

void setGPIOOutputValue(struct GPIORegisters_t *gpio_regs, int gpio, bool output_value) {

        pr_alert("%s: register value is 0x%x\n",__FUNCTION__,(1<<(gpio %32)));

        if (gpio_regs == NULL)
                return;

        if (output_value)
                gpio_regs->GPSET[gpio / 32] = (1 << (gpio % 32));
        else
                gpio_regs->GPCLR[gpio / 32] = (1 << (gpio % 32));
}
```

Listing 4.2: GPIO device driver's `write()` routine

Read

Reading from a pin in ARGOS is done through a call to the `read()` function, providing the number of the pin to be read. The value is sent back to the user space through a pointer.

```
/* gpiomodule.c */

ssize_t gpio_read(struct file *pfile, char __user *p_buff, size_t len, loff_t *poffset){

        int pin_number, pin_value;
        GPIORegisters_t *pdev;

        /* Check for file nullity */
        if(unlikely(pfile->private_data == NULL))
                return -EFAULT;

        /* Fetch peripheral address */
        pdev = (struct GPIORegisters_t *)pfile->private_data;

        /* Get pin number Fetch peripheral addressfrom user string */
        // pin_number = strToInt(p_buff);

        pin_number = len;
        printk("%s: pin number: %d\n", __FUNCTION__, pin_number);

        /* Assert pin number's validity */
        if (pin_number<0 || pin_number>=MAX_PIN_NUMBER)
                return -EBADRQC;

        if (configs[pin_number] != INPUT) {
                printk("%s: pin %u is not configured as an input\n", __FUNCTION__,
                    pin_number);
                return -1;
        }

        /* Get pin value */
        pin_value = getGPIOInputValue(pdev, pin_number);

        printk("%s: pin value: %d\n", __FUNCTION__, pin_value);

        // copy_to_user(buf, memory_buffer, 1);
        raw_copy_to_user(p_buff, (uint8_t*)&pin_value, 1);

        return 0;
}

/* gpio_util.c */

uint32_t getGPIOInputValue(struct GPIORegisters_t *gpio_regs, int gpio) {

        if (gpio_regs == NULL)
                return -EFAULT;
```

```
        printk("%s: value: %d\n", __FUNCTION__, (gpio_regs->GPLEV[gpio/32] >> gpio%32) & 1);

        return (gpio_regs->GPLEV[gpio/32] >> gpio%32) & 1;
}
```

Listing 4.3: GPIO device driver's `read()` routine

I/O Control

General configurations on the GPIO are to be done through a call to the `ioctl()` method. The only configuration implemented in the module is the assignment of function to each individual pin. This function can be defined as either `INPUT` or `OUTPUT` their configuration requires a pointer to a `GPIOPinFunction_t` structure containing the number of the pin of which the logic level is to be switched as well as a number indicating the value it is to be switched to. This is visible in listing 4.4.

```
/* gpio_util.h */

typedef struct GPIOPinFunction {

        uint32_t pin;
        GPIOFunction_t function;

} GPIOPinFunction_t ;


/* gpiomodule.c */

long gpio_ioctl(struct file* pfile, uint32_t request, long unsigned int value) {

        GPIORegisters_t *pdev;

        pr_alert("%s: called\n",__FUNCTION__);

        switch (request) {

        case GPIO_IOCTL_FUNCTION:

                pr_alert("%s: Defining GPIO function \n",__FUNCTION__);

                /* Check for file nullity */
                if(unlikely(pfile->private_data == NULL))
                        return -EFAULT;

                /* Fetch peripheral address */
                pdev = (struct GPIORegisters_t *)pfile->private_data;

                /* Set required function */
                setGPIOFunction(pdev, (GPIOPinFunction_t*)value);
```

```
34
                return 0;

        default:
                pr_alert("%s: Invalid request\n",__FUNCTION__);
39              return -EBADRQC;
        }
}

/* gpio_util.c */
44
void setGPIOFunction(struct GPIORegisters_t *gpio_regs, GPIOPinFunction_t* data) {

        int gpio = data->pin;
        int function_code = data->function;
49
        int register_index = gpio / 10;     /* GPFSEL register */
        int bit = (gpio % 10) * 3;          /* Position of first bit in register */

        unsigned old_value = gpio_regs->GPFSEL[register_index];
54      unsigned mask = 0b111 << bit;

        /* Keep the old value of GPFSELx generally but write the function code bits */
        gpio_regs->GPFSEL[register_index] = (old_value & ~mask) | ((function_code << bit) &
            mask);
}
```

Listing 4.4: GPIO device driver's `ioctl()` routine

4.3 Daemon

The ARGOS's shutdown daemon follows the standard rules for daemon creation [29]:

1. Perform a fork(), after which the parent exits and the child continues.
2. The child calls setsid() to start a new session and free itself of any association with a controlling terminal;
3. Ensure the daemon isn't the session leader;
4. Clear the process umask or set one that fits the permission needs of the application;
5. Change the process's current working directory, typically to the root directory (/);
6. Close the unused file descriptors (the daemon might need certain inherited file descriptors);

The listing 4.5 represents the shutdown daemon setup. Note that, as stated before, the process for daemon creation starts with the fork call. In listing 4.5, one performs the initial fork creating a new child process, after which the parent process exits and the child proceeds, running in the background. Note that the at this point the daemon isn't a process group leader since its Process ID is different than the inherited Process Group ID.

```
void daemonSetup(){

    /* Fork off the parent process and create a new child process */
    switch (fork()){
    case -1: exit(EXIT_FAILURE);    // An error occurred
    case 0: break;                  // Child proceeds
    default: exit(EXIT_SUCCESS);    // While parent terminates
    }
    ...
```

Listing 4.5: Daemon: daemonSetup (1)

In listing 4.6, the child calls setid to create a new session. The latter then becomes the leader of the new session and the process group leader of the new process group. The process is now detached from its controlling terminal (CTTY).

```
    ...
    /*Start a new session ( setsid() )*/
    /* On success: The child process becomes session leader now detached
        from its controlling terminal (CTTY)*/
    if (setsid() < 0)
        exit(EXIT_FAILURE);
    ...
```

Listing 4.6: Daemon: daemonSetup (2)

In listing 4.7 one can observe the the second fork performed. The parent exits again and the (grand)child continues executing. This ensures that the child is not the session leader (PID != SID) and the process can't reacquire the TTY (terminal).

```
...
/* Fork off for the second time to ensure that the daemon doesn't
    have access to the TTY again (it's not the session leader)*/
switch (fork()){
case -1: exit(EXIT_FAILURE);    // An error occurred
case 0: break;                  // Child proceeds
default: exit(EXIT_SUCCESS);    // While parent terminates
}
...
```

Listing 4.7: Daemon: daemonSetup (3)

In listing 4.8 one clears the process umask, but note that the value for the umask could be different if it was stipulated for the daemon to have fewer permissions, which isn't the case.

```
...
/* Set new file permissions by changing the file mode mask:
    may vary depending on the daemon's permissions */
umask(0);
...
```

Listing 4.8: Daemon: daemonSetup (4)

In listing 4.9 the process' current working directory is changed to the root directory (/). This is done because daemons tend to run until system shutdown, hence it would be a problem if the daemon was running on a different directory that umounted at some point in time. This is troublesome because it is not possible to unmount a file system that is busy. Despite, one could place the daemon on a directory that was certainly never going to be unmounted. In ARGOS's case, to keep it simple, one opted for the daemon working in the root directory.

```
...
/* Change the working directory to the root directory */
/* or another appropriated directory */
chdir("/");
...
```

Listing 4.9: Daemon: daemonSetup (5)

Finally, in listing 4.10, the file descriptors for stdin and stdout are closed since they are unused and the daemon's log file is opened.

```
...
/* Close unused file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);

/* Open the log file */
```

```
    openlog ("shutdownDaemon", LOG_PID, LOG_DAEMON);
}
```

Listing 4.10: Daemon: daemonSetup (6)

In listing 4.11, one creates the daemon with the daemonSetup call. Inside the while loop is the daemon's main thread where it should check for the shutdown command. The application resorts on perror because this function is thread-safe, unlike its stdout counterpart (printf).

```
int main(){

    // Daemon Setup
    daemonSetup();
    perror("[ARGOS] Shutdown Daemon Up and Running");
    syslog (LOG_NOTICE, "[ARGOS] Shutdown Daemon Up and Running");

    // Daemon Main Thread
    while (1){

        // Check for shutdown
        break;
    }

    syslog (LOG_NOTICE, "[ARGOS] Shutdown Daemon Terminated");
    closelog();
    return EXIT_SUCCESS;
}
```

Listing 4.11: Daemon: main

4.4 Vision Module

Regarding ARGOS's vision system, one relied on the **Video4Linux2 (V4L2)** API, since it allows for camera interface to capture raw frames in various pixel formats or encoded stream data, and offers more control over the devices than the typical OpenCV API. The latter is associated with low-level drivers in the kernel space for interfacing with the hardware and also user space API that might serve several multimedia applications. In figure 4.4.1 is represented a shallow overview of the aforementioned framework.

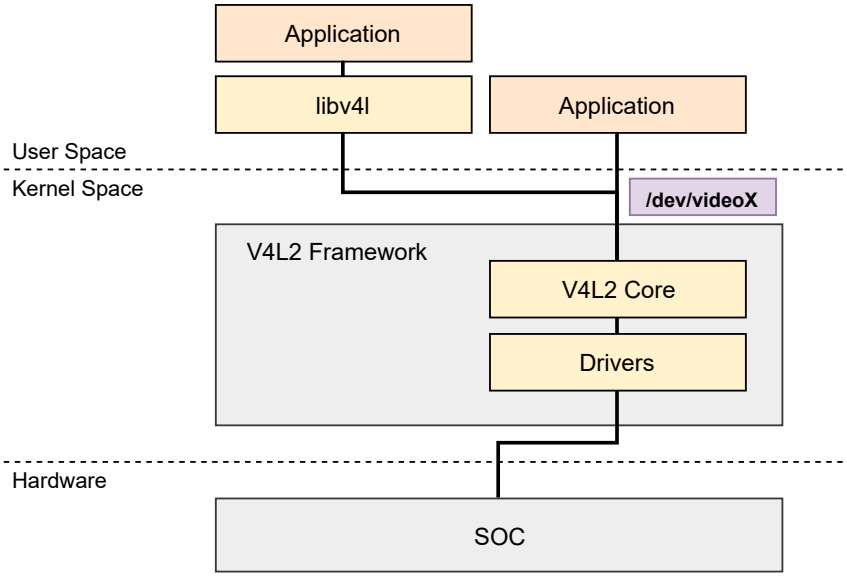


Figure 4.4.1: Video4Linux2 (V4L2) Example Overview

4.4.1 Camera Class

For this specific purpose, one devised a camera class in **C++** capable of interfacing with video devices, capture frames with **ffmpeg**, and perform actions over the frames, specifically with **imagemagick**. The code for the class header file can be found in listing C.1.

A V4L2 application usually follows the following format [26]:

1. Opening a descriptor to the device;
2. Changing device properties, selecting a video and/or audio input, video standard, picture brightness;
3. Negotiating a data format;
4. Negotiating an input/output method;
5. The actual input/output loop;
6. Closing the descriptor to the device;

Constructor and Destructor

Analysing the constructor and destructor of the camera class in listing 4.12, one can notice that the constructor opens a file descriptor for a device in a specified format, and the destructor ensures the file descriptor for the latter is closed, assuring one has a method of performing the first and last of the aforementioned steps. It's important to note that every device in Linux can be treated as a file, and, therefore, can be easily accessed through the `/dev` directory, as one can see in figure 4.4.1 in purple.

```
...
Camera::Camera(const std::string& device, const Camera::Format& format) {

        streaming = false;
        openDevice(device, format, true);
}
...
Camera::~Camera() {

        if (isOpen())
                closeDevice();
}
...
```

Listing 4.12: Camera Class: Constructor and Destructor

Class Method: openDevice

Regarding the openDevice function, firstly, one verifies if the device is already opened, if it is one proceeds to close it. This can be seen in listing 4.13, as well as the isOpen function.

```
Camera::Error Camera::openDevice(const std::string& dev_name, const Camera::Format& format,
    bool silent) {

        struct stat st;
        int descriptor;

        if (isOpen())
                closeDevice();
        ...
}
...
bool Camera::isOpen()  {
        return dev.descriptor != -1;
}
```

Listing 4.13: Camera Class: openDevice method (1)

Following, the method continues to verify if the specified device exists based on its name passed as an argument to this function (listing 4.14). Note that the function also receives a boolean variable that specifies if the logs should be suppressed or not. If the device does not exist then the function returns the appropriate error value.

```
    ...
    /* Verify the existence of the device */
    if (-1 == stat(dev_name.c_str(), &st)) {

        if (!silent) {
                log::error << "Cannot identify device '" << dev_name << "'" << std::endl;
                LOG_DETAILS();
        }
        return NO_DEVICE;
    }
    ...
```

Listing 4.14: Camera Class: openDevice method (2)

The verification of the existence of the file descriptor is performed by the Linux kernel function stat. This function (stat) returns information about a specific file in the buffer pointed by statbuf and with path pathname (listing 4.15). Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned and errno set to indicate the error, altering the variable of type struct stat (st in this case) passed as one of the parameters [27].

```
#include <sys/stat.h>
...
int stat(const char *restrict path, struct stat *restrict buf);
```

Listing 4.15: Linux: stat function declaration

As one can see in listing 4.16, the next step is to check the type of the device file descriptor, by using the stat header macro S_ISCHR, which evaluates to a non-zero value if the test is true, and zero if the test is false [27]. The value supplied to the macro is the value of st_mode from a stat structure (listing 4.17). If the file has the correct type then one continues and attempts to open the specified file representing the multimedia device and saves relevant device data.

```
    ...
    /* Check the device type */
    if (!S_ISCHR(st.st_mode)) {
        log::error << "Device '" << dev_name << "' is of the wrong type" << std::endl;
        LOG_DETAILS();

        return WRONG_DEVICE;
    }
    /* Attempt to open the device */
    descriptor = open(dev_name.c_str(), O_RDWR | O_NONBLOCK, 0);
    if (-1 == descriptor) {
        log::error << "Cannot open device ' " << dev_name << "'" << std::endl;
        LOG_DETAILS();
    }

        /* Keep relevant device data */
        this->dev.name = dev_name;
        this->dev.descriptor = descriptor;
```

```
        this->format = format;

        log::ok << "Device is open" << std::endl;

        initialize();

        return OK;
}
```

Listing 4.16: Camera Class: openDevice method (3)

```
struct stat {
   dev_t     st_dev;         /* ID of device containing file */
   ino_t     st_ino;         /* Inode number */
   mode_t    st_mode;        /* File type and mode */
   nlink_t   st_nlink;       /* Number of hard links */
   uid_t     st_uid;         /* User ID of owner */
   gid_t     st_gid;         /* Group ID of owner */
   dev_t     st_rdev;        /* Device ID (if special file) */
   off_t     st_size;        /* Total size, in bytes */
   blksize_t st_blksize;     /* Block size for filesystem I/O */
   blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

   struct timespec st_atim;  /* Time of last access */
   struct timespec st_mtim;  /* Time of last modification */
   struct timespec st_ctim;  /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

Listing 4.17: Linux: stat structure

Class Method: initialize

Since the openDevice method uses the initialize function, the latter will be described next. Primarily, in the initialize method, one starts by checking whether the file descriptor for the device is opened or not to proceed with the initialisation (listing 4.18).

```
Camera::Error Camera::initialize() {

        struct v4l2_capability cap;
        struct v4l2_cropcap cropcap;
        struct v4l2_crop crop;
        struct v4l2_format fmt;
        unsigned int min;

        /* Check if a device is open and ready to be initialized */
```

```
        if (!isOpen()) {
                log::error << "Device needs to be opened before initializing" << std::endl;
                return NO_DEVICE;
        }
    ...
}
```

Listing 4.18: Camera Class: initialize method (1)

The second part of the initialisation process (listing 4.20) is to identify the device capabilities by querying it. This is done utilising the function xioctl, which wraps the Linux kernel system call ioctl. The ioctl() system call manipulates the underlying device parameters of special files. In this case, xioctl performs continuous system calls (ioctl) until the result is success and errno is different than EINTR, as one can see in listing 4.19. Regarding the VIDIOC_QUERYCAP ioctl specifically, it is used to identify kernel devices compatible with V4L2 and to obtain information about driver and hardware capabilities [26].

```
#include <sys/ioctl.h>
...
static int xioctl(int fh, int request, void *arg)
{
    int r;
    do {
        r = ioctl(fh, request, arg);
    } while (-1 == r && EINTR == errno);
    return r;
}
```

Listing 4.19: xioctl function

```
    ...
    /* Query device capabilities */
    if (-1 == xioctl(this->dev.descriptor, VIDIOC_QUERYCAP, &cap)) {
        log::error << "Device capability query failed";

        if (EINVAL == errno)
                log::generic << " (it is not a V4L2 device)" << std::endl;

        LOG_DETAILS();

        return DEV_NOT_CAP;
    }
    ...
```

Listing 4.20: Camera Class: initialize method (2)

Concerning the listing 4.21, one checks if the device selected is capable of capturing video, if so one can proceed.

```
    ...
    /* Check if the selected device is one of video capture */
```

```
    if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) {
        log::error << "Not a video capture device" << std::endl;
        LOG_DETAILS();
        return DEV_NOT_CAP;
    }
    log::ok << "Device is capable of video capture" << std::endl;
    ...
```

Listing 4.21: Camera Class: initialize method (3)

The listing 4.22 refers to the part where the selection of the type of the capture buffer is done and where the cropping capabilities are analysed. As it's possible to observe, first one cleans the V4L2 cropcap structure (named cropcap), which gives the coordinates of the top left corner, width and height of the area which can be sampled (figure 4.4.2). The cropcap type is then set for video capture, and xioctl (listing 4.19) is used again to interface with the hardware for video capture and crop. The VIDIOC_CROPCAP ioctl is typically used to query the cropping limits, the pixel aspect of images and to calculate scale factors [26]. Moreover, VIDIOC_S_CROP ioctl is used for changing the cropping rectangle (figure 4.4.2).

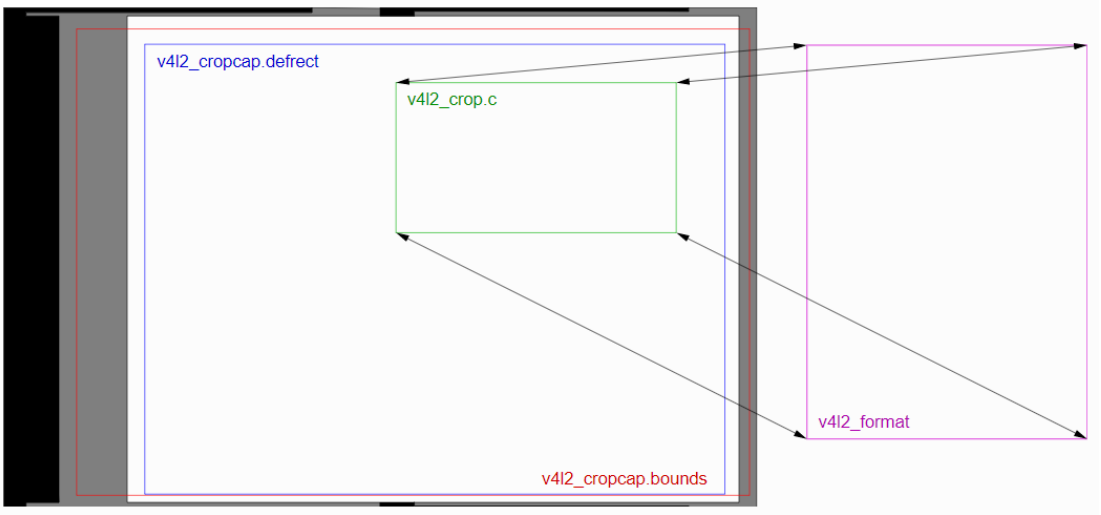


Figure 4.4.2: Video4Linux2 (V4L2): Image Cropping, Insertion and Scaling

```
    ...
    /* Select video capture buffer type and check cropping capabilities*/
    #define CLEAR(x) memset(&(x), 0, sizeof(x))/* not originally here */
    CLEAR(cropcap);
    cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    if (0 == xioctl(dev.descriptor, VIDIOC_CROPCAP, &cropcap)) {

        crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        crop.c = cropcap.defrect;

        if (-1 == xioctl(dev.descriptor, VIDIOC_S_CROP, &crop)) {
```

```
                log::warning << "Device not capable of video crop" << std::endl;
                LOG_DETAILS();
        }
    }
    ...
```

Listing 4.22: Camera Class: initialize method (4)

Listing 4.23 refers to the selection of the video format. Initially, one clears the V4L2 format structure (named fmt) specifies the format as video capture. The xioctl function is used to define the encoding for the video device and the return errors are handled. The VIDIOC_S_FMT ioctl is used to negotiate the format of data (typically image format) exchanged between the driver and the application [26]. Since the driver reads and adjusts the parameters against hardware despite the the hardware capabilities, it's a standard good practise to query for device capabilities (present in previous listings) before resorting to this ioctl.

```
    ...
    /* Select video format */
    CLEAR(fmt);
    fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

    fmt.fmt.pix.width = this->format.width;
    fmt.fmt.pix.height = this->format.height;
    fmt.fmt.pix.pixelformat = this->format.encoding;
    fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;

    if (-1 == xioctl(dev.descriptor, VIDIOC_S_FMT, &fmt)) {

        if (errno == EINVAL)
                log::error << "Unsupported encoding" << std::endl;
        else // errno = EBUSY
                log::error << "Device unavailable" << std::endl;

        LOG_DETAILS();
        return DEV_NOT_CAP;
    }
    ...
```

Listing 4.23: Camera Class: initialize method (5)

