CAIRO UNIVERSITY

FACULTY OF ENGINEERING

ELECTRONICS AND ELECTRICAL COMMUNICATIONS DEPARTMENT

Advanced Driving Assistance System Using Embedded Linux

A Graduation Project Thesis Submitted to
The Faculty of Engineering at Cairo University
in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science
in
Electronics and Communications Engineering

Prepared By
Ahmed Osama Abdulmaksoud
Hussam Ali Ahmed
AbdelRahman Youssrey Fekry
Nour AbdelLatif
AbdelRahman Mohamed Farid

Supervised By

Prof. Neamat Abd-ElKader

Sponsored By

Valeo

Mentored By

Eng. Moatassem ElSayed

JULY 2023

ACKNOWLEDGMENT

In the Name of Allah, the Most Compassionate, the Most Merciful, we begin by expressing our gratitude to Allah for His blessings, guidance, and mercy throughout our academic journey. Without His help, we could not have achieved this milestone in our lives.

We would like to acknowledge our mentor and advisor, Prof. Neamat Abdel-Kader, for her mentorship, guidance, and support throughout this project. Her expertise has been invaluable to us, and we are grateful for her contributions.

We would like to acknowledge Cairo University for providing us with the opportunity to pursue our academic goals and for supporting us throughout our studies.

We would like to thank Valeo for their support and involvement in our project. Your collaboration and insights have been highly valuable and have contributed significantly to our research.

And We would like to specifically recognize Moatassem El Sayed, an engineer from Valeo, for his contributions to our project. His expertise and insights have been instrumental in the success of our research. We are grateful for his collaboration and support.

We pray that Allah blesses you all and rewards you for your time and effort.

Table of Contents

List of Figures

Chapter 1: Introduction

1.1 Abstract

Every year, several hundred people are killed due to road accidents, in fact, the National Safety Council estimated around 40,000 deaths and 4.5 million injuries only in the past few years. Unfortunately, it happens often that the driver doesn't have a long enough time frame to process the given information or take a decision making them responsible for more than 90% of road accidents. Even so if they do have a long time frame to take a decision some external factors might affect their judgment, and these factors include exhaustion from work, or drowsiness from a long trip which often ends up putting truck drivers on drugs just so they can stay focused, not paying proper attention to traffic lights/traffic signs which people always fail to acknowledge that they are only made for their safety, and not to forget, the remarkable decrease in the attention span due to the modern technology and the constant flow of information.

All these reasons were enough to suggest that maybe humans were not to be trusted completely to drive a 3000 pounds piece of metal on their own or that maybe with the modern technology that we have, driving is now beneath the human brain.

This is why car manufacturers are trying to develop technologies that make driving safer, smoother and more comfortable. This is precisely the interest of the ADAS "Advanced Driver Assistance Systems".

1.2 Brief History:

From the previous section, we can see the importance of ADAS, that it is not a delicacy or a quirk that can be ignored. So clearly, we were not the first to think of creating an ADAS, quite the contrary as ADAS has a relatively long history, with the first ADAS feature, ABS (Anti-lock Braking System), introduced in the 1970s. ABS prevents the wheels from locking up during hard braking, allowing the driver to maintain control of the vehicle and steer around obstacles.

In the 1990s, automakers began to introduce more sophisticated ADAS features, such as traction control, which prevents the wheels from spinning on slippery surfaces, and electronic stability control, which helps prevent skidding and loss of control during sudden maneuvers.

The 2000s saw the introduction of more advanced ADAS features, including adaptive cruise control, lane departure warning, blind-spot detection, and forward collision warning. These features use sensors and cameras to detect potential hazards on the road and warn the driver or take corrective action to avoid a collision.

Today, ADAS takes many shapes and forms and continues to evolve, with new features such as automatic emergency braking, pedestrian, traffic lights, traffic signs detection, and driver monitoring systems being introduced. As technology advances, we can expect to see even more sophisticated ADAS features in the future, making driving safer and more efficient.

1.3 Project description:

In this project, we create our own operating system that is based on Linux using a tool called Yocto (more on that in the following chapters). The reason behind this is to have the best performance possible for our applications to ensure the highest possible accuracy and allow us to run multiple applications without having many issues on the hardware of choice, which in our case is Raspberry Pi 4.