Finally, to end the initialize method, in listing 4.24 one performs some corrections to the driver and reserves memory space for video capture.

```
    ...
    /* Driver-related corrections */
    min = fmt.fmt.pix.width * 2;
    if (fmt.fmt.pix.bytesperline < min)
                fmt.fmt.pix.bytesperline = min;
    min = fmt.fmt.pix.bytesperline * fmt.fmt.pix.height;
    if (fmt.fmt.pix.sizeimage < min)
                fmt.fmt.pix.sizeimage = min;
```

```
9
    /* Reserve memory space */
    initializeMemoryMap();

    log::ok << "Device is initialized" << std::endl;
14
    return OK;
}
```

Listing 4.24: Camera Class: initialize method (6)

Class Method: initializeMemoryMap

As seen in the listing 4.24, the function initialize uses the initializeMemoryMap function. Due to that fact the latter will now be adressed. In listing 4.25, one starts by attempting to initialise the I/O mapped in memory for video capture and check for memory mapping support by the hardware, once again utilising the xioctl system call. Here specifically, one makes use of the VIDIOC_REQBUFS ioctl. This ioctl is used to initiate memory mapped, user pointers or DMABUF (DMA buffer sharing) based I/O. Memory-mapped buffers are located in the device's memory and must be allocated with this ioctl before they can be mapped into the application's address space [26].

```
Camera::Error Camera::initializeMemoryMap() {

        struct v4l2_requestbuffers req;
4       struct v4l2_buffer buf;

        /* Attempt to initiate memory mapped I/O */
        CLEAR(req);
        req.count = BUFFER_COUNT;
9       req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        req.memory = V4L2_MEMORY_MMAP;

        /* Check for memory mapping support */
        if (-1 == xioctl(dev.descriptor, VIDIOC_REQBUFS, &req)) {
14
                if (errno == EINVAL) {
                        log::error << "Memory mapping could not be initiated: " \
                            "no such capability on this device" << std::endl;
                        LOG_DETAILS();
19                      return DEV_NOT_CAP;

                } else {
                        log::error << "Memory mapping could not be initiated" << std::endl;
                        LOG_DETAILS();
24                      return DEV_NOT_CAP;
                }
        }
    ...
```

Listing 4.25: Camera Class: initializeMemoryMap method (1)

Additionally, in listing 4.26, one created mappings for each buffer in memory, but firstly queried the status of the buffer each iteration with the VIDIOC_QUERYBUF ioctl. This ioctl is part of the streaming I/O method, and it can be used to query the status of a buffer at any time after buffers have been allocated with the VIDIOC_REQBUFS ioctl, as seen in listing 4.25. The memory map creation was performed using the Linux kernel function mmap (listing 4.27). The latter creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr and the length argument specifies the length of the mapping (which must be greater than 0) [27] [28] (figure 4.4.3).

```
    ...
    /* Create mappings for each buffer in memory */
        for (int n_buffers = 0; n_buffers < req.count; ++n_buffers) {

                /* Query the status of the buffer */
                CLEAR(buf);
                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_MMAP;
                buf.index = n_buffers;

                if (-1 == xioctl(dev.descriptor, VIDIOC_QUERYBUF, &buf)) {
                        log::error << "Buffer unavailable" << std::endl;
                        LOG_DETAILS();
                        return RESOURCE_UNAVAIL;
                }

                buffers[n_buffers].length = buf.length;
                buffers[n_buffers].start =
                        mmap(NULL /* start anywhere */,
                                        buf.length,
                                        PROT_READ | PROT_WRITE /* required */,
                                        MAP_SHARED /* recommended */,
                                        dev.descriptor, buf.m.offset);

                if (MAP_FAILED == buffers[n_buffers].start) {
                        log::error << "Failed to map buffer" << n_buffers << \
                                        " to a space in memory" << std::endl;
                        LOG_DETAILS();
                        return RESOURCE_UNAVAIL;
                }
        }
        log::ok << "Created and mapped " << BUFFER_COUNT << " buffers" << std::endl;
        return OK;
}
```

Listing 4.26: Camera Class: initializeMemoryMap method (2)

```
#include <sys/mman.h>
...
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Listing 4.27: Linux: mmap function

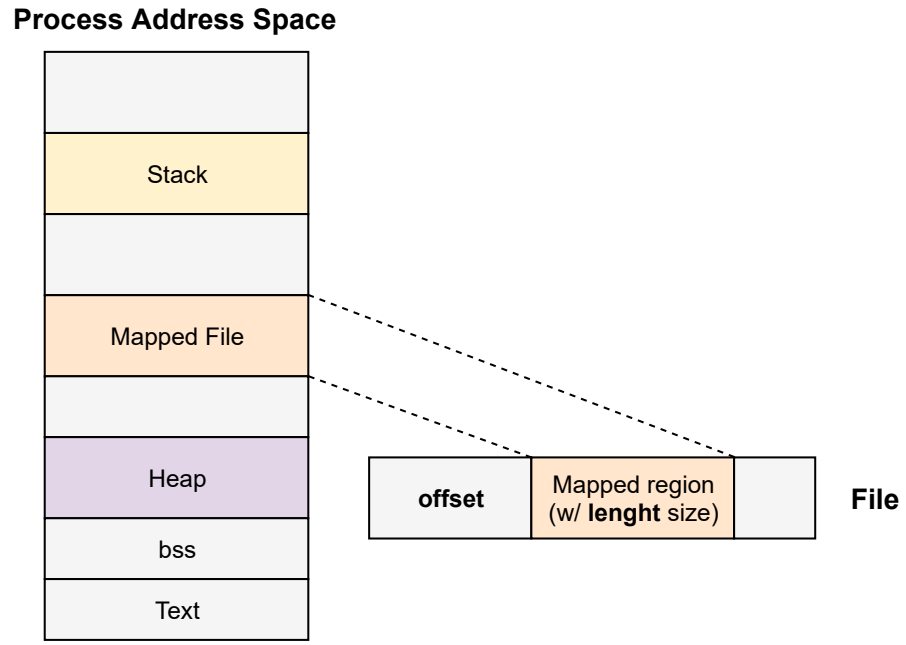


Figure 4.4.3: Mapping a file into a process's address space

Class Method: closeDevice

As stated before, the camera class also has a method for closing a file descriptor, represented in listing 4.28. The method verifies if the device is open before proceeding with its closure. After, the device is close and uninitialised recurring to the class method uninitialize.

```
void Camera::closeDevice()  {

        if (isOpen()) {
                close(this->dev.descriptor);
                this->dev.name = "";
                this->dev.descriptor = -1;
                uninitialize();
        }

        log::ok << "Device closed" << std::endl;
}
```

Listing 4.28: Camera Class: closeDevice method

Class Method: uninitialize

As a standard practice, every method that initialises something must have a counterpart method for uninitialising. The uninitialize method of the camera class is portrayed in listing 4.29. As one would expect the method relies on the Linux kernel function munmap to unmap the previously allocated memory for the buffers. munmap removes any

mappings for entire pages containing any part of the address space of the process starting at addr and continuing for len bytes [27] (listing 4.30).

```
Camera::Error Camera::uninitialize() {

        for (uint32_t i = 0; i < BUFFER_COUNT; i++) {
                if (-1 == munmap(buffers[i].start, buffers[i].length)) {
                        log::error << "Failed to unmap buffer " << i << \
                                        "; possibly not mapped" << std::endl;
                        LOG_DETAILS();
                        return BAD_UNMAP;
                }
        }

        log::ok << "Device uninitialized" << std::endl;

        return OK;
}
```

Listing 4.29: Camera Class: uninitialize method

```
#include <sys/mman.h>
...
int munmap(void *addr, size_t len);
```

Listing 4.30: Linux: munmap function

Class Method: start

The method start initiates the camera streaming. Note that the device was preemptively prepared for buffer handling, this was explained throughout the previous listings. When one refers to the frame capture process, one or more buffers have to be submitted to the device in question through an enqueue. Once again one makes use of system calls (xioctl) to communicate with the camera, as it is possible to see in listing 4.31. Applications call the VIDIOC_QBUF ioctl to enqueue an empty (capturing) or filled (output) buffer in the driver's incoming queue [26].

```
Camera::Error Camera::start() {

        unsigned int i;
        enum v4l2_buf_type type;
        struct v4l2_buffer buf;

        /* Enqueue buffers for capture */
        for (i = 0; i < BUFFER_COUNT; i++) {

                CLEAR(buf);
                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_MMAP;
                buf.index = i;
```

```
                if (-1 == xioctl(dev.descriptor, VIDIOC_QBUF, &buf)) {
                        log::error << "Failed to enqueue buffer " << i << \
                                        " for capture" << std::endl;
                        LOG_DETAILS();
                        return UNDEF;
                }
        }
        ...
```

Listing 4.31: Camera Class: start method (1)

Still within the start method, the next step is to attempt streaming by calling VIDIOC_STREAMON ioctl, as mentioned wrapped in xioctl. Capture hardware is disabled and no input buffers are filled (if there are any empty buffers in the incoming queue) until VIDIOC_STREAMON has been called.

```
        ...
        /* Attempt to start streaming */
        type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        if (-1 == xioctl(dev.descriptor, VIDIOC_STREAMON, &type)) {
                log::error << "Could not start stream" << std::endl;
                LOG_DETAILS();

                /* At this point, none of the well-defined errors may apply */
                return UNDEF;
        }

        log::ok << "Started streaming" << std::endl;

        streaming = true;
}
```

Listing 4.32: Camera Class: start method (2)

Class Method: stop

Similarly to the start method, the stop method stops the streaming, but calls the VIDIOC_STREAMOFF ioctl to do so. This ioctl, removes all buffers from the incoming and outgoing queues, which means that all images captured but not dequeued yet will be lost [26].

```
Camera::Error Camera::stop() {

        struct v4l2_buffer buf;

        /* Signal if no stream was running */
        if (!streaming)
                log::info << "No capture is currently active" << std::endl;

        /* Stop video stream */
        CLEAR(buf);
```

```
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        xioctl(dev.descriptor, VIDIOC_STREAMOFF, &buf);

        streaming = false;

        log::ok << "Stopped streaming" << std::endl;

        return OK;
}
```

Listing 4.33: Camera Class: stop method

Class Method: capture

As the name indicates, the capture method tries to capture a frame of the camera video stream within a certain time, smaller than the timeout specified. In listing 4.34 is possible to visualise that the method checks for an opened device to begin the capture. The I/O descriptor sets are stored as bit fields in arrays of integers, the macros used in listing 4.34 (FD_*) manipulate such descriptor sets. On the one hand, the FD_ZERO one initialises a descriptor set (in this case the descriptor named fds) to the null set. On the other hand, the FD_SET macro includes a particular descriptor into the set passed as a parameter. Following, the intended timeout is set and the system call select is used. This function allows a program to monitor multiple file descriptors (in a set), waiting until one or more of the file descriptors become "ready" for some class of I/O operation (a read in this case). Using the select function assures that the system call to perform doesn't stay in a blocking state waiting for data to read. This is advantageous since one is able to read frames from the camera's stream in an unblocking manner.

```
Camera::Error Camera::capture(std::string& filename) {

        fd_set fds;
        struct timeval tv;
        int32_t result;
        struct v4l2_buffer buf;

        if(!isOpen()) {
                log::error << "Device must be opened before capturing" << std::endl;
                return DEV_NOT_OPEN;
        }

        FD_ZERO(&fds);
        FD_SET(dev.descriptor, &fds);

        /*Wait for the descriptor to be ready for reading until timeout*/
        tv.tv_sec = 0;
        tv.tv_usec = FETCH_TIMEOUT*1000;
        result = select(dev.descriptor + 1, &fds, NULL, NULL, &tv);
        ...
```

Listing 4.34: Camera Class: capture method (1)

The listing 4.35 performs some error and timeout handling and proceeds to read a frame from the buffer.

```
        ...
        /* Error */
        if (result == -1) {
                if (errno == EINTR) {
                        log::error << "Failed to capture frame" << std::endl;
                        log::details << "Descriptor unavailable" << std::endl;
                        return RESOURCE_UNAVAIL;
                }

        /* Timeout */
        } else if (result == 0) {
                log::error << "Timed out while trying to capture frame" << std::endl;
                return CAPTURE_TIMEOUT;
        }

        /* Read the frame and return if failed */
        result = (int32_t)readFrame(buf, filename);

        if (result != OK)
                return static_cast<Camera::Error>(result);

        return OK;
}
```

Listing 4.35: Camera Class: capture method (2)

Class Method: readFrame

The method readFrame is utilised in the frame capture function. In listing 4.36, one makes an attempt at dequeuing the buffer using the VIDIOC_DQBUF ioctl and handles the result. The application call this ioctl to dequeue a filled (capturing) buffer from the driver's outgoing queue [26].

```
Camera::Error Camera::readFrame (v4l2_buffer& buf, std::string filename) {

        ofstream out_file;

        Error return_val = OK;

        /* Read frame */
        CLEAR(buf);
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;

        /* Attempt to dequeue buffer */
        if (-1 == xioctl(dev.descriptor, VIDIOC_DQBUF, &buf)) {

                log::error << "Failed to capture frame" << std::endl;
```

```
17                switch (errno) {
                  case EAGAIN:
                          log::details << "No buffer in the outgoing queue" << std::endl;
                          return NO_BUF_OUT;

22                case EIO:
                          log::details << "Internal error" << std::endl;
                          /* TODO: Implement buffer recovery procedure (see spec)*/
                          return INTERNAL;

27                default:
                          log::details << "Undefined error" << std::endl;
                          return UNDEF;
                  }
          }
32
          log::ok << "Read frame" << std::endl;
          ...
```

Listing 4.36: Camera Class: readFrame method (1)

In listing 4.37, one opens the output file stream to which the image captured will be written.

```
1     ...
      out_file.open(filename, ios::out | ios::trunc | ios::binary);

      if (!out_file.is_open()) {
          log::error << "Could not open file as requested " << std::endl;
6         return_val = RESOURCE_UNAVAIL;
          goto cleanup;
      }

      log::ok << "Opened " << filename << " for writing" << std::endl;
11    ...
```

Listing 4.37: Camera Class: readFrame method (2)

Lastly, in listing 4.38, one performs the aforementioned write, saving a certain frame. The method then continues to cleanup, closing the output stream and setting up the filed for the next frame with the VIDIOC_QBUF ioctl.

```
      ...
      out_file.write(static_cast<char*>(buffers[buf.index].start), buf.bytesused);

4         if (!out_file.good()) {
                  log::error << "An error occured while writing to file " << std::endl;
                  goto cleanup;
          }

9         log::ok << "Saved " << filename << std::endl;

          cleanup:
```

```
        out_file.close();

        if (-1 == xioctl(dev.descriptor, VIDIOC_QBUF, &buf)) {
                log::error << "Failed to enqueue buffer for capture" << std::endl;
                LOG_DETAILS();
                return UNDEF;
        }

        return return_val;
}
```

Listing 4.38: Camera Class: readFrame method (3)

Class Method: drawRectangle

The method represented in listing 4.39 draws the detection boxes on a specific frame based on its parameters using imagemagick functionalities.

```
Camera::Error Camera::drawRectangle(std::string input_file,
                std::string output_file, const Camera::Point2D& start,
                const Camera::Point2D& end, const Camera::Color& fill,
                const Camera::Color& stroke) {

        /* Command format:
                convert [input_file] -strokewidth 2 -stroke "[stroke color code]"
                -fill "[fill color code]" -draw "rectangle [start.x],[start.y]
                [end.x],[end.y] " [output_file]*/

        std::stringstream command;
        std::stringstream stroke_color_code;
        std::stringstream fill_color_code;
        std::stringstream rectangle;


        stroke_color_code << "rgba(" << (uint32_t)stroke.r << ',' << \
        (uint32_t)stroke.g << ',' << (uint32_t)stroke.b << ',' << \
        (uint32_t)stroke.a << ")";

        fill_color_code << "rgba(" << (uint32_t)fill.r << ',' << (uint32_t)fill.g \
        << ',' << (uint32_t)fill.b << ',' << (uint32_t)fill.a << ")";

        rectangle << "rectangle " << start.x << ',' << start.y << ' ' << end.x << \
        ',' << end.y;

        command << "convert \"" << input_file << "\" -strokewidth 2 -stroke \"" << \
        stroke_color_code.str() << "\" -fill \"" << fill_color_code.str() << \
        "\" -draw \"" << rectangle.str() << "\" \"" << output_file << "\"";

        log::info << "Running [" << command.str() << ']' << std::endl;
```

```
33          system(command.str().c_str());
}
```

Listing 4.39: Camera Class: drawRectangle method

Class Method: drawText

Additionally, the method in listing 4.40, draws the detection labels over the boxes with their correspondent percentages coming as a result of running inference.

```
1  Camera::Error Camera::drawText(std::string input_file, std::string output_file,
                        std::string text, const Camera::Point2D& start, uint32_t thickness,
                        uint32_t size, const Camera::Color& stroke) {

        /* Command format:
6               convert [input_file] -gravity Northwest -weight [thickness] -pointsize
                [size] -annotate +[start.x]+[start.y] "[text]" [output_file]    */

        std::stringstream command;

11      command << "convert \"" << input_file << "\" -gravity Northwest -weight " \
        << thickness << " -pointsize " << size << " -annotate +" << start.x << '+' \
        << start.y << " \"" << text << "\" \"" << output_file << "\"";

        log::info << "Running [" << command.str() << ']' << std::endl;
16
        system(command.str().c_str());
}
```

Listing 4.40: Camera Class: drawText method

4.5 Threads

For the purpose of setting up all the system threads, one devised a module for thread operations. The header file for the module can be observed in listing C.3.

4.5.1 Threads: createThreads

The listing 4.41 represents the beginning of function used for thread creation. Initially, one starts by dynamically allocating memory for the new thread object with new, logging the attempt to create the latter resorting to the log module (C.2). Additionally, the intended name for the thread is assigned to the Thread object's name (listing 4.41).

```
const Thread* createThread (const Thread::Config& config) {

    /* Dinamically allocate memory for the Thread object */
    Thread* thread = new Thread;

    log::info << "Creating thread " << config.name << std::endl;

    thread->name = config.name;
    ...
```

Listing 4.41: Threads: createThread (1)

In listing 4.42, one initializes the thread object's attributes (named native_attr) with the default ones through the function pthread_attr_init. After, one gets the current scheduler policy with pthread_attr_getschedpolicy and the current scheduler parameters with pthread_attr_getschedparam. This was important to check for the policy and parameters in use.

```
    ...
    /* Initialize attr object with the default attributes */
    pthread_attr_init(&(thread->native_attr));

    /* Get current scheduler's policy */
    pthread_attr_getschedpolicy (&(thread->native_attr), &(thread->thread_policy));

    /* Get current scheduler's parameters */
    pthread_attr_getschedparam (&(thread->native_attr), &(thread->thread_param));
    ...
```

Listing 4.42: Threads: createThread (2)

Following, in listing 4.43, one set the scheduler policy to SCHED_RR (round-robin with a certain timeslice) calling the function pthread_attr_setschedpolicy, passing the thread object's attributes as an argument as well. In the same listing it is also possible to observe that the maximum and minimum priorities were assessed to ensure the priority enum (Priority, under 4.44) defined the priority levels accordingly. The desired priority as then set the value passed in the parameter Config struct (named config).

```
    ...
    /* Set current scheduler's policy */
```

```
    pthread_attr_setschedpolicy (&(thread->native_attr), SCHED_RR);

    /* Get current scheduler's max and min priorities */
    int rr_min_priority = sched_get_priority_min (SCHED_RR);
    int rr_max_priority = sched_get_priority_max (SCHED_RR);

    /* Define the real priority as the desired priority */
    thread->thread_param.sched_priority = config.priority;
    ...
```

Listing 4.43: Threads: createThread (3)

```
typedef enum Priority_t {
    IDLE = 0,
    LOW = 15,
    BELOW_NORMAL = 30,
    NORMAL = 45,
    ABOVE_NORMAL = 60,
    HIGH = 75,
    REAL_TIME = 90
} Priority;
```

Listing 4.44: Threads: Priority enum

In terms of the listing 4.45, one set the scheduler parameters again to ensure the priority change would take effect and then used the aforementioned get function (pthread_attr_getschedparam to retrieve and verify if the change took place.

```
    ...
    /* Request to attribute the desired priority value to the attributes
    structure and then get the real value */
    pthread_attr_setschedparam (&(thread->native_attr), &(thread->thread_param));
    pthread_attr_getschedparam(&(thread->native_attr), &(thread->thread_param));

    /* Check if priority change request failed */
    if (thread->thread_param.sched_priority != config.priority) {
        log::warning << "Prority change request failed" << std::endl;
    }
    ...
```

Listing 4.45: Threads: createThread (4)

Finally, under the listing 4.46, one started the newly created thread with pthread_create and defined its main routine as the one passed in the config structure. Some error handling is then done to ensure the thread was correctly set up and the its creation is logged resorting, once again, to the log module (C.2).

```
  ...
   thread->thread_status = pthread_create(&(thread->native), &(thread->native_attr), config
       .start_routine, (void*)config.args);
```

```
4       if(thread->thread_status)
        {
            if(thread->thread_status==EPERM)
                log::error << "Thread creation returned EPERM" << std::endl;
            else if(thread->thread_status==EINVAL)
9               log::error << "Thread creation returned EINVAL" << std::endl;
            else
                log::error << "Thread creation returned neither EPERM nor EINVAL" << std::endl;
                    exit(1);
          }
14
        log::ok << "Created thread " << thread->name << std::endl;
        log::details << "Priority: " << thread->thread_param.sched_priority << std::endl;

        return thread;
19  }
```

Listing 4.46: Threads: createThread (5)

4.5.2 Threads: deleteThreads

The listing 4.47 represents the function used for thread deletion. As its possible to see, the pthread attribute is destroyed with the function pthread_attr_destroy and the memory for the thread object is freed using delete.

```
1   int32_t deleteThread (Thread& thread) {

            int32_t status;

            /* Destroy attr object */
6           status = pthread_attr_destroy(&(thread.native_attr));

            /* Free objects memory */
            delete &thread;

11      return status;
    }
```

Listing 4.47: Threads: deleteThread

4.5.3 Computational Unit: Timeline Structure

Taking into consideration the version of the computational unit's thread timeline, referred in section 3.5.6, one devised a refined version of the latter based on the implementation achieved (figure 4.5.1).

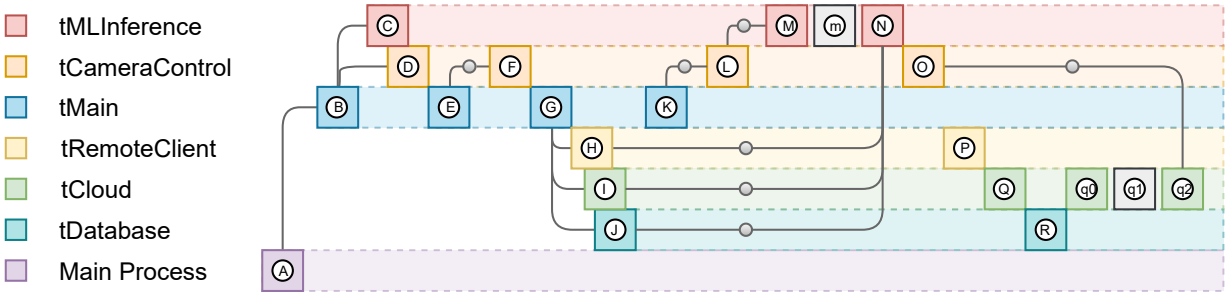


Figure 4.5.1: Computational Unit Updated Thread Scheme (Aug. in Appendix D.2)

In the table 4.2 are described all the letter-marked events of the computational unit's thread timeline of figure 4.5.1.

Event ID	Event Description	Event ID	Event Description
Ⓐ	Main Process creates tMain	Ⓛ	Capture frames and signal cCaptureDone when done
Ⓑ	tMain creates tMLInference and tCameraControl	Ⓜ	tMLInference initiates the Python interpreter
Ⓒ	tMLInference awaits cCaptureDone cond. signal	ⓜ	Run Inference (Python Script) outputting result to file
Ⓓ	tCameraControl inits the camera for frame capture	Ⓝ	Draw boxes and labels on the frames (based on file) and signal threat found
Ⓔ	tMain awaits cCameraInit cond. signal	Ⓞ	tCameraControl awaits cCloudFinished cond.signal
Ⓕ	Signal cCameraInit sent and wait for cThreadsCreated cond. signal	Ⓟ	tRemoteClient creates a detached daughter thread for sending notifications
Ⓖ	tMain creates tDatabase, tCloud and tRemoteClient	Ⓠ	tCloud creates a detached daughter thread for frame submission
Ⓗ	tRemoteClient awaits cNotification cond. signal	ⓠ0	d_tCloud (tCloud daughter) initiates the Python interpreter
Ⓘ	tCloud awaits cThreatFoundCloud cond. signal	ⓠ1	Run Python script for sending frames to Google Drive folder
Ⓙ	tDatabase awaits cThreatFoundDB cond. signal	ⓠ2	d_tCloud signals cCloudFinished when done
Ⓚ	tMain signals cThreadsCreated	Ⓡ	tDatabase creates a detached daughter thread for database update

Table 4.2: Computational Unit Timeline Events

In figure 4.5.2, it is possible to see all the condition variable used to communicate between threads.

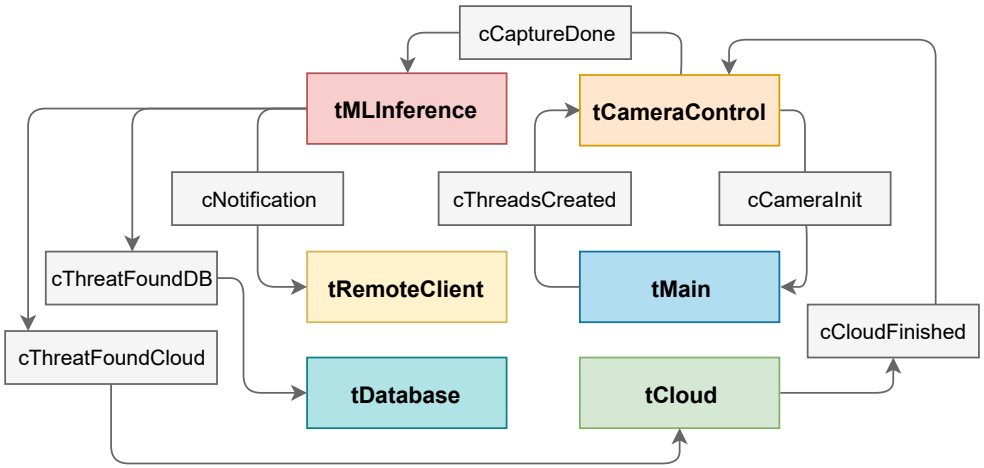


Figure 4.5.2: Computational Unit Condition Variable Synchronisation

The code for the static allocation of the aforementioned condition variables and also the mutexes associated with them is represented in listing 4.48. Note that a pthread_cond_wait() call dynamically binds a certain mutex to a condition variable. The use of more than one mutex for concurrent pthread_cond_wait() calls, for example, in different threads may lead to undefined results. Considering this, the number of mutexes was greater than anticipated, so one could follow this rule. On the one hand, the initialisation of a statically allocated condition variable is done by assigning the value PTHREAD_COND_INITIALIZER. On the other hand, the initialisation of a statically allocated mutex is done by assigning it the value of PTHREAD_MUTEX_INITIALIZER. These synchronisation objects could also be dynamically allocated, but since the application didn't require it one opted for the static allocation.

```
...
/* Statically allocated condition variables */
static pthread_cond_t cCameraInit = PTHREAD_COND_INITIALIZER;
static pthread_cond_t cThreatFoundDB = PTHREAD_COND_INITIALIZER;
static pthread_cond_t cThreatFoundCloud = PTHREAD_COND_INITIALIZER;
static pthread_cond_t cCaptureDone = PTHREAD_COND_INITIALIZER;
static pthread_cond_t cThreadsCreated = PTHREAD_COND_INITIALIZER;
static pthread_cond_t cNotification = PTHREAD_COND_INITIALIZER;
static pthread_cond_t cCloudFinished = PTHREAD_COND_INITIALIZER;

/* Statically allocated mutexes */
static pthread_mutex_t mCamera = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t mNotifyCache = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t mFrameCache = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t mFrameCacheCam = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t mMetaCacheDB = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t mMetaCacheCloud = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t mCameraWait = PTHREAD_MUTEX_INITIALIZER;
...
```

Listing 4.48: Static allocation of the condition variables and mutexes

Firstly, in event Ⓐ, one can clearly observe that the ARGOS' thread scheme is originated from the main process (listing 4.49). All the threads created in the application use the threads module devised and explained in section 4.5. Additionally, one can also see that the routine for the tMain thread is mainHandler.

```
int main(int argc, char** argv) {

    /* perror is thread-safe */
    perror("[ARGOS] Starting");

    Thread::Config main;
    Thread* main_thread;

    main.name = "tMain";
    main.priority = LOW;
    main.args = NULL;
    main.start_routine = mainHandler;

    /* tMain thread creation */
    main_thread = const_cast<Thread*>(createThread(main));

    while (true){}

    perror("[ARGOS] Shutting Down");
    return 0;
}
```

Listing 4.49: Main process creates the tMain thread

As aforementioned, and referring to the event Ⓑ, the handler for tMain (mainHandler in listing 4.50), starts by creating the tMLInference and tCameraControl threads, designed specifically for running inference and controlling the system's camera, respectively.

```
static void* mainHandler(void* arg){
    perror("[ARGOS] Starting tMain");

    /* Inference thread setup */
    Thread::Config inference;
    Thread* inference_thread;
    inference.name = "tMLInference";
    inference.priority = HIGH;
    inference.args = NULL;
    inference.start_routine = inferenceHandler;

    /* Inference thread creation */
    inference_thread = const_cast<Thread*>(createThread(inference));

    /* Camera control thread setup */
    Thread::Config camera_ctrl;
    Thread* cam_ctrl_thread;
    camera_ctrl.name = "tCameraControl";
```

```
    camera_ctrl.priority = REAL_TIME;
    camera_ctrl.args = NULL;
    camera_ctrl.start_routine = cameraHandler;

    /* Camera control thread creation */
    cam_ctrl_thread = const_cast<Thread*>(createThread(camera_ctrl));

    /* Attempt to connect the network */
    ...
```

Listing 4.50: Thread tMain creates the threads tMLInference and tCameraControl

In event Ⓒ in listing 4.51, one can see that, as stated before, the tMLInference thread (with routine inferenceHandler) awaits in pthread_cond_wait for the cCaptureDone condition variable to be signalled. There is also the option of calling pthread_cond_timedwait instead if one wants to specify a certain time to leave the blocking state if the condition signal isn't received.

```
static void* inferenceHandler(void* arg){
    perror("[ARGOS] Starting tMLInference");

    std::string filename = root_dir + " ... /argos_tflite_detection_image.py";
    bool timed = false;

    /* Transform the directory string into a C string */
    const char* _filename = filename.c_str();

    #define MAX_WAIT_TIME_IN_SECONDS 10

    struct timespec max_wait = {0, 0};
    clock_gettime(CLOCK_REALTIME, &max_wait);
    max_wait.tv_sec += MAX_WAIT_TIME_IN_SECONDS;

    FILE* file = fopen(_filename, "r");

    while(true){

        /* Blocks on frame cache mutex waiting for the camera captures */
        if(pthread_mutex_lock(&mFrameCache) > 0)
                perror("[tMLInference] pthread_mutex_lock failed on mFrameCache");
        while (!capture_done){
                /* Automatically unlocks mutex when signalled */
                if(timed)
                    if (pthread_cond_timedwait(&cCaptureDone, &mFrameCache, &max_wait) > 0)
                                perror("[tMLInference] pthread_cond_timedwait failed on
                                    cCaptureDone");
                else
                if (pthread_cond_wait(&cCaptureDone, &mFrameCache) > 0)
                                perror("[tMLInference] pthread_cond_wait failed on
                                    cCaptureDone");
        }
```

```
        if(pthread_mutex_unlock(&mFrameCache) > 0)
33              perror("[tMLInference] pthread_mutex_unlock failed on mFrameCache");

        capture_done = false;
        ...
```

Listing 4.51: Thread tMLInference awaits cCaptureDone condition signal

In event Ⓓ the tCameraControl thread routine starts, the latter proceeds in specifying the device to open, in this case `/dev/video0`, and prepares the camera for capturing frames (more in depth in section 4.4.1).

```
static void* cameraHandler(void* arg){
    perror("[ARGOS] Starting tCameraControl");

4   /* Init I2C and light sensor */

    std::string device = "/dev/video0";
    /* Prepare the camera */
    if (cam.openDevice(device, (Camera::Format){640, 480, \
9               Camera::Encoding::MJPEG}) != Camera::Error::OK ) {
        cam.closeDevice();
        return (void*)1;
    }
    ...
```

Listing 4.52: Thread tCameraControl inits the camera for frame capture

In event Ⓔ, after creating the aforementioned threads, tMain locks the mCamera in order to check the state of the camera_init variable and waits for the camera to finish initialising so it can receive the cCameraInit condition signal. Note that the pthread_cond_wait() call releases the mutex passed as parameter when signalled. A condition variable is always used in conjunction with a mutex. The mutex provides mutual exclusion for accessing the shared variable, while the condition variable is used to signal changes in the variable's state [29]. In this case the shared variable is camera_init, which is a bool that informs if the camera is already initialised. This is depicted in listing 4.53.

```
    ...
2   /* Blocks on camera mutex waiting for the camera init */
    pthread_mutex_lock(&mCamera);
    while (!camera_init){
        /* Automatically unlocks mutex when signalled */
        if (pthread_cond_wait(&cCameraInit, &mCamera) > 0)
7           perror("[tMain] pthread_cond_wait failed on cCameraInit");
    }
    ...
```

Listing 4.53: Thread tMain awaits cCameraInit condition signal

In event Ⓕ, when the camera is initialised, a condition signal is sent to the thread waiting in the cCameraInit (tMain in this case). Moreover, the camera thread waits on the condition variable cThreadsCreated for tMain to finish

creating the last three threads, related to the remote client, cloud and database. This can be observed in listing 4.54.

```
    ...
    camera_init = true;

    /* Signal the condition variable */
    if (pthread_cond_signal(&cCameraInit) > 0)
        perror("[tCameraControl] pthread_cond_signal failed on cCameraInit");

    /* Blocks on camera wait mutex waiting for tMain to create the other threads */
    pthread_mutex_lock(&mCameraWait);
    while (!threads_created){
        /* Automatically unlocks mutex when signalled */
        if (pthread_cond_wait(&cThreadsCreated, &mCameraWait) > 0)
            perror("[tCameraControl] pthread_cond_wait failed on cThreadsCreated");
    }
    ...
```

Listing 4.54: Thread tCameraControl signals cCameraInit and waits for cThreadsCreated condition signal

In event Ⓖ, after receiving the condition signal from tCameraControl on cCameraInit, the tMain thread continues and creates the tRemoteClient, tCloud, and tDatabase threads, as illustrated in listing 4.55.

```
    ...
    /* Remote client thread setup */
    Thread::Config remote_client;
    Thread* remote_client_thread;
    remote_client.name = "tRemoteClient";
    remote_client.priority = ABOVE_NORMAL;
    remote_client.args = NULL;
    remote_client.start_routine = rclientHandler;

    /* Remote client thread creation */
    remote_client_thread = const_cast<Thread*>(createThread(remote_client));

    /* Cloud thread setup */
    Thread::Config cloud;
    Thread* cloud_thread;
    cloud.name = "tCloud";
    cloud.priority = NORMAL;
    cloud.args = NULL;
    cloud.start_routine = cloudHandler;

    /* Cloud thread creation */
    cloud_thread = const_cast<Thread*>(createThread(cloud));

    /* Database thread setup */
    Thread::Config database;
    Thread* database_thread;
    database.name = "tDatabase";
    database.priority = BELOW_NORMAL;
```

```
    database.args = NULL;
    database.start_routine = databaseHandler;

    /* Database thread creation */
    database_thread = const_cast<Thread*>(createThread(database));
    ...
```

Listing 4.55: Thread tCameraControl signals cCameraInit

In event (H), presented in listing 4.56, the tRemoteClient thread awaits on the condition variable cNotification (associated with the mNotifyCache mutex) for the signal that indicates that a threat was found and that it needs to send a notification to warn the user.

```
static void* rclientHandler(void* arg){
    perror("[ARGOS] Starting tRemoteClient");

    /* Listen for connection */

    while(true){
        /* Blocks on notify cache mutex waiting for a threat detection */
        if(pthread_mutex_lock(&mNotifyCache) > 0)
                perror("[tRemoteClient] pthread_mutex_lock failed on mNotifyCache");
        while (!notify_rc){
                /* Automatically unlocks mutex when signalled */
                if (pthread_cond_wait(&cNotification, &mNotifyCache) > 0)
                        perror("[tRemoteClient] pthread_cond_wait failed on cNotification");
        }
        notify_rc = false;
        ...
```

Listing 4.56: Thread tRemoteClient awaits cNotification condition signal

The event (I), represents the tCloud thread wait for a condition signal indicating some threat was found (listing 4.57). The connection to the cloud done later in the daughter thread launched by tCloud.

```
static void* cloudHandler(void* arg){
    perror("[ARGOS] Starting tCloud");
    while(true){
        /* Blocks on metadata cache mutex waiting for a threat detection */
        if(pthread_mutex_lock(&mMetaCacheCloud) > 0)
                perror("[tCloud] pthread_mutex_lock failed on mMetaCacheCloud");
        while (!threat_found){
                /* Automatically unlocks mutex when signalled */
                if (pthread_cond_wait(&cThreatFoundCloud, &mMetaCacheCloud) > 0)
                        perror("[tCloud] pthread_cond_wait failed on cThreatFoundCloud");
        }
        thread_unblocked[0] = true;
        ...
```

Listing 4.57: Thread tCloud awaits cThreatFoundCloud condition signal

Additionally, the event (J), depicts the tDatabase thread wait for the condition signal that represents the identification of a threat (listing 4.58).

```
static void* databaseHandler(void* arg){
    perror("[ARGOS] Starting tDatabase");

    /* Request connection */

    while(true){
        /* Blocks on metadata cache mutex waiting for a threat detection */
        if(pthread_mutex_lock(&mMetaCacheDB) > 0)
                perror("[tDatabase] pthread_mutex_lock failed on mMetaCacheDB");
        while (!threat_found){
                /* Automatically unlocks mutex when signalled */
                if (pthread_cond_wait(&cThreatFoundDB, &mMetaCacheDB) > 0)
                        perror("[tDatabase] pthread_cond_wait failed on cThreatFoundDB");
        }
        thread_unblocked[1] = true;
        ...
```

Listing 4.58: Thread tDatabase awaits cThreatFoundDB condition signal

The event marked in (K) represents the stage where the tMain thread sends the condition signal to the threads waiting for it on the condition variable cThreadsCreated, in this case this thread is tCameraControl. The signalling is depicted in listing 4.59.

```
    ...
    threads_created = true;
    /* Signal the condition variable */
    if (pthread_cond_signal(&cThreadsCreated) > 0)
        perror("[tMain] pthread_cond_signal failed on cThreadsCreated");
    ...
```

Listing 4.59: Thread tMain signals cThreadsCreated

When all the threads are created, in event (L) the tCameraControl thread starts capturing frames from the camera's stream and signals the cCaptureDone condition variable. The thread waiting for this signal is tMLInference. The listing 4.60 represents the aforementioned event.

```
    ...
    /* Start Camera */
    cam.start();

    while(true){

        if(warm_start){...}
        warm_start = true;

        /* Sample frames periodically */
        for (uint32_t i = 0; i < FRAME_GRAB; i++) {
```

```
            sleep(1);
            capture_str[28] = (char)(i+0x30);
            cam.capture(capture_str);
        }

        capture_done = true;
        /* Signal the condition variable */
        if (pthread_cond_signal(&cCaptureDone) > 0)
                perror("[tCameraControl] pthread_cond_signal failed on cCaptureDone");


        /* Adjust camera exposure */
    }
    cam.stop();
    cam.closeDevice();

    pthread_exit(NULL);
}
```

Listing 4.60: Thread tCameraControl starts capturing frames and signals cCaptureDone when done

In the event Ⓜ, the tMLInference initiates the Python interpreter (Py_Initialize()) necessary for running the inference script on the last captured frames. Following, in event ⓜ, one runs the inference script which outputs the inference results to a text file. The opening of the file is done in a standard C manner but the execution of the script relies on the call of the PyRun_SimpleFile() function. Finally the interpreter is closed (Py_Finalize()) and the thread proceeds. One used python in this step to facilitate the development and accelerate the system's prototyping [30]. The event is represented in listing 4.61.

```
        /* Requires */
        #include <Python.h>
        ...
        /* Initialize the Python interpreter */
        Py_Initialize();

        PyRun_SimpleString("import sys");
        PyRun_SimpleString("import os");
        PyRun_SimpleString("print('[tMLInference] Executing Inference Python Script')");

        /* Run Inference Script */
        if(file != NULL)
            PyRun_SimpleFile(file, _filename);

        /* Close the Python Interpreter */
        Py_Finalize();
        ...
```

Listing 4.61: Thread tMLInfernce initiates the Python interpreter and runs inference outputting results to file

After running inference comes the event Ⓝ. In the latter, the tMLInference thread proceeds to draw boxes and labels with respective percentages (using imagemagick), based on the aforementioned text file (generated by the Python inference script). If the number of weapons detected surpasses the threshold stipulated then the threads waiting for the threat signal are notified (tRemoteClient, tCloud, tDatabase). This can be observed in listing 4.62.

```
        ...
        std::vector<std::string> lines;
        std::vector<std::string> aux;
        std::string temp;
        std::ifstream info;

        info.open(root_dir + " ... /model_output.txt");
        if(info.is_open()){
                perror("[tMLInference] Model output file opened");

                while(std::getline(info,temp)){
                        if(temp.size() > 0)
                                lines.push_back(temp);
                }
        }
        info.close();

        for(size_t i = 0; i < lines.size(); i++){
                std::vector<std::string> aux;
                split_str(lines.at(i), aux);

            detections.push_back(aux.at(0).c_str());
            int img = atoi(aux.at(0).c_str());
            capture_str[28] = (char)(img+0x30);

            int ymin = atoi(aux.at(1).c_str());
            int xmin = atoi(aux.at(2).c_str());
            int ymax = atoi(aux.at(3).c_str());
            int xmax = atoi(aux.at(4).c_str());

            std::string label = aux.at(5) + " " + aux.at(6);

            cam.drawRectangle(capture_str, capture_str,
                        Camera::Point2D(xmin, ymin), Camera::Point2D(xmax,
                        ymax), Camera::Color(0), Camera::Color(255, 0, 255, 0));
            cam.drawRectangle(capture_str, capture_str,
                        Camera::Point2D(xmin, ymin), Camera::Point2D(xmin+70, ymin-20),
                        Camera::Color(255, 255, 255, 255), Camera::Color(0));
            cam.drawText(capture_str, capture_str, label,
                        Camera::Point2D(xmin+5, ymin-15), 25, 15, Camera::Color(255, 0, 0,
                            0));

        }

        if(lines.size() > THREAT_THRESHOLD){
```

```
            /* Condition variable broadcast would be better
            for signalling the two threads */

            threat_found = true;

            /* Signal the condition variable */
            if (pthread_cond_signal(&cThreatFoundCloud) > 0)
                perror("[tMLInference] pthread_cond_signal failed on cThreatFoundCloud");

            /* Signal the condition variable */
            if (pthread_cond_signal(&cThreatFoundDB) > 0)
                perror("[tMLInference] pthread_cond_signal failed on cThreatFoundDB");

            notify_rc = true;

            /* Signal the condition variable */
            if (pthread_cond_signal(&cNotification) > 0)
                perror("[tMLInference] pthread_cond_signal failed on cNotification");
        }

        if(thread_unblocked[0] && thread_unblocked[1])
                threat_found = false;
    }
    fclose(file);
    pthread_exit(NULL);
}
```

Listing 4.62: Thread tMLInference reads inference output file and draws boxes and labels (signal if threat found)

In event ⓪, one sees the tCameraControl thread waiting on the cCloudFinished for the condition signal informing the frames where submitted to the cloud (listing 4.63).

```
    ...
    while(true){

        if(warm_start){
        /* Blocks on frame cache mutex waiting for tCloud to submmit the frames */
        pthread_mutex_lock(&mFrameCacheCam);
        while (!cloud_finished){
                /* Automatically unlocks mutex when signalled */
                if (pthread_cond_wait(&cCloudFinished, &mFrameCacheCam) > 0)
                        perror("[tCameraControl] pthread_cond_wait failed on cCloudFinished"
                            );
                }
        /* Clear shared variable */
        cloud_finished = false;
        }
        ...
```

Listing 4.63: Thread tCameraControl awaits cCloudFinished condition signal

In the events Ⓟ and Ⓡ, the threads tRemoteClient and tDatabase create a detached daughter thread for sending notifications (d_tRemoteClient) and database update (d_tDatabase), respectively. This can be seen in listing 4.64.

```
static void* d_rclientHandler(void* arg){
    perror("[d_tRemoteClient] Starting d_tRemoteClient");

    /* Send threat notification to the remote client */

    perror("[d_tRemoteClient] Last notification sent");

    /* Don't care about thread's return */
    pthread_detach(pthread_self());

    perror("[d_tRemoteClient] Remote Client daughter thread termination");
    pthread_exit(NULL);
}

/* In rclientHandler */
        ...
        perror("[tRemoteClient] Remote client notification trial");

        /* Remote client daughter thread setup */
        Thread::Config rc_daughter;
        const Thread* rc_daughter_thread;
        rc_daughter.name = "d_tRemoteClient";
        rc_daughter.priority = NORMAL;
        rc_daughter.args = NULL;
        rc_daughter.start_routine = d_rclientHandler;

        /* Remote client thread creation */
        rc_daughter_thread = const_cast<Thread*>(createThread(rc_daughter));
    }
    pthread_exit(NULL);
}

static void* d_databaseHandler(void* arg){
    perror("[d_tDatabase] Starting d_tDatabase");

    /* Update database */

    perror("[d_tDatabase] Updated database");

    /* Don't care about thread's return */
    pthread_detach(pthread_self());

    perror("[d_tDatabase] Database daughter thread termination");
    pthread_exit(NULL);
}

/* In databaseHandler */
        ...
```

```
        perror("[tDatabase] Database update trial");

        /* Database daughter thread setup */
        Thread::Config db_daughter;
        const Thread* db_daughter_thread;
        db_daughter.name = "d_tDatabase";
        db_daughter.priority = BELOW_NORMAL;
        db_daughter.args = NULL;
        db_daughter.start_routine = d_databaseHandler;

        /* Database daughter thread creation */
        db_daughter_thread = const_cast<Thread*>(createThread(db_daughter));
    }
    pthread_exit(NULL);
}
```

Listing 4.64: Threads tRemoteClient and tDatabase create detached daughter threads for threat scenario handling

In the event (Q), the tCloud thread creates a detached daughter thread as well for frame submission to Google Drive (listing 4.65).

```
        ...
        perror("[tCloud] Frame upload trial");

        /* Cloud daughter thread setup */
        Thread::Config cloud_daughter;
        const Thread* cloud_daughter_thread;
        cloud_daughter.name = "d_tCloud";
        cloud_daughter.priority = ABOVE_NORMAL;
        cloud_daughter.args = NULL;
        cloud_daughter.start_routine = d_cloudHandler;

        /* Cloud daughter thread creation */
        cloud_daughter_thread = const_cast<Thread*>(createThread(cloud_daughter));
    }
    pthread_exit(NULL);
}
```

Listing 4.65: Thread tCloud creates a detached daughter thread for frame submission to cloud

The last three events (q0) (q1) (q2) are related to the d_tCloud daughter thread. This thread also makes use of Python to connect to the cloud and store the frames captured in Google Drive. The routine executes the drive submission Python script and then signals the cCloudFinished condition variable.

```
static void* d_cloudHandler(void* arg){
    perror("[d_tCloud] Starting d_tCloud");

    std::string script_name = root_dir + " ... /argos_drive.py";
```

```
    /* Transform the directory string into a C string */
    const char* _script_name = script_name.c_str();

    /* Submmit Frames to cloud */
    /* Initialize the Python interpreter */
    Py_Initialize();

    PyRun_SimpleString("import sys");
    PyRun_SimpleString("import os");
    PyRun_SimpleString("print('[d_tCloud] Executing Google Drive Python Script')");

    /* Run Script */
    FILE* file = fopen(_script_name, "r");
    if(file != NULL)
        PyRun_SimpleFile(file, _script_name);
    fclose(file);

    /* Close the Python interpreter */
    Py_Finalize();

    perror("[d_tCloud] Last Frames uploaded");

    /* Signal the condition variable */
    cloud_finished = true;
    if (pthread_cond_signal(&cCloudFinished) > 0)
        perror("[d_tCloud] pthread_cond_signal failed on cCloudFinished");

    /* Don't care about thread's return */
    pthread_detach(pthread_self());

    perror("[d_tCloud] Cloud daughter thread termination");
    pthread_exit(NULL);
}
```

Listing 4.66: Thread d_tCloud submits the frames to the cloud

The adapted Python scripts for the application can be found in C.4 and C.5. [18] [31].

4.6 Wi-Fi Setup

Besides the tools configured in buildroot and explained in 4.1, for connecting to a WLAN network, the machine needs a series of commands and configuration files for setting up a network interface and configuring the connection to a certain network.

The first of such files is `/etc/network/interfaces`. There can be defined a list of commands to execute in different stages of the process of bringing up and down each interface interface. In this case, the excerpt added to the file (in listing 4.67 defines:

- That the created interface should be brought automatically as the device boots;
- A command to execute before the interface is brought up. It calls the WPA supplicant, specifying what driver to use, as well as the configuration file for the interface;
- A command to execute after the interface has been brought down. It kills the WPA supplicant.

```
auto wlan0
iface wlan0 inet dhcp
  pre-up wpa_supplicant -B -D nl80211 -i wlan0 -c /etc/wpa_supplicant.conf
  post-down killall -q wpa_supplicant
```

Listing 4.67: Declaration of the `wlan0` interface

The last file is the configuration file. The excerpt added to this file, represented in listing 4.68 declares the network one wishes to connect to, its ID, passphrase and security policies.

```
network={
        ssid="MYNETWORK"
        scan_ssid=1
        proto=RSN WPA
        key_mgmt=WPA-PSK
        pairwise=CCMP TKIP
        group=CCMP TKIP
        auth_alg=OPEN
        priority=0
        id_str="My Network"
        psk="my_passcode"
}
```

Listing 4.68: List of networks and security policies

4.7 Post-Build Script

In order to automate the loading of modules, scripts and libraries onto the freshly-built image, the post-build script provided by Buildroot was taken advantage of.

```
#!/bin/sh
# [buildroot]/board/raspberrypi

set -u
set -e

# Add a console on tty1
if [ -e ${TARGET_DIR}/etc/inittab ]; then
    grep -qE '^tty1::' ${TARGET_DIR}/etc/inittab || \
        sed -i '/GENERIC_SERIAL/a\
tty1::respawn:/sbin/getty -L  tty1 0 vt100 # HDMI console' ${TARGET_DIR}/etc/inittab
fi

cp package/busybox/S10mdev ${TARGET_DIR}/etc/init.d/S10mdev
chmod 755 ${TARGET_DIR}/etc/init.d/S10mdev
cp package/busybox/mdev.conf ${TARGET_DIR}/etc/mdev.conf

# copy interfaces and wpa_supplicant files from desktop to the board folder
cp "$HOME/Desktop/ARGOS/argos/4. Implementation/2. Software Implementation/4. Support/
    dev_files/important_files/system/interfaces" board/raspberrypi/interfaces
cp "$HOME/Desktop/ARGOS/argos/4. Implementation/2. Software Implementation/4. Support/
    dev_files/important_files/system/wpa_supplicant.conf" board/raspberrypi/wpa_supplicant.
    conf
cp "$HOME/Desktop/ARGOS/argos/4. Implementation/2. Software Implementation/4. Support/
    dev_files/important_files/system/libtensorflow-lite.a" board/raspberrypi/libtensorflow-
    lite.a
cp "$HOME/Desktop/ARGOS/argos/4. Implementation/2. Software Implementation/4. Support/
    dev_files/important_files/system/S60gpio" board/raspberrypi/S60gpio
cp "$HOME/Desktop/ARGOS/argos/4. Implementation/2. Software Implementation/1. Main/drivers/
    gpio/gpio.ko" board/raspberrypi/gpio.ko


# copy interfaces and wpa_supplicant files from the board folder to the target
cp board/raspberrypi/interfaces ${TARGET_DIR}/etc/network/interfaces
cp board/raspberrypi/wpa_supplicant.conf ${TARGET_DIR}/etc/wpa_supplicant.conf
cp board/raspberrypi/libtensorflow-lite.a ${TARGET_DIR}/usr/lib/libtensorflow-lite.a
cp "$HOME/Desktop/ARGOS/argos/4. Implementation/2. Software Implementation/4. Support/
    dev_files/important_files/system/S60gpio" ${TARGET_DIR}/etc/init.d/S60gpio

# make S00gpio executable
chmod +x ${TARGET_DIR}/etc/init.d/S60gpio

# create directory for gpio driver
if [ ! -d ${TARGET_DIR}/lib/modules/gpio ]
then
        mkdir ${TARGET_DIR}/lib/modules/gpio
```

```
fi

cp board/raspberrypi/gpio.ko ${TARGET_DIR}/lib/modules/gpio/gpio.ko

# create /home directory for applications
if [ ! -d ${TARGET_DIR}/home ]
then
        mkdir ${TARGET_DIR}/home
fi

# create first-boot signal in target
touch ${TARGET_DIR}/home/.firstboot

# cp board/raspberrypi/sshd_config ${TARGET_DIR}/etc/ssh/sshd_config
```

Listing 4.69: Buildroot post-build script

4.8 Machine Learning

As mentioned in section 3.5.4, one intended to devise an Object Detection Classifier (ODC) with the purpose of detecting various types of guns in frames captured by the Raspberry Pi's vision module.

4.8.1 Dataset Preparation

As aforementioned, all the images respected what was stipulated in the design phase, having less than 10.0 KB of size, to make the training swifter. The overall image pool ended up containing 333 images. The latter were divided into three datasets based on the figure 2.7.3. The training set contains 70% of images and the validation and test sets contain 15% each (figure 4.8.1). Additionally, all the images had to be manually labelled one by one to ensure a correct initial setup.

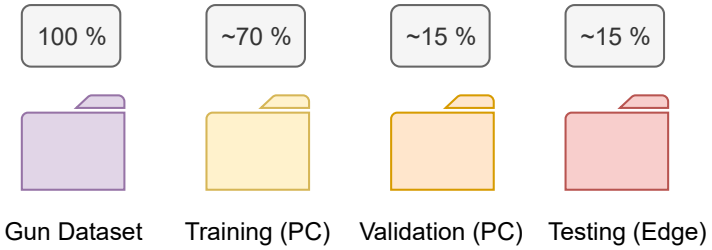


Figure 4.8.1: Implementation - Dataset Division

4.8.2 Training and Validation

To perform the model training and validation, one used the SSD-MobileNet model, a pre-trained ML model optimized for running on edge devices. But opted for its quantized version, since it used 8-bit integer values instead of 32-bit floating values within the Neural Network (NN), which allowed for a more efficient run on the GPU.

To take a previously trained model (Quantized SSD-MobileNet) and adapt it to the application 4.8.3, one recurred to Transfer Learning [18]. This form of learning is based on the idea of overcoming the isolated learning paradigm and utilising knowledge acquired for one task to solve related ones [24] [25]. In figure 4.8.2, one can see an abstract representation of transfer learning. The learning of the first task generates some type of knowledge useful for learning the second task, related in some sense to the first task.

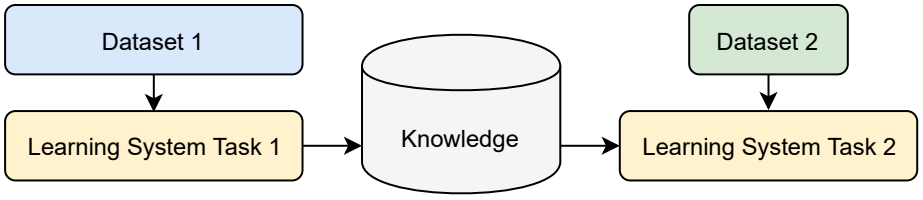


Figure 4.8.2: Transfer Learning Block Diagram (Abstract Scenario)

In this case, the model for ARGOS is similar to the pre-trained model only because they are both ODCs, the quantized SSD-MobileNet model doesn't detect weapons or guns. Considering this, the ARGOS' gun predictor model relies on a smaller amount of labels and training data, with only one label (gun).

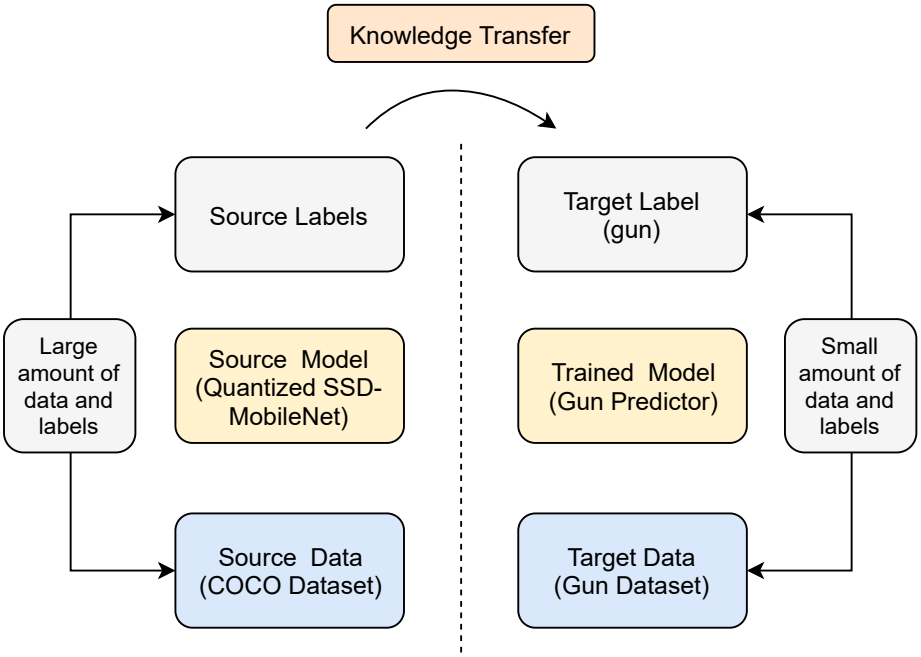


Figure 4.8.3: Transfer Learning Block Diagram (ARGOS Scenario)

Recurring to this technique, one was able to successfully re-train the model to detect guns. The training was stopped when the loss was consistently below 2 [18] (marked in green), at 54384 steps. The figure 4.8.4 depicts some training steps where the loss was below 2, but the training was far from over since this value was inconsistent. That fact can be assessed further by looking at the loss graphs 4.8.5.

```
I0127 14:15:21.889530 12192 learning.py:512] global step 54595: loss = 1.9252 (0.952 sec/step)
INFO:tensorflow:global step 54596: loss = 1.5192 (0.992 sec/step)
I0127 14:15:22.881181 12192 learning.py:512] global step 54596: loss = 1.5192 (0.992 sec/step)
INFO:tensorflow:global step 54597: loss = 1.2495 (0.935 sec/step)
I0127 14:15:23.816576 12192 learning.py:512] global step 54597: loss = 1.2495 (0.935 sec/step)
INFO:tensorflow:global step 54598: loss = 1.8924 (0.965 sec/step)
I0127 14:15:24.782705 12192 learning.py:512] global step 54598: loss = 1.8924 (0.965 sec/step)
INFO:tensorflow:global step 54599: loss = 1.3859 (0.943 sec/step)
I0127 14:15:25.727658 12192 learning.py:512] global step 54599: loss = 1.3859 (0.943 sec/step)
INFO:tensorflow:global step 54600: loss = 1.4379 (0.973 sec/step)
I0127 14:15:26.710882 12192 learning.py:512] global step 54600: loss = 1.4379 (0.973 sec/step)
INFO:tensorflow:global step 54601: loss = 1.0022 (1.021 sec/step)
I0127 14:15:27.734829 12192 learning.py:512] global step 54601: loss = 1.0022 (1.021 sec/step)
INFO:tensorflow:global step 54602: loss = 1.5487 (1.092 sec/step)
I0127 14:15:28.829581 12192 learning.py:512] global step 54602: loss = 1.5487 (1.092 sec/step)
INFO:tensorflow:global step 54603: loss = 1.1777 (1.035 sec/step)
I0127 14:15:29.864516 12192 learning.py:512] global step 54603: loss = 1.1777 (1.035 sec/step)
INFO:tensorflow:global step 54604: loss = 1.3801 (1.080 sec/step)
I0127 14:15:30.944046 12192 learning.py:512] global step 54604: loss = 1.3801 (1.080 sec/step)
INFO:tensorflow:global step 54605: loss = 1.4307 (1.022 sec/step)
I0127 14:15:31.965893 12192 learning.py:512] global step 54605: loss = 1.4307 (1.022 sec/step)
INFO:tensorflow:global step 54606: loss = 1.6680 (1.130 sec/step)
I0127 14:15:33.106709 12192 learning.py:512] global step 54606: loss = 1.6680 (1.130 sec/step)
INFO:tensorflow:global step 54607: loss = 1.1497 (0.984 sec/step)
I0127 14:15:34.093091 12192 learning.py:512] global step 54607: loss = 1.1497 (0.984 sec/step)
INFO:tensorflow:global step 54608: loss = 1.6513 (0.939 sec/step)
I0127 14:15:35.041638 12192 learning.py:512] global step 54608: loss = 1.6513 (0.939 sec/step)
```

Figure 4.8.4: Model Training

4.8.3 Loss Analysis

The aforementioned loss graphs attained utilising the TensorBoard tool of TensorFlow are represented in figure 4.8.5. When one analyses closely, it's noticeable that the overall loss value was erratic by nature, acknowledging this, one opted to proceed when the training reached about 12h total, with most loss values per step below 2.

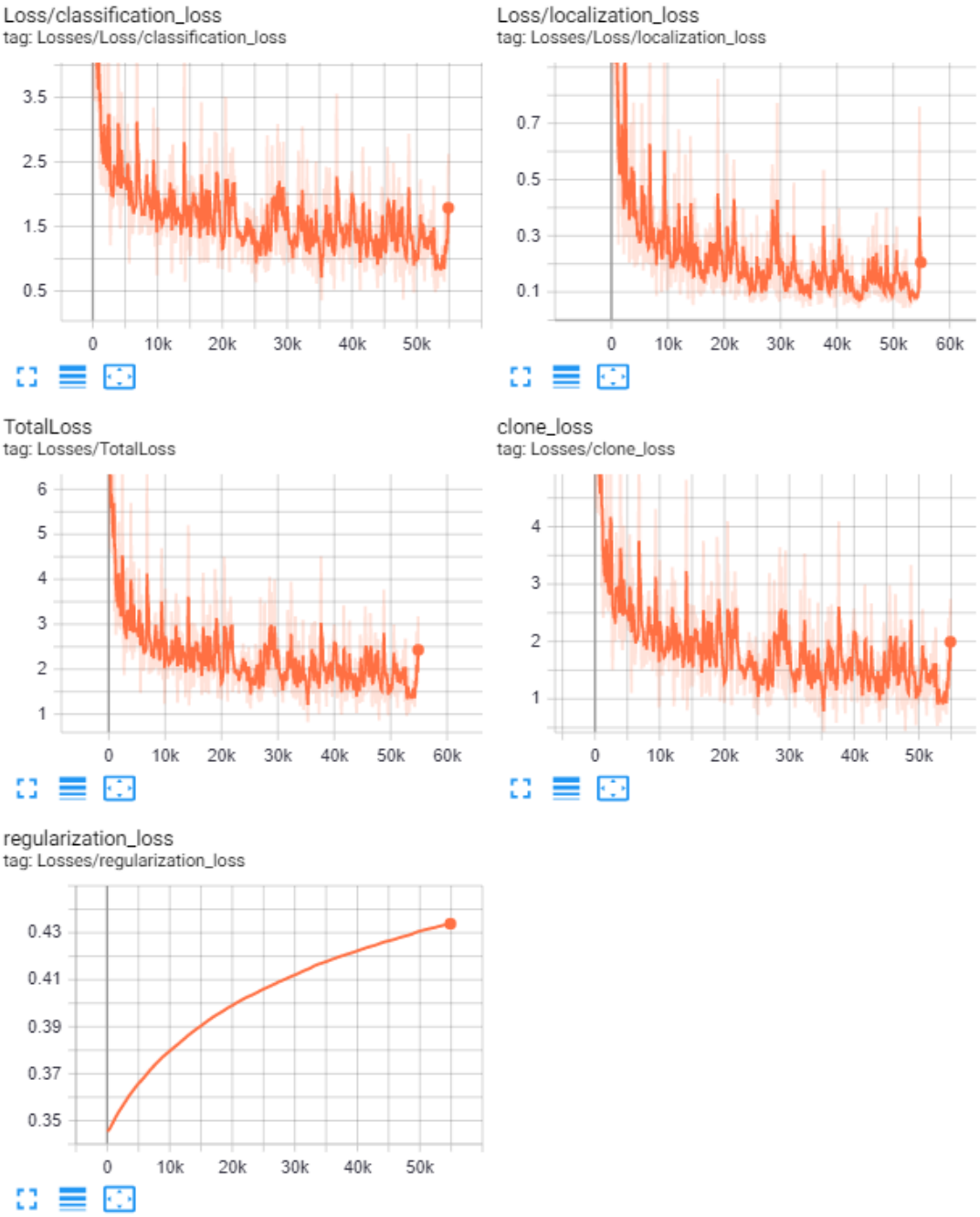


Figure 4.8.5: Model Loss Graphs

4.8.4 Model Conversion

The model conversion phase required the TensorFlow Lite Converter (TOCO) installation. The converter took the frozen graph attained at the end of the training stage and translated it to a TFLite file that represents the ARGOS' gun predictor ML model. Note that the after-training model used Keras and TensorFlow API during training. In figure 4.8.6 one can see an overview of the conversion process.

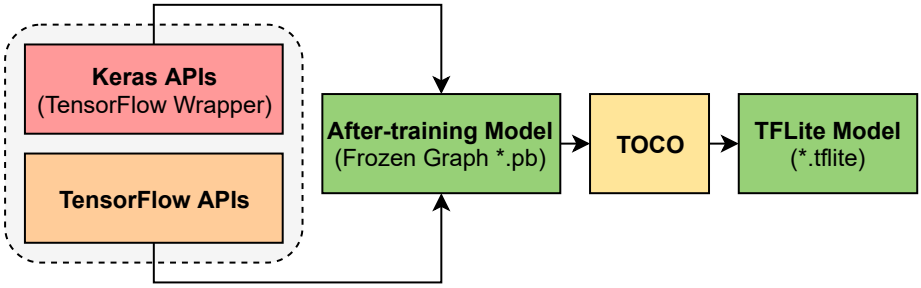


Figure 4.8.6: Model Conversion Process

4.8.5 Deployment on Embedded Target

The deployment stage regarding the TFlite model itself consisted of sending the latter and the label map to the edge device (Raspberry). Although, as mentioned in section 4.1, the TensorFlow package it's not available through the Buildroot tool. Hence, it was necessary to cross-compile a TensorFlow library for the Raspberry (armv7l). This static library was placed under the libs folder, as it's possible to observe in figure 4.8.7.

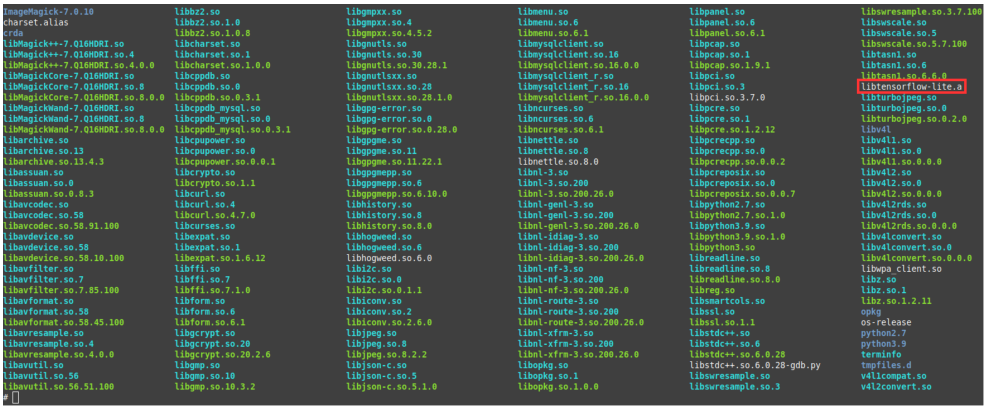


Figure 4.8.7: Deployment of the TensorFlow Library

However, despite the efforts to incorporate TensorFlow in the system, python didn't detect the library, even when converting it into a shared object as depicted in listing 4.70.

```
~/Desktop/buildroot-2020.11.1/output/host/bin/arm-buildroot-linux-uclibcgnueabihf-g++ -
    shared -o libtflite.so libtensorflow-lite.a
```

Listing 4.70: Conversion of the static Tensorflow library

4.9 Database

The conversion from the database logic model stipulated in section 3.7.2 figure 3.7.4, presupposes three types of scripts for the creation of the database itself, its tables, constraints, attributes, and functionalities.

- **Data Definition Language (DDL)** scripts;
- **Data Manipulation Language (DML)** scripts;
- **Data Query Language (DQL)** scripts;

4.9.1 DDL Scripts

DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. Therefore, they are associated with the creation of the database and all the tables it includes, in the listing 4.71 one can see the creation of the ARGOS database (argos_db) and its first table (users), using MySQL.

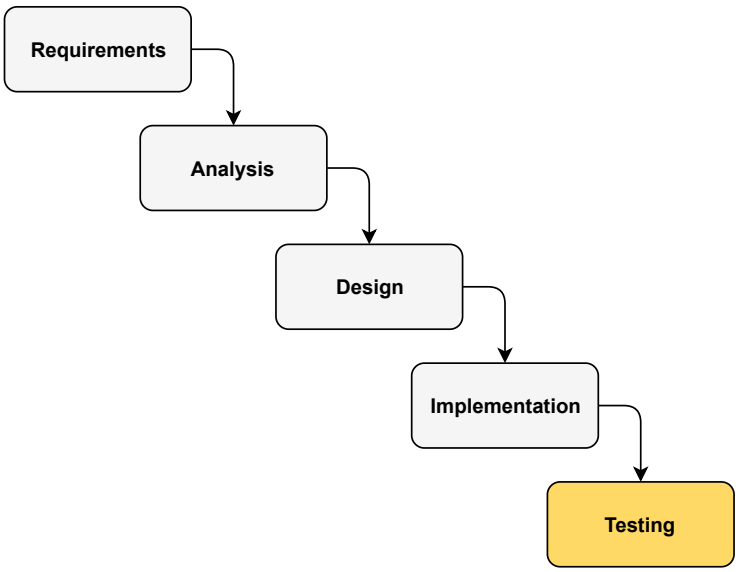
```
drop database if exists argos_db;
create database argos_db;
use argos_db;

drop table if exists users;
create table users (
        user_id int not null auto_increment primary key,
        username varchar(255),
        _address varchar(255) default null,
        email varchar(255) default null,
        country varchar(30) default null,
);

drop table if exists cameras;
create table cameras (
        camera_id int not null auto_increment primary key,
        user_id int,
        camera_name varchar(255),
        camera_resolution varchar(5),
        camera_model(255),
        camera_mode(255),
        constraint user_fk foreign key (user_id) references users(user_id) on update cascade
);

/*Other table creation*/
...
```

Listing 4.71: ARGOS Database/users Table/cameras Table Creation (Extended in B.1)

Chapter 5

Testing

This phase will present the results of the unit and integration tests performed on the ARGOS system. Hence, the expected results will be compared to the real results to gain insight into what could be improved in the future through a verification stage.



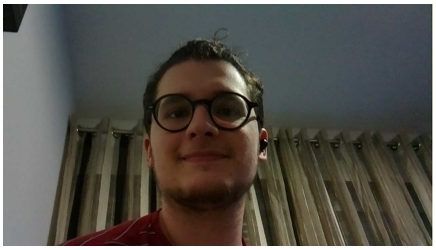
5.1 Test Setup and Results

As specified in section 3.9, some unit and integration tests were conducted on the system.

5.1.1 Unit Tests: Camera and Local Storage

The camera interface and local storage of frames were key objectives for the ARGOS' system. The camera setup is depicted in figure 5.1.1a and one picture taken with it is represented in figure 5.1.1b. For this effect, the command `ffmpeg -f video4linux2 -s 1280x720 -input_format mjpeg -i /dev/video0 -vframes 1 test.jpg` was used. It states that a single `1280x720` picture in the `MJPEG` format should be taken using device `/dev/video0`.

(a) Camera Setup



(b) Example Image capture

Figure 5.1.1: Frame capture tests

Moreover, the local storage of the images captured by the camera can be seen in figure 5.1.2.

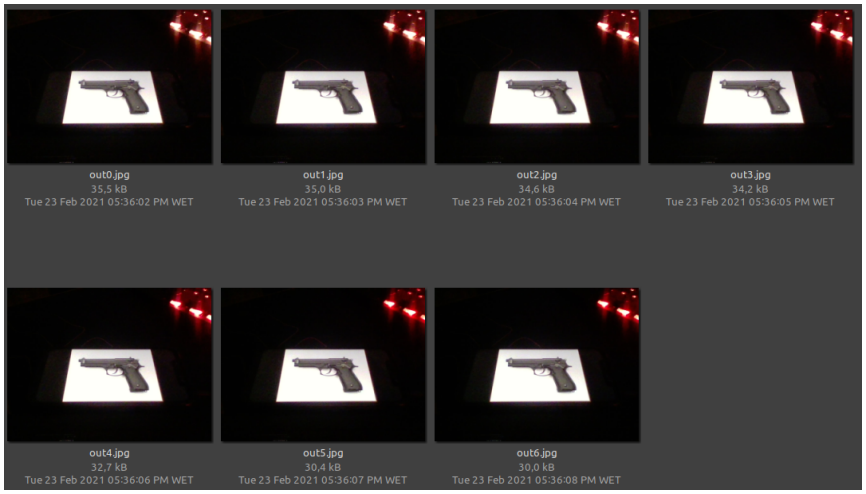


Figure 5.1.2: Local Storage: Save Frames Locally

5.1.2 Unit Tests: Cloud

As mentioned before, since the capacity of cloud-based storage is much greater than the local one, the project also had the objective of sending the preemptively captured images to the cloud. In this case, one used the Google Cloud API to submit the data to a certain google drive folder, linked with a personal Google account. The drive Python script creates a service based on the `client_secrets.json` file, containing the client ID, client secret, and other OAuth 2.0 parameters before uploading the frames. In figure 5.1.3 it is possible to see the upload of some random images using only the Python script.

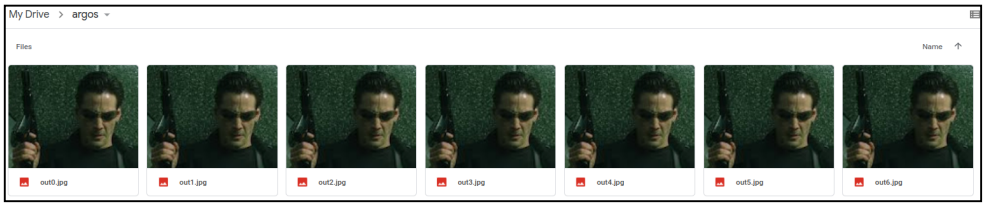


Figure 5.1.3: Cloud: Frame submission test

5.1.3 Unit Tests: Machine Learning

For machine learning, one performed tests running the built model. Note that the model was pre-emptively converted to an embedded-optimized version to ensure it would at least perform reasonably onto the edge. The tests will be described and analysed in this report section.

In terms of the machine learning model, one conducted single, double and multiple gun detection tests. This contributed to identifying the level of accuracy in the detection and framing of the weapons.

Model Tests: Single Gun Detection

Firstly, one tried to test gun detection in frames where there was only one weapon per frame. Both of the tests proved to be successful since the two different guns were well framed and the predictor was almost 100% sure they were indeed guns, as its possible to see in figure 5.1.4.

(a) Frame 1

(b) Frame 2

Figure 5.1.4: Single Gun Detection Tests

Model Tests: Double Gun Detection Tests

Proceeding, the tests moved to detect two guns in a frame, either with one person holding the two guns or with two different holders. These tests were also successful, but with the introduction of the second gun, the predictor started to show a lower percentage of certainty towards one of them. Additionally, all the weapons were detected despite the higher or lower degree of confidence displayed (figures 5.1.5 and 5.1.6).

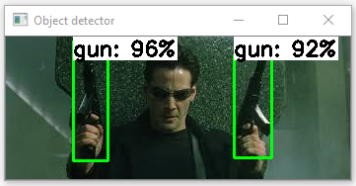
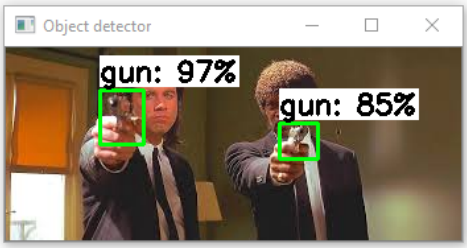
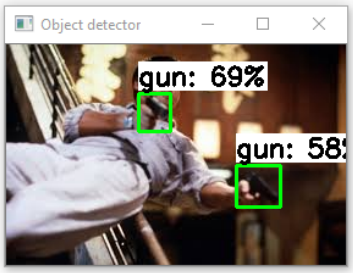


Figure 5.1.5: Double Gun Detection Tests



(a) Frame 2

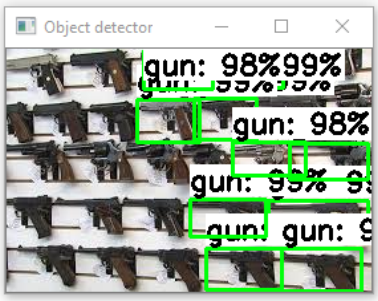


(b) Frame 3

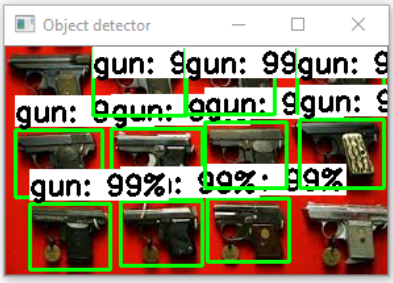
Figure 5.1.6: Double Gun Detection Tests

Model Tests: Multiple Gun Detection Tests

Concerning the case where the model should be able to identify multiple weapons (figure 5.1.7), one can conclude that in some cases where the model is overwhelmed with images of this type, the framing starts to fail. Despite that, for the guns that were detected the model predicts correctly.



(a) Frame 1



(b) Frame 2

Figure 5.1.7: Multiple Gun Detection Tests

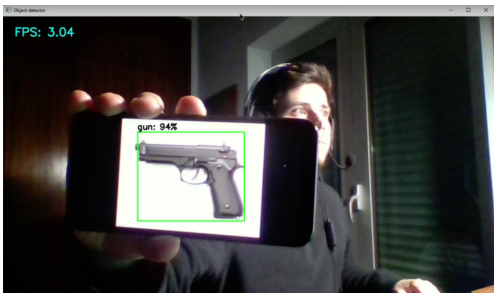
Model Tests: Video Gun Detection Tests

After the tests on still images, one moved on to test on the PC's webcam to further assess the model's reliability.

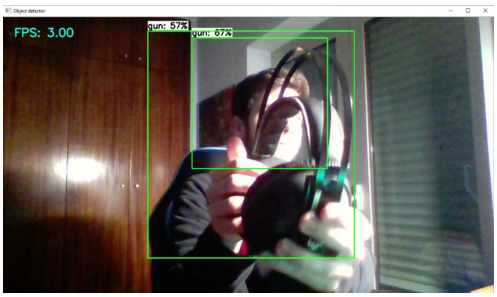
Initially, the model was tested with a smartphone picture of a gun (figure 5.1.8a). The test proved to be successful, with the model's accurate gun pinpoint (94%).

In the second test (figure 5.1.8b), some flaws of the model started to show. The headphones were detected as two different guns because of the weapon-like hand grip and the headphones' blackish colour scheme. In the third and fourth frames (figures 5.1.9a and 5.1.9b), one can see that, once again, the headphones are mistaken by a gun for the same reasons.

Summarising, the model appeared to be reliable at this stage since it could detect and frame most of the weapons presented up to a certain degree of confidence. In the tests where non-gun objects were detected, the overall percentage was always lower than the one presented for the actual guns, as expected. Additionally, one noticed that some of the problems might be caused by model overfitting.

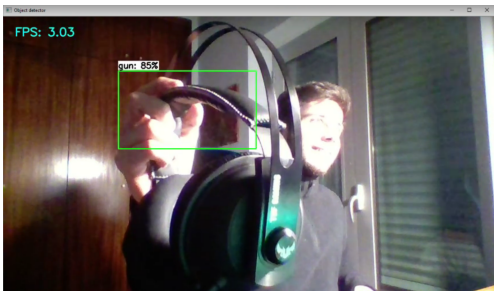


(a) Frame 1

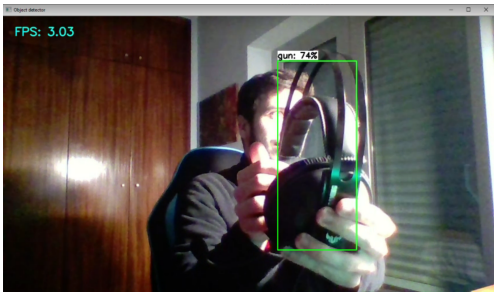


(b) Frame 2

Figure 5.1.8: Webcam Gun Detection Tests



(a) Frame 3



(b) Frame 4

Figure 5.1.9: Webcam Gun Detection Tests

It's also important to note that the drawing of the boxes and labels in the images presented in this section was done using the OpenCV tool. Although, the final version of the system uses imagemagick to perform the aforementioned task and save the output frames in a specific folder.

5.1.4 Unit Tests: Daemon

Regarding the Daemon, the tests performed are depicted in figures 5.1.10 and 5.1.12. With these tests, it was possible to see that the Daemon was successfully created and that it outputted onto `syslog`.

```
(base) silent@Silent-Asus:~/Desktop/ARGOS/argos$ ps -xj | grep shutdownDaemon
  13386   16094   16093   13386 pts/5      16093 S+    1000   0:00 grep --color=auto shutdownDaemon
(base) silent@Silent-Asus:~/Desktop/ARGOS/argos$
```

Figure 5.1.10: Daemon: Test 1

```
(base) silent@Silent-Asus:~/Desktop/ARGOS/argos$ grep shutdownDaemon /var/log/syslog
Feb 13 01:17:00 Silent-Asus shutdownDaemon[29880]: Shutdown Daemon: Up and Running.
Feb 13 01:20:50 Silent-Asus shutdownDaemon[30208]: Shutdown Daemon: Up and Running.
Feb 13 01:35:51 Silent-Asus shutdownDaemon[31480]: Shutdown Daemon: Up and Running.
Feb 13 01:45:01 Silent-Asus shutdownDaemon[32651]: [ARGOS] Shutdown Daemon Up and Running
Feb 13 01:49:11 Silent-Asus shutdownDaemon[33428]: [ARGOS] Shutdown Daemon Up and Running
Feb 13 01:51:41 Silent-Asus shutdownDaemon[33785]: [ARGOS] Shutdown Daemon Up and Running
```

Figure 5.1.11: Daemon: Test 2

5.1.5 Unit Tests: Device Drivers

For a simple GPIO device driver test, a client application was developed to turn on and off an LED using GPIO pin 17 as an output and read back the digital value on that same pin using GPIO pin 27 as an input.

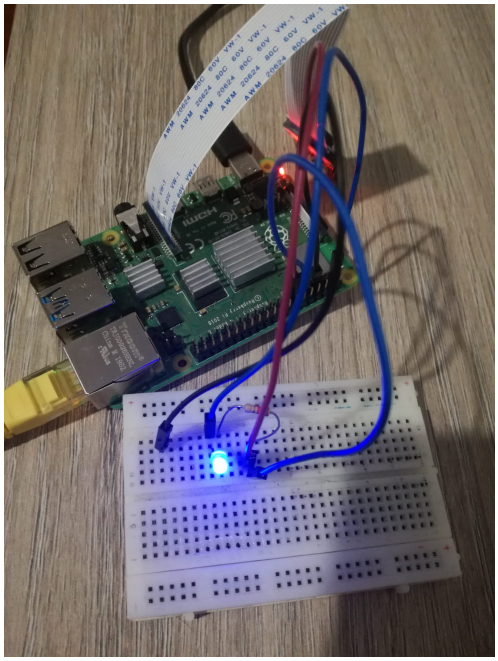


Figure 5.1.12: Device Driver: LED Control Test

5.2 Integration Tests

In terms of integration, the final system can capture frames from the camera stream, and store them locally (overwriting exiting ones) to later run inference with the first Python script, and once again store the post-inference frames in the specified folder (figure 5.2.1).

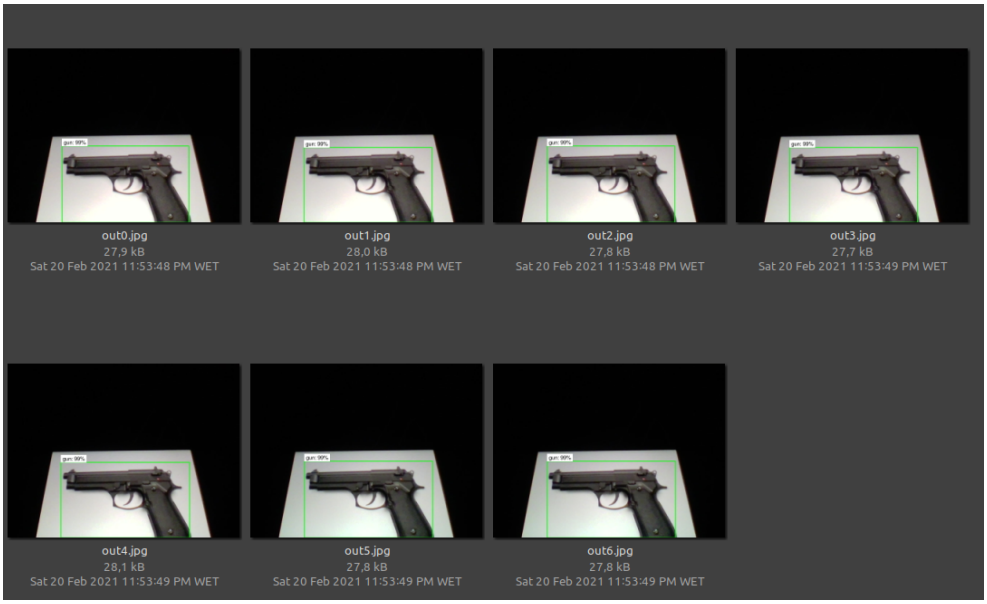


Figure 5.2.1: Local Storage

In figure 5.2.2 one can see an example of one of the frames captured after inference was run. The model was able to detect the weapon and identify its position in the frame accurately.

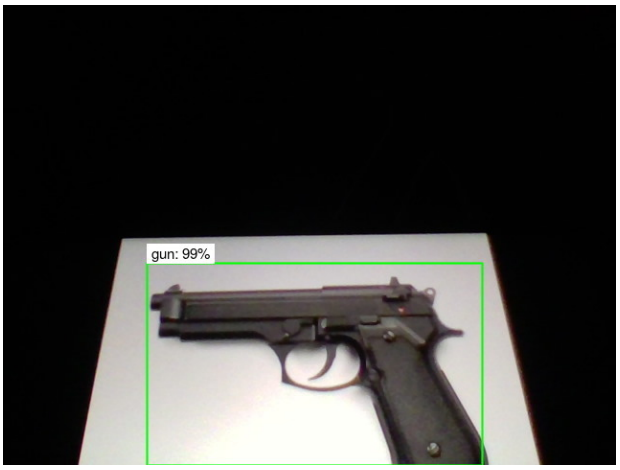


Figure 5.2.2: Single Frame Post-Inference

Additionally, after the inference is performed, the cloud daughter thread executes the second python script (described earlier) to submit the images to the cloud. This can be seen in figure 5.2.3.

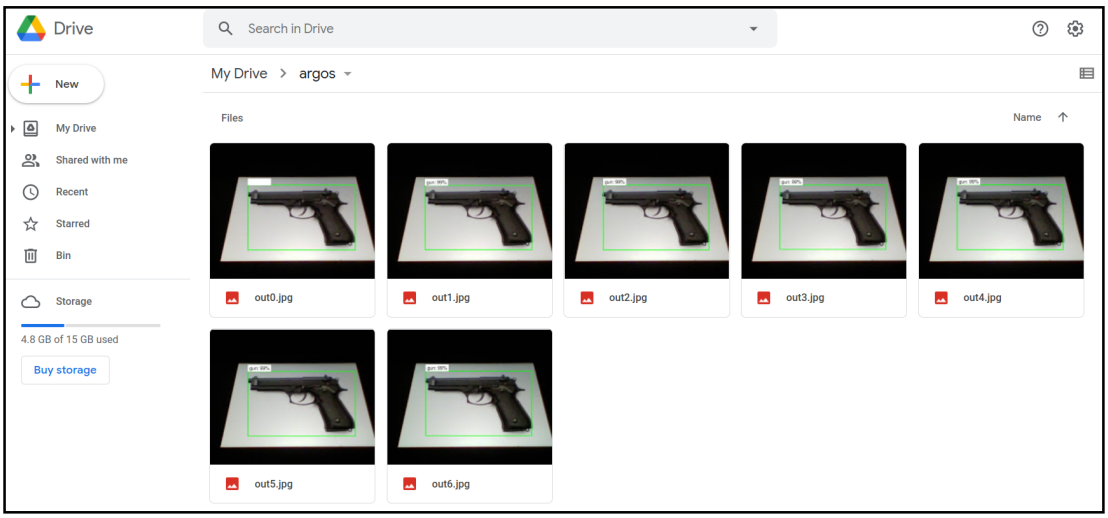


Figure 5.2.3: Remote Storage (Cloud)

The final application scheme is represented in figure 5.2.4.

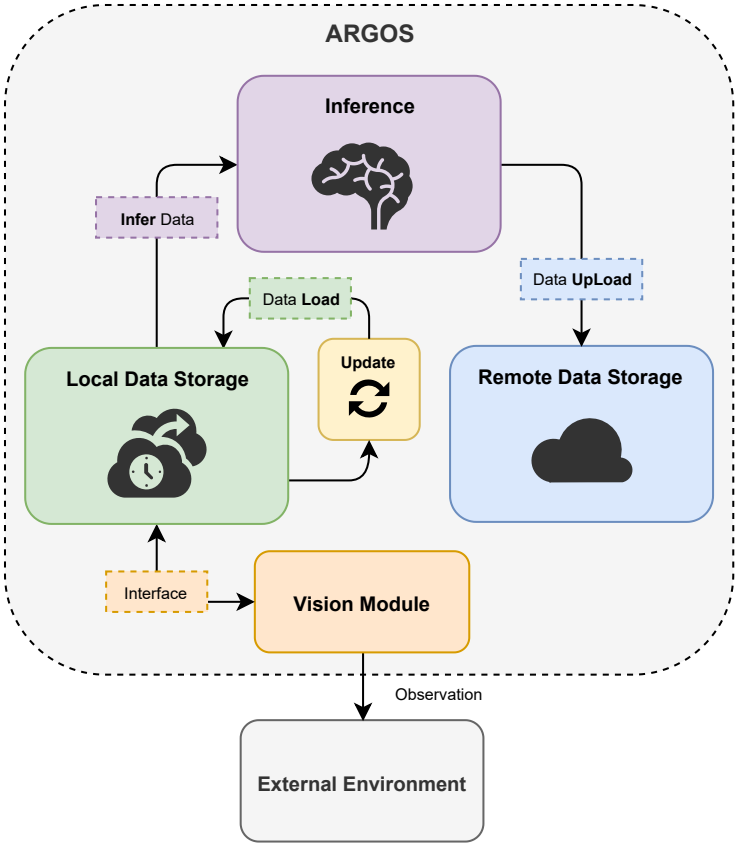


Figure 5.2.4: ARGOS: Final Scheme

5.3 Verification

At this stage one compares the expected test results with the actual results in order to identify points to improve in the next chapter.

5.4 Computational Unit Test Cases

5.4.1 Computational Unit: Unit Tests

Computational Unit: Unit Tests	Expected Results	Real Results
Press the power on button while the device is off	Successful power on sequence and connection with the Remote Client, Database and Cloud	Not tested
Press the power on button while the device is on	Disconnection from all remote systems and successful power off sequence without timeout	Not tested
Display a prototyope weapon in front of the camera	The device identifies the threat	The device identifies the threat
Trigger weapon detection without connection to database, cloud or Remote Client and then connect	The device stores the data locally and , upon connection, sends the data, guaranteeing no repetitions and deleting unnecessary local data	The device stores the data locally, sends the data and deletes unnecessary local data
Change amount of light in the area around the camera	Light sensor readings change accordingly	Not tested
Connect to the Wi-Fi	Connection successful	Connection successful
Control LEDs	LEDs change accordingly	LEDs change accordingly
Read button state	Button state read correctly	Button state read correctly

Table 5.1: Computational Unit Test Cases

5.5 Remote Client Test Cases

5.5.1 Remote Client: Unit Tests

Remote Client: Unit Tests	Expected Results	Real Results
Open app	App opens and displays login screen	Not tested
Insert credentials	App attempts to establish a connection with the Computational Unit	Not tested
Connection rejected	Show warning and allow for credentials reinsertion	Not tested
Connection accepted	Go into the control panel, immediately fetching pertinent information from the cloud and database	Not tested
Click specific event button	Show frames and metadata about that event	Not tested

Table 5.2: Remote Client: Unit Tests

5.6 Integration Tests

Integration Tests	Expected Results	Real Results
Display prototype weapon in front of the camera	Weapon is identified and frames and data start appearing in the Remote Client's GUI	Weapon is identified and frames are stored locally and on the cloud
Turn off camera	Warning shows up in Remote Client and device turns off within the specified time	Not Tested

Table 5.3: Integration Tests

Chapter 6

Conclusion

This project played a very important role both in providing the group with fundamental knowledge on how to manage a group project as well as solidifying the concepts that were learned throughout the semester. The group considers these strands to have been very well explored and considers the knowledge that lies therein to be of great value in the future. Although not everything was implemented exactly as intended, the group showed versatility and ingenuity in the ways in which it overcame some of the problems with which it was faced.

However, there were still some gaps in the work that was implemented in comparison to what was idealised. Although, the group is certain to have done everything possible to overcome the setbacks along the way, making everything that does work, do so reasonably well.

6.1 Future Works

Knowing that not everything works as well as expected, the group is left feeling that some things could be improved upon.

Integrate the Inference Engine Into the Target One of the toughest challenges to overcome was certainly making the Machine Learning algorithm work, mainly due to technical issues. That problem was eventually overcome in the host system, but it never worked in the target system, although it did feel like it was very close to being solved, as previously discussed.

Remote Client Application Due to the non-completion of other more essential components of the project, the Remote Client's application was left out of the list of implemented features. This represents, however, a big gap in the product's functionality, as this app plays a very important role in ARGOS as a product. It is, after all, what gives it the usability and expandability intended for the product since the beginning.

Enclosure for the Camera Another deemed less-essential feature, although still very important for ARGOS as a product is the enclosure for the Computational Unit. This was projected in the design phase but it never saw the light of day, as finding a way to print the enclosure in 3D during the confinement period proved to be too time-consuming for what it was worth.

Implement Light Sensing Capability Yet another feature of less priority that ended up not being implemented is the light-sensing capability. This is also another feature that felt like it was very close to being implemented since the interface with the hardware is provided by the official device driver for the chosen hardware, which is already part of the list of device drivers in the Linux kernel. This leaves only the standard device driver interface to be implemented.

Improve the Accuracy in Inference Although the act of inferring the presence of weapons in the analysed frames proved to be very successful for a wide variety of weapons, some issues persist regarding incorrect classification, as previously discussed.

Implement Advanced Functionality in the GPIO Device Driver Although the device driver for the GPIO shows exactly the intended results for simple read and write operations, features that would make it more versatile and allow for more efficient use at the application level, such as signal output and rising/falling edge interrupt on the input could be implemented to make the application more power-efficient.

Implement the Power Switch The projected push-button power switch was implemented in hardware but its software counterpart in the Shutdown Daemon and in the main application was not. That is yet another feature that is very important for ARGOS as a product, although not part of the core functionality.

Implement Communication Between the Computational Unit and the Remote Client The main communication pipe between both halves of the system was also missing despite being essential for the core the functionality of the product.

Implement Connection to the Database Although an individual module for the database service was devised, it never went on-line. This is also a very important service, as it allows for storing relevant event data to be analysed. As such, it should absolutely be implemented in a future revision of the project.

Bibliography

[1] Market, V., 2020. Video Surveillance Market By System, Marketsandmarkets™. [online] Marketsandmarkets.com. Available at:<https://www.marketsandmarkets.com/Market-Reports/video-surveillance-market-645.html> [Accessed 4 November 2020].

[2] Géron, A., 2019. Hands-On Machine Learning With Scikit-Learn and Tensorflow. Sebastopol, CA: O'Reilly Media, Inc

[3] GitHub. 2021. Sasankyadati/Guns-Dataset. [online] Available at: <https://github.com/SasankYadati/Guns-Dataset/tree/master/Labels> [Accessed 26 January 2021].

[4] Branco, S., Ferreira, A. and Cabral, J., 2019. Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey. Electronics, 8(11), p.1289.

[5] TensorFlow. 2020. Build Tensorflow Lite For Raspberry Pi. [online] Available at: <https://www.tensorflow.org/lite/guide/build_rpi> [Accessed 20 November 2020].

[6] Ltd., A., 2020. Cortex-A – Arm Developer. [online] Arm Developer. Available at: <https://developer.arm.com/ip-products/processors/cortex-a> [Accessed 20 November 2020].

[7] Sakr, F., Bellotti, F., Berta, R. and De Gloria, A., 2020. Machine Learning on Mainstream Microcontrollers. Sensors, 20(9), p.2638.

[8] Core Electronics. 2020. Raspberry Pi 4 Vs Raspberry Pi 3 Plus (Model B) - Tutorial. [online] Available at: <https://core-electronics.com.au/tutorials/raspberry-pi-4-vs-3-model-b-performance-benchmark.html> [Accessed 25 November 2020].

[9] Element14. 2020. Raspberry Pi 4 Model B Default GPIO Pinout With Poe Header. [online] Available at: <https://www.element14.com/community/docs/DOC-92640/l/raspberry-pi-4-model-b-default-gpio-pinout-with-poe-header> [Accessed 25 November 2020].

[10] Afzal, S., 2020. I2C Primer: What Is I2C? (Part 1) | Analog Devices. [online] Analog.com. Available at: <https://www.analog.com/en/technical-articles/i2c-primer-what-is-i2c-part-1.html> [Accessed 26 November 2020].

[11] Synopsys.com. 2020. Implementing MIPI Camera And Display Interfaces In New Applications Beyond Mobile. [online] Available at: <https://www.synopsys.com/designware-ip/technical-bulletin/implementing-mipi-camera.html> [Accessed 28 November 2020].

[12] Products, E., 2020. Camera Serial Interface (CSI-2) Sensors In Embedded Designs - Electronic Products. [online] Electronic Products. Available at: <https://www.electronicproducts.com/camera-serial-interface-csi-2-sensors-in-embedded-designs/> [Accessed 28 November 2020].

[13] Brenner, J. and Masney, B., 2020. Device Driver For Monitoring Ambient Light Intensity (Lux) Within The TAOS Tsl258x Family Of Devices (Tsl2580, Tsl2581, Tsl2583). [online] Github. Available at: <https://github.com/torvalds/linux/blob/

master/drivers/iio/light/tsl2583.c> [Accessed 28 November 2020].

[14] Nichols, B., Buttlar, D. and Farrell, J., 1998. Pthreads Programming. Beijing [China]: O'Reilly.

[15] Kernel.org. 2002. Part I - Video For Linux API — The Linux Kernel Documentation. [online] Available at: <https://www.kernel.org/doc/html/v4.12/media/uapi/v4l/v4l2.html> [Accessed 29 November 2020].

[16] Microsoft. 2020. COCO: Common Objects In Context. [pdf] Available at: <https://arxiv.org/abs/1405.0312> [Accessed 29 November 2020].

[17] GitHub. 2020. Tensorflow Object Detection API Tutorial Train Multiple Objects Windows 10. [online] Available at: <https://github.com/EdjeElectronics/TensorFlow-Object-Detection-API-Tutorial-Train-Multiple-Objects-Windows-10> [Accessed 29 November 2020].

[18] GitHub. 2021. Edjeelectronics/Tensorflow-Lite-Object-Detection-On-Android-And-Raspberry-Pi. [online] Available at: <https://github.com/EdjeElectronics/TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi> [Accessed 26 January 2021].

[19] Waveshare.com. 2020. TSL2580-81 Datasheet. [online] Available at: <https://www.waveshare.com/w/upload/6/69/TSL2580-81_DS000417_1-00.pdf> [Accessed 30 November 2020].

[20] Smbus.org. 2020. Smbus Specifications. [online] Available at: <http://www.smbus.org/specs/SMBus_3_1_20180319.pdf> [Accessed 30 November 2020].

[21] Nxp.com. 2020. Designing With I2C Bus Devices. [online] Available at: <https://www.nxp.com/docs/en/training-reference-material/DESIGNING-WITH-I2C-BUS-DEVICES.pdf> [Accessed 30 November 2020].

[22] Waveshare.com. 2020. Light Sensor Schematic. [online] Available at: <https://www.waveshare.com/w/upload/a/a2/Light-Sensor-Schematic.pdf> [Accessed 30 November 2020].

[23] 2020. MAX16150 Nanopower Pushbutton On/Off Controller And Battery Freshness Seal. 3rd ed. [ebook] Maxim Integrated. Available at: <https://datasheets.maximintegrated.com/en/ds/MAX16150.pdf> [Accessed 24 November 2020].

[24] Medium. 2021. A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning. [online] Available at: <https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a> [Accessed 28 January 2021].

[25] Sarkar, D., Bali, R. and Ghosh, T., n.d. Hands-on transfer learning with Python.

[26] Kernel.org. 2021. Part I - Video for Linux API — The Linux Kernel documentation. [online] Available at: <https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/v4l2.html> [Accessed 14 February 2021].

[27] Man7.org. 2021. Michael Kerrisk - man7.org. [online] Available at: <https://man7.org/index.html> [Accessed 15 February 2021].

[28] O'Reilly Online Learning. 2021. Linux System Programming, 2nd Edition. [online] Available at: <https://www.oreilly.com/library/view/linux-system-programming/9781449341527/ch04.html> [Accessed 15 February 2021].

[29] Michael Kerrisk. The Linux programming interface: a Linux and UNIX system programming handbook.

[30] Codeproject.com. 2021. Embedding Python program in a C/C++ code. [online] Available at: <https://www.codeproject.com/Articles/820116/Embedding-Python-program-in-a-C-Cplusplus-code> [Accessed 23 February 2021].

[31] Learn Data Analysis. 2021. Google Drive API in Python | Getting Started (Lesson 1) - Learn Data Analysis. [online] Available at: <https://learndataanalysis.org/google-drive-api-in-python-getting-started-lesson-1/> [Accessed 23 February 2021].

Appendices

Appendix A

Augmented Figures

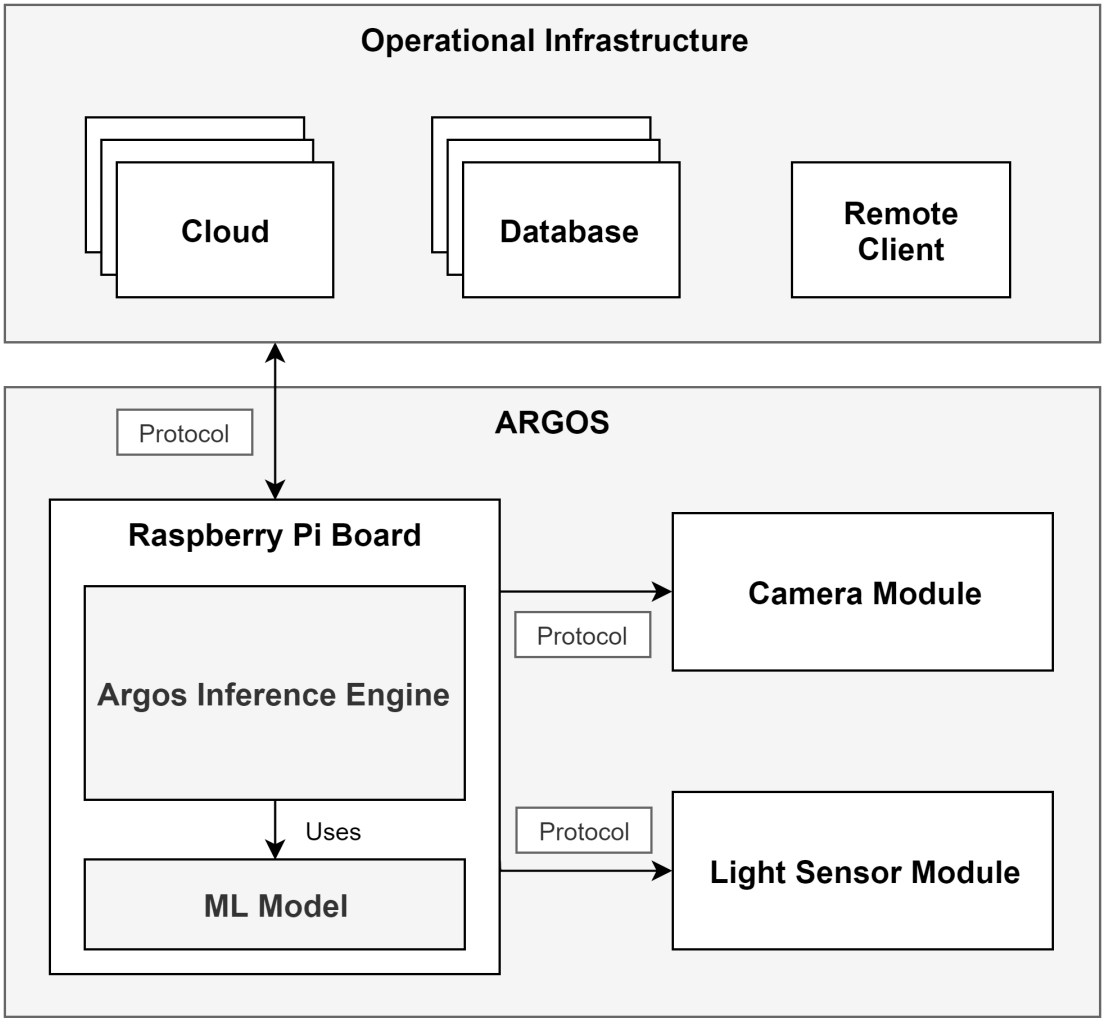


Figure A.1: System Overview Diagram Augmented

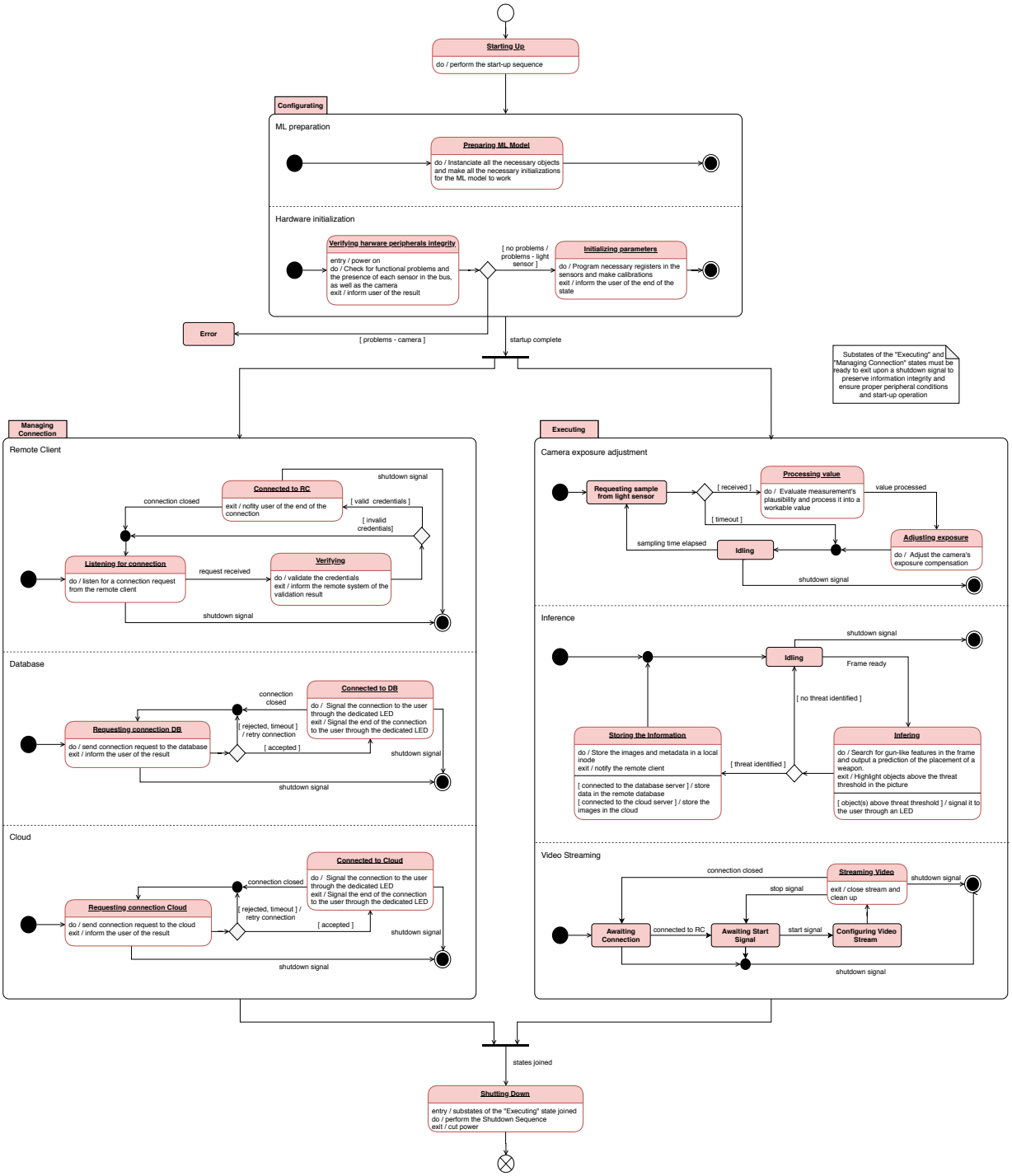


Figure A.2: State Machine Diagram - Computational Unit Augmented

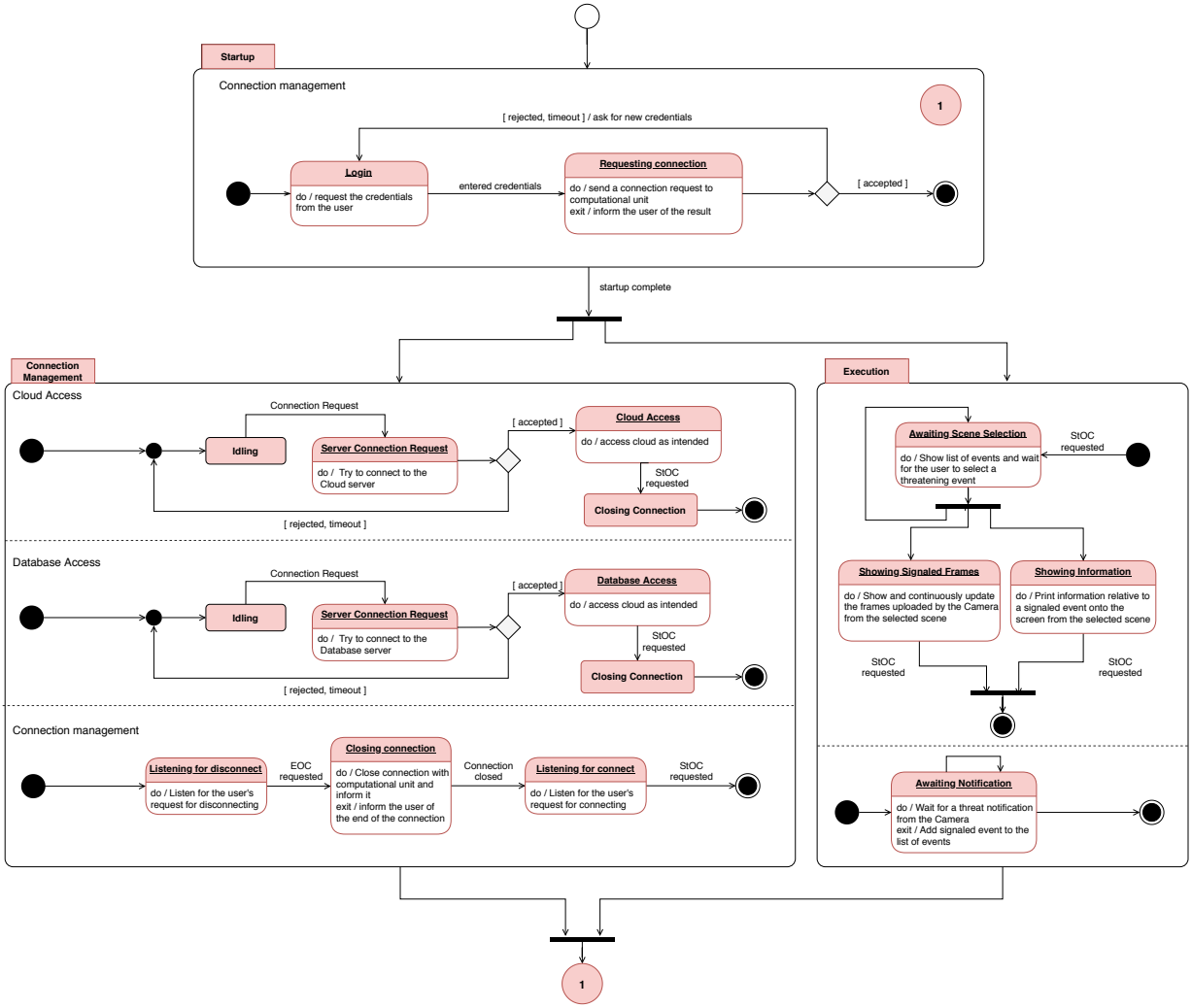


Figure A.3: State Machine Diagram - Remote Client Augmented

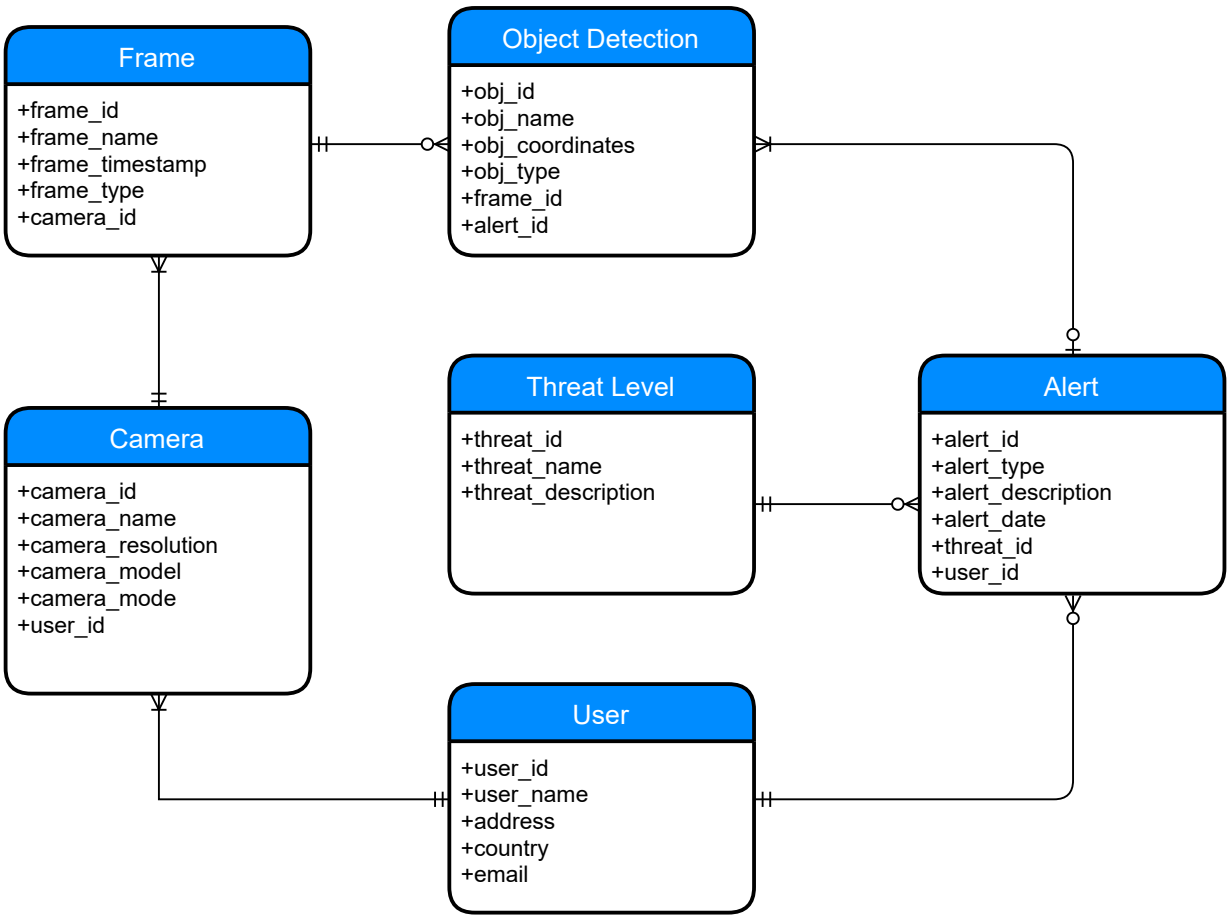


Figure A.4: ARGOS Database ER Diagram w/ Attributes Augmented

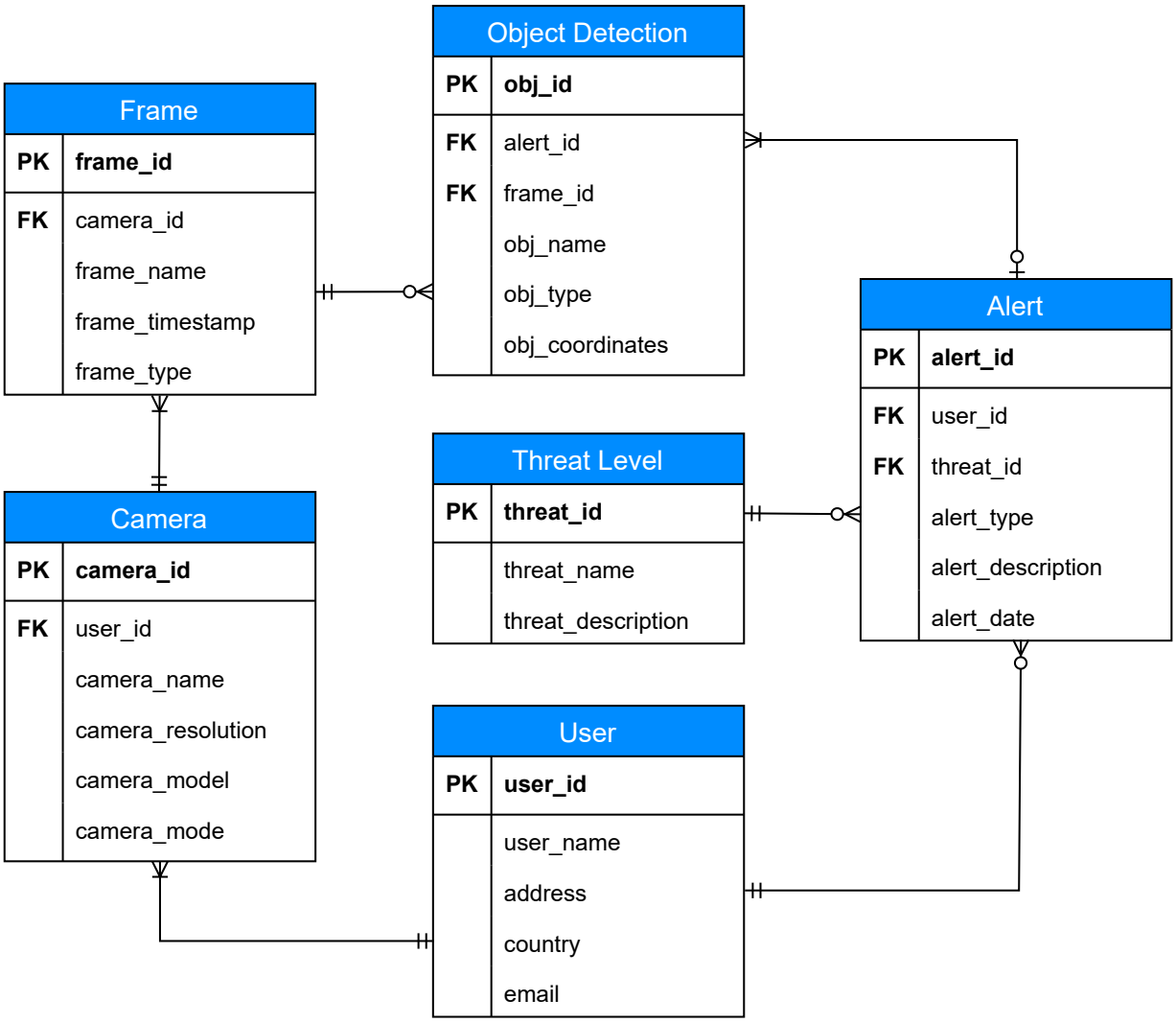


Figure A.5: ARGOS Database ER Logic Diagram Augmented

Appendix B

DLL Scripts

```
drop database if exists argos_db;
create database argos_db;
use argos_db;

drop table if exists users;
create table users (
        user_id int not null auto_increment primary key,
        username varchar(255),
        _address varchar(255) default null,
        email varchar(255) default null,
        country varchar(30) default null
);

drop table if exists cameras;
create table cameras (
        camera_id int not null auto_increment primary key,
        user_id int,
        camera_name varchar(255),
        camera_resolution varchar(5),
        camera_model(255),
        camera_mode(255),
        constraint user_fk foreign key (user_id) references users(user_id) on update cascade
);

drop table if exists frames;
create table frames (
        frame_id int not null auto_increment primary key,
        camera_id int,
        frame_name varchar(255),
        frame_timestamp timestamp,
        frame_link varchar(255),
        frame_type varchar(255),
        constraint camera_fk foreign key (camera_id) references cameras(camera_id) on update
             cascade
);

drop table if exists threat_lvls;
create table threat_lvls (
        threat_id int not null auto_increment primary key,
        threat_name varchar(255),
        threat_description varchar(255)
);
```

```
drop table if exists obj_detections;
create table obj_detections (
        obj_id int not null auto_increment primary key,
        alert_id int,
        frame_id int,
        obj_name varchar(255),
        obj_label varchar(255),
        obj_coords varchar(255),
        constraint alert_fk foreign key (alert_id) references alerts(alert_id) on update
            cascade,
        constraint frame_fk foreign key (frame_id) references frames(frame_id) on update
            cascade
);

drop table if exists alerts;
create table alerts (
        alert_id int not null auto_increment primary key,
        user_id int,
        threat_id int,
        alert_type varchar(255),
        alert_description varchar(255),
        alert_date timestamp,
        constraint user_fk foreign key (user_id) references users(user_id) on update cascade
            ,
        constraint threat_fk foreign key (threat_id) references threat_lvls(threat_id) on
            update cascade
);
```

Listing B.1: ARGOS Database and Table Creation

Appendix C

Extra Code

```
#ifndef _CAMERA_HPP_H
#define _CAMERA_HPP_H

#include <string>
#include <vector>
#include <linux/videodev2.h>

#define BUFFER_COUNT    4
#define FETCH_TIMEOUT   50

class Camera {

public:
        enum Encoding {
                MJPEG = V4L2_PIX_FMT_MJPEG,
                JPEG = V4L2_PIX_FMT_JPEG,
                MPEG = V4L2_PIX_FMT_MPEG,
                MPEG1 = V4L2_PIX_FMT_MPEG1,
                MPEG2 = V4L2_PIX_FMT_MPEG2,
                MPEG4 = V4L2_PIX_FMT_MPEG4,
                YUV420 = V4L2_PIX_FMT_YUV420,
                YUV422P = V4L2_PIX_FMT_YUV422P,
        };

        struct Format {
                uint32_t width;
                uint32_t height;
                enum Encoding encoding;

                Format(uint32_t width = 640, uint32_t height = 480,
                                enum Encoding encoding = \
                                    Encoding::MJPEG) {
                        this->width = width;
                        this->height = height;
                        this->encoding = encoding;
                }

                Format(const Format& fmt) {
                        this->encoding = fmt.encoding;
                        this->height = fmt.height;
                        this->width = fmt.width;
                }
```

```
        };

        struct Point2D {
                uint32_t x;
                uint32_t y;

                Point2D(uint32_t x, uint32_t y) {
                        this->x = x;
                        this->y = y;
                }
        };

        struct Color {
                uint8_t r;
                uint8_t g;
                uint8_t b;
                uint8_t a;

                Color(uint32_t a, uint32_t r = 0, uint32_t g = 0, \
                        uint32_t b = 0) {
                        this->a = a;
                        this->r = r;
                        this->g = g;
                        this->b = b;
                }
        };

        enum Error {
                OK=0,
                NO_DEVICE=-1,
                WRONG_DEVICE=-2,
                OP_NOT_PERM=-3,
                NOT_OPEN=-4,
                DEV_NOT_CAP=-5,
                INCOMP_ENCODING=-6,
                NO_MEM=-7,
                RESOURCE_UNAVAIL=-8,
                BAD_UNMAP=-9,
                CAPTURE_TIMEOUT=-10,
                NO_BUF_OUT=-11,
                DEV_NOT_OPEN=-12,
                INTERNAL=-13,
                UNDEF=-14,
        };

        struct Device {

                std::string name;
                int32_t descriptor;

                Device(std::string name = "", \
```

```
                    int32_t descriptor = -1) : \
                        name(name), descriptor(descriptor){
                }
        };

private:
        struct Buffer {
                void *start;
                size_t length;
        };

private:
        Buffer buffers[BUFFER_COUNT];

        Device dev;
        Format format;

        Error initialize();
        Error uninitialize();
        Error initializeMemoryMap();

        Error readFrame(v4l2_buffer& buf, std::string filename);

        bool streaming;

public:
        Camera(const std::string& device = std::string(""), \
            const Format& format = Format());
        ~Camera();

        Error openDevice(const std::string& device, \
            const Format& format, bool silent = false);
        bool isOpen();
        void closeDevice();

        Error start();
        bool isStreaming();
        Error stop();

        Error capture(std::string& filename);

        Error drawRectangle(std::string input_file,     \
            std::string output_file,                    \
            const Point2D& start, const Point2D& end,   \
            const Color& fill, const Color& stroke);

        Error drawText(std::string input_file,          \
            std::string output_file,                    \
                std::string text, const Point2D& start,     \
                uint32_t thickness,                         \
                uint32_t size, const Color& stroke);
```

```
};

#endif
```

Listing C.1: camera.hpp

```
#ifndef _LOG_HPP_H
#define _LOG_HPP_H

#include <iostream>
#include <sstream>
#include <string>

using namespace std;

namespace log {

        struct Logger {

        private:

                std::string prefix;
                std::ostream& stream;

        public:

                friend std::ostream& operator<<(const log::Logger& logger, const std::string
                    & str) {

                        logger.stream << logger.prefix << str;
                        return logger.stream;
                }

                friend std::ostream& operator<<(const log::Logger& logger, const char* str)
                    {

                        logger.stream << logger.prefix << str;
                        return logger.stream;
                }

                friend std::ostream& operator<<(const log::Logger& logger, int32_t num) {

                        logger.stream << logger.prefix << num;
                        return logger.stream;
                }

                friend std::ostream& operator<<(const log::Logger& logger, uint32_t num) {

                        logger.stream << logger.prefix << num;
                        return logger.stream;
                }
```

```
                friend std::ostream& operator<<(const log::Logger& logger, float num) {

                        logger.stream << logger.prefix << num;
                        return logger.stream;
                }

                Logger(std::string prefix, std::ostream& stream,
                                bool only_verbose = false)
                                : prefix(prefix), stream(stream) {

                }

        };

        const Logger   error("  [ERROR]: ", std::cerr);
        const Logger    info("   [INFO]: ", std::cout, true);
        const Logger warning("[WARNING]: ", std::cout);
        const Logger      ok("     [OK]: ", std::cout, true);
        const Logger details("\t[DETAILS]: ", std::cout);
        const Logger generic("",  std::cout);
};

#endif
```

Listing C.2: log.hpp

```
#ifndef _THREADS_HPP
#define _THREADS_HPP

#include <string>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/resource.h>

typedef struct sched_param sched_param;

typedef enum Priority_t {
    IDLE = 0,
    LOW = 15,
    BELOW_NORMAL = 30,
    NORMAL = 45,
    ABOVE_NORMAL = 60,
    HIGH = 75,
    REAL_TIME = 90
} Priority;
```

```
struct Thread{

    public:
        std::string name;
        pthread_t native;
        pthread_attr_t native_attr;

        sched_param thread_param;
        int thread_policy;
        int thread_status;

    public:
        struct Config {

                std::string name;

                uint32_t priority;
                void* args;
                void* (*start_routine) (void *);
            };
};

const Thread* createThread(const Thread::Config& config);

int32_t deleteThread (const Thread& thread);

#endif
```

Listing C.3: Threads: threads.hpp

```
# Import packages
import os
import argparse
from cv2 import cv2
import numpy as np
import sys
import glob
import importlib.util

root_dir = " ... "

# Define and parse input arguments
parser = argparse.ArgumentParser()
parser.add_argument('--modeldir', default='...')
parser.add_argument('--graph', default='.../detect.tflite')
parser.add_argument('--labels', default='.../labelmap.txt')
parser.add_argument('--threshold', default=0.5)
parser.add_argument('--image', default=None)
parser.add_argument('--imagedir', default='...')
```

```
args = parser.parse_args()
MODEL_NAME = args.modeldir
GRAPH_NAME = args.graph
LABELMAP_NAME = args.labels
min_conf_threshold = float(args.threshold)

# Parse input image name and directory.
IM_NAME = args.image
IM_DIR = args.imagedir

# Import TensorFlow libraries
# If tflite_runtime is installed, import interpreter from tflite_runtime, else import from
    regular tensorflow
pkg = importlib.util.find_spec('tflite_runtime')
if pkg:
    from tflite_runtime.interpreter import Interpreter
else:
    from tensorflow.lite.python.interpreter import Interpreter

# Get path to current working directory
CWD_PATH = os.getcwd()

# Define path to images and grab all image filenames
if IM_DIR:
    PATH_TO_IMAGES = os.path.join(CWD_PATH,IM_DIR)
    images = glob.glob(PATH_TO_IMAGES + '/*')

elif IM_NAME:
    PATH_TO_IMAGES = os.path.join(CWD_PATH,IM_NAME)
    images = glob.glob(PATH_TO_IMAGES)

# Path to .tflite file, which contains the model that is used for object detection
PATH_TO_CKPT = os.path.join(CWD_PATH,MODEL_NAME,GRAPH_NAME)

# Path to label map file
PATH_TO_LABELS = os.path.join(CWD_PATH,MODEL_NAME,LABELMAP_NAME)

# Load the label map
with open(PATH_TO_LABELS, 'r') as f:
    labels = [line.strip() for line in f.readlines()]

# Load the Tensorflow Lite model.
interpreter = Interpreter(model_path=PATH_TO_CKPT)

interpreter.allocate_tensors()

# Get model details
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
height = input_details[0]['shape'][1]
width = input_details[0]['shape'][2]
```

```
floating_model = (input_details[0]['dtype'] == np.float32)

input_mean = 127.5
input_std = 127.5

f = open(root_dir + "Desktop/model_output.txt", "w")
print("[OK]: Opened model output file")
index = 0
# Loop over every image and perform detection
for image_path in images:

    # Load image and resize to expected shape [1xHxWx3]
    image = cv2.imread(image_path)
    imH, imW, _ = image.shape
    image_resized = cv2.resize(image, (width, height))
    input_data = np.expand_dims(image_resized, axis=0)

    # Normalise pixel values if using a floating model (i.e. if model is non-quantized)
    if floating_model:
        input_data = (np.float32(input_data) - input_mean) / input_std

    # Perform the actual detection by running the model with the image as input
    interpreter.set_tensor(input_details[0]['index'],input_data)
    interpreter.invoke()

    # Retrieve detection results
    # Bounding box coordinates of detected objects
    boxes = interpreter.get_tensor(output_details[0]['index'])[0]
    # Class index of detected objects
    classes = interpreter.get_tensor(output_details[1]['index'])[0]
    # Confidence of detected objects
    scores = interpreter.get_tensor(output_details[2]['index'])[0]

    # Loop over all detections
    for i in range(len(scores)):
        if ((scores[i] > min_conf_threshold) and (scores[i] <= 1.0)):

            # Get bounding box coordinates and draw box
            ymin = int(max(1,(boxes[i][0] * imH)))
            xmin = int(max(1,(boxes[i][1] * imW)))
            ymax = int(min(imH,(boxes[i][2] * imH)))
            xmax = int(min(imW,(boxes[i][3] * imW)))

             # Look up object name from "labels" array using class index
            object_name = labels[int(classes[i])]
            label = '%s: %d%%' % (object_name, int(scores[i]*100))

            f.write(str(index) + " " + str(ymin) + " " + str(xmin) + " " + str(ymax) + " " +
                 str(xmax) + " " + str(label) +"\n")
```

```
    index = index + 1

f.close()
print("[OK]: Closed model output file")
```

Listing C.4: Inference Python script

```
from googleapiclient.http import MediaFileUpload
import pickle
import datetime
import os
from google_auth_oauthlib.flow import Flow, InstalledAppFlow
from googleapiclient.discovery import build
from google.auth.transport.requests import Request

def Create_Service(client_secret_file, api_name, api_version, *scopes):
    print(client_secret_file, api_name, api_version, scopes, sep='-')
    CLIENT_SECRET_FILE = client_secret_file
    API_SERVICE_NAME = api_name
    API_VERSION = api_version
    SCOPES = [scope for scope in scopes[0]]
    print(SCOPES)

    cred = None
    pickle_file = f'token_{API_SERVICE_NAME}_{API_VERSION}.pickle'

    if os.path.exists(pickle_file):
        with open(pickle_file, 'rb') as token:
            cred = pickle.load(token)

    if not cred or not cred.valid:
        if cred and cred.expired and cred.refresh_token:
            cred.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(CLIENT_SECRET_FILE, SCOPES)
            cred = flow.run_local_server()

        with open(pickle_file, 'wb') as token:
            pickle.dump(cred, token)

    try:
        service = build(API_SERVICE_NAME, API_VERSION, credentials=cred)
        print(API_SERVICE_NAME, 'service created successfully')
        return service
    except Exception as e:
        print(e)
        return None


def convert_to_RFC_datetime(year=1900, month=1, day=1, hour=0, minute=0):
    dt = datetime.datetime(year, month, day, hour, minute, 0).isoformat() + 'Z'
```

```
    return dt

root_dir = "..."
CLIENT_SECRET_FILE = root_dir + '.../Client_secret.json'
API_NAME = 'drive'
API_VERSION = 'v3'
SCOPES = ['https://www.googleapis.com/auth/drive']

service = Create_Service(CLIENT_SECRET_FILE, API_NAME, API_VERSION, SCOPES)

folder_id = '...'

file_names = ['out0.jpg', 'out1.jpg', 'out2.jpg', 'out3.jpg', 'out4.jpg', 'out5.jpg', 'out6.
    jpg',]
mime_types = ['image/jpeg', 'image/jpeg', 'image/jpeg', 'image/jpeg', 'image/jpeg', 'image/
    jpeg', 'image/jpeg']

media_dir = root_dir + ".../out/{0}"
for file_name, mime_type in zip(file_names, mime_types):
    file_metadata = {
        'name': file_name,
        'parents': [folder_id]
    }

    media = MediaFileUpload(media_dir.format(file_name), mimetype = mime_type)

    service.files().create(
        body=file_metadata,
        media_body=media,
        fields='id'
    ).execute()
```

Listing C.5: Cloud submission Python script

Appendix D

Thread Timelines

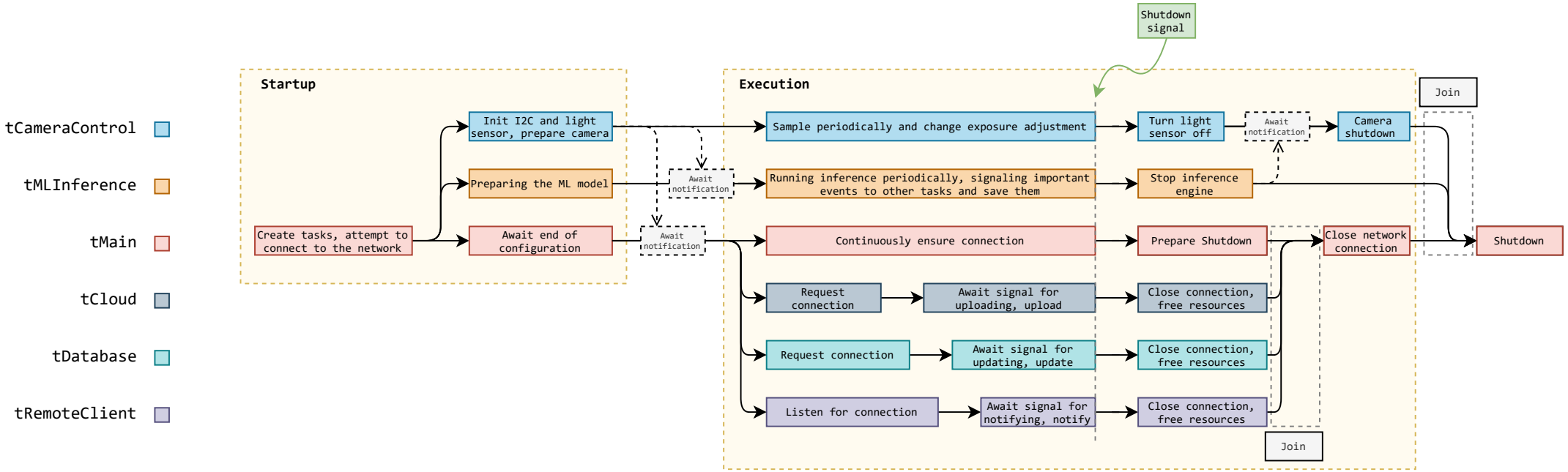


Figure D.1: Thread Timeline Augmented

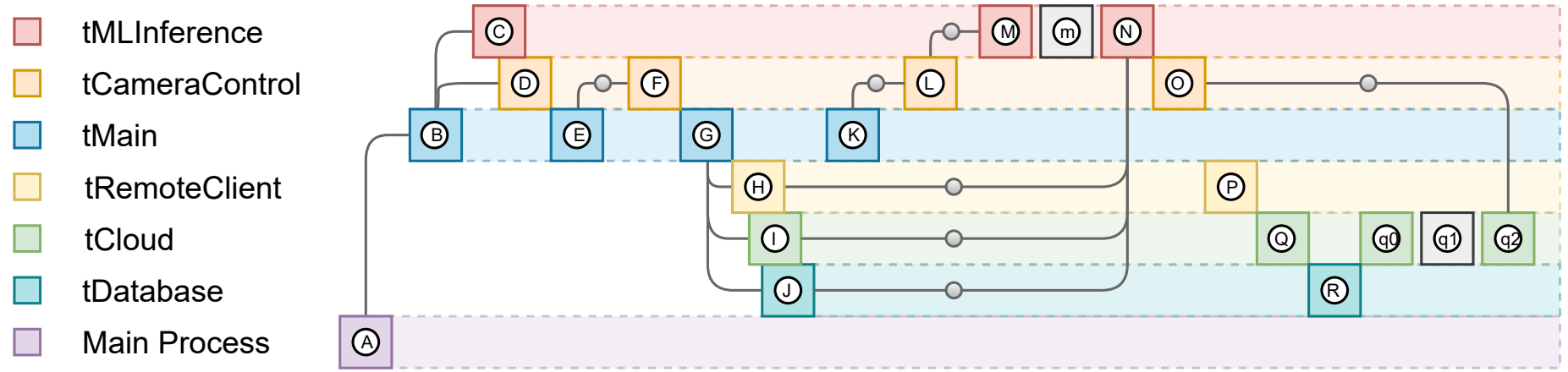
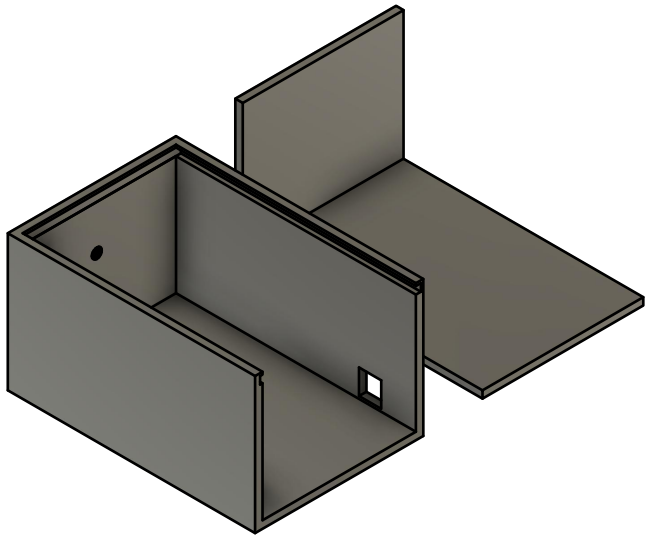


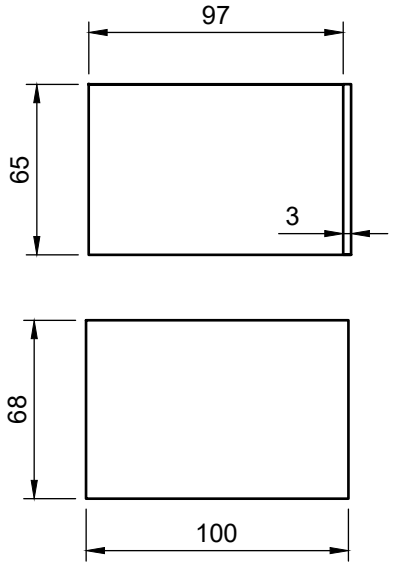
Figure D.2: Computational Unit Updated Thread Scheme Augmented

Appendix E

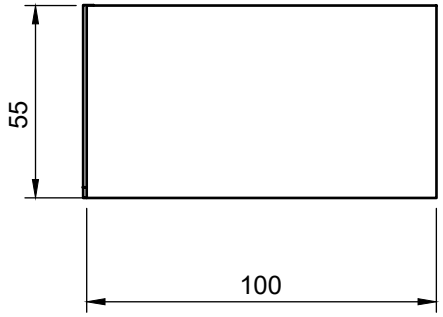
Technical Drawing

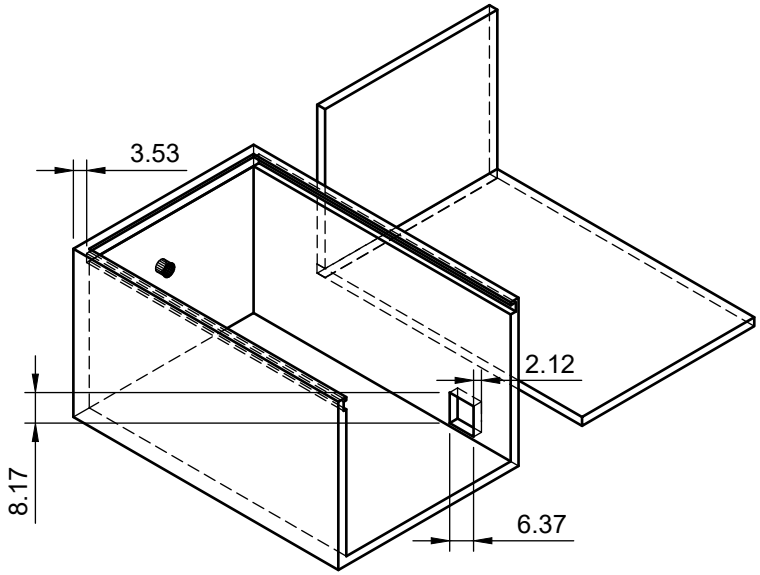


Bottom View

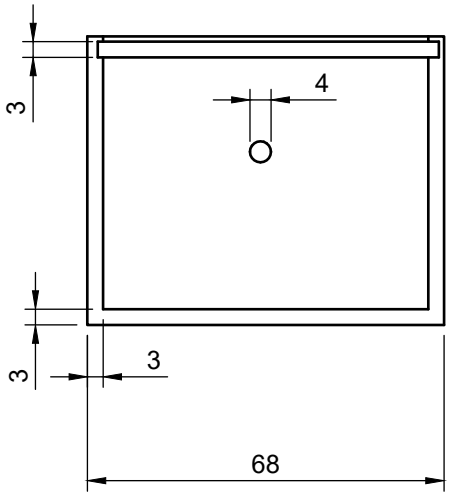


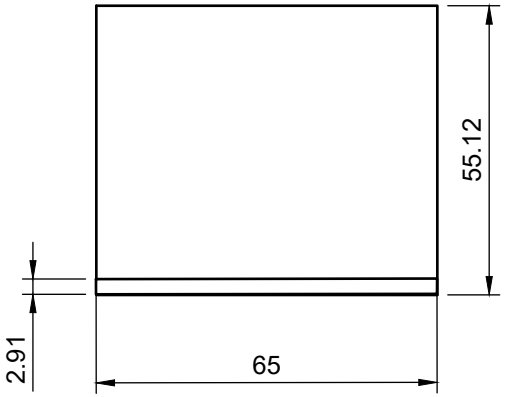
Side View





Back View





Dept. ESRG	Technical reference	Created by Group 7 25/11/2020	Approved by
		Document type Technical Drawing	Document status
		Title ARGOS	DWG No. 1
			Rev. 1 / Date of issue / Sheet 1/1

Appendix F

Project Planning

	Task Name	Duration	Start	ETA
1	**M0: Planning**			
1.1	Project Choice	2 days	15.10.20	16.10.20
1.2	Problem Statement	3 days	17.10.20	19.10.20
1.3	Project Plan	7 days	20.10.20	26.10.20
2	**M1: Analysis**			
2.1	System Overview	2 days	20.10.20	21.10.20
2.2	Requirements and constraints	1 day	22.10.20	22.10.20
2.3	System Architecture	2 days	23.10.20	24.10.20
2.4	System Analysis	2 days	25.10.20	26.10.20
2.5	Market Research	2 days	27.10.20	28.04.12
2.6	Estimated Budget	2 days	29.10.20	30.10.20
2.7	Diagrams	10 days	31.10.20	9.11.20
3	**M2: Design**			
3.1	Analysis Review	3 days	10.11.20	12.11.20
3.2	Hardware Specification	5 days	13.11.20	17.11.20
3.3	Component Shipping	14 days	18.11.20	7.12.20
3.4	Software Specification	10 days	18.11.20	27.11.20
3.5	ML Model Design and Training	8 days	28.11.20	5.12.20
4	**M3: Implementation**			
4.1	Hardware Testing	7 days	8.12.20	11.12.20
4.2	Software Implementation	32 days	1.12.20	31.12.20
4.3	Hardware Implementation	32 days	1.12.20	31.12.20
4.4	ML Model Conversion	5 days	1.12.20	5.12.20
5	**M4: Testing & Verification**			
5.1	Software Unit Testing	4 days	1.1.21	4.1.21
5.2	Software Integrated Testing	5 days	4.1.21	8.1.21
5.3	Hardwareware Unit Testing	4 days	1.1.21	4.1.21
5.4	Hardware Integrated Testing	5 days	4.1.21	8.1.21
5.5	Verification	3 days	9.1.21	11.1.21

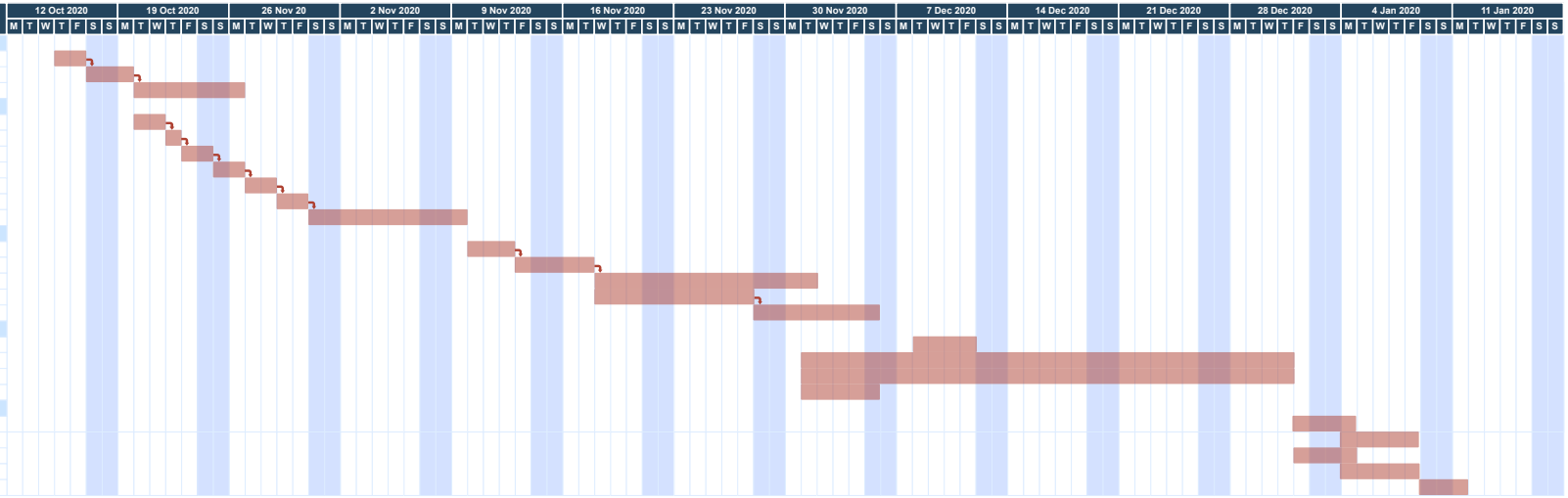


Figure F.1: Gantt Diagram