For the applications, we chose ones that the driver will find most beneficial and helpful, and these include:

1. GUI (graphical user interface)
2. Lane detection
3. Drowsiness detection
4. Traffic signs detection
5. Traffic lights detection
6. Pedestrian detection

Chapter 2: Embedded Linux

2.1 Types of Embedded Systems

In this section, we will explore the different types of embedded systems and compare the features of each to the needs of our project.

First, there is the traditional embedded systems or the one called, in layman's terms, bare metal. A bare metal device is a physical device that is completely dedicated to running a single dedicated application, which could for example be a thermostat control program. Interestingly enough, this is how computers worked back then before the days of PCs, only one program could be booted at the time and only one single application could run at the time. However, when we compare the capabilities of this system to the need of our application, we will find them lacking as we are expected to run more than one application that requires high processing and will need multiple threads and processes.

Next, there is RTOS or Real-Time Operating Systems. An RTOS has a relatively simple design, but, unlike Bare Metal, it can start and stop different processes concurrently and to do so, it does of course require a much stronger hardware to run a scheduler giving it a bit more overhead. The scheduler opens up the possibility of multi-threading that allows us to run some tasks concurrently, but RTOS is not as powerful as an OS (operating system). The most significant difference is usually memory protection and virtualization. Which makes RTOS a good candidate until we take into prospective the development time and the amount of low-level code that will be required.

All which takes us to Embedded Linux, or as some might call it, General purpose embedded system. This type of system requires a much capable microprocessor, with an MMU (memory management unit), and have access to RAM and external memory. Moreover, Embedded Linux has access to multi-threading and multi-processing and many common libraries making the code

more portable. And finally, it has access to high level languages and advanced features. Making it ideal for developing a project similar to ours.

In the next section, we will dive through Embedded Linux in a more detailed manner and discuss all of its components and details.

2.2 What is Embedded Linux?

Embedded Linux is a type of Linux operating system/kernel that is designed to be installed and used within embedded devices and appliances. In other words, it's a compact version of Linux that offers features and services in line with the operating and application requirement of the embedded system. Although it uses the same kernel, Embedded Linux is quite different from the standard operating system. First of all, it gets tailored for embedded systems and, therefore, is much smaller in size, requires less processing power, and has minimal features compared to that of a fully-fledged operating system. The Linux Kernel is modified and optimized as an embedded Linux version. Such a Linux instance can only run applications created specifically for the device.

Embedded Linux also offers its developers with several advantages over other systems such as:

1- Cross-Compilation for any supported platform.
2- Community reflection of common vulnerabilities and exposures (CVE) fixes in updated releases.
3- Deployment to commonly used Linux infrastructure and tools.
4- Modern, cloud-native environment.
5- Broad hardware support.
6- Productive lifecycle through community LTS.

2.3 Embedded Linux Architecture

At the most basic level, an Embedded Linux system is one that uses Linux as the operating system that sits between hardware and the application of an embedded device. There are five key components to an Embedded Linux system which we will go through, and these components are:

1- Bootloader.
2- Kernel.
3- Root filesystem.
4- Services.
5- Applications/Programs.

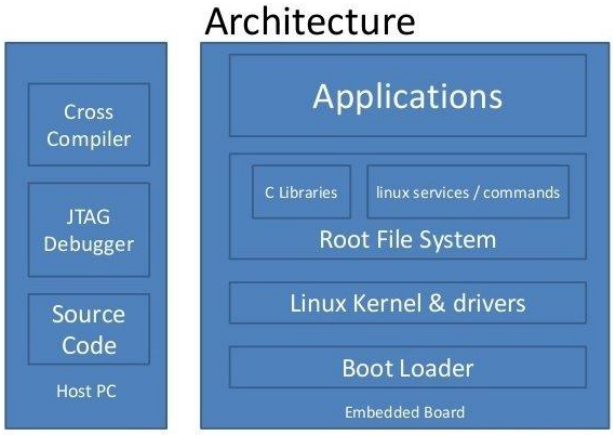

Figure 1 Embedded Linux Architecture

2.3.1 Bootloader

When the computer is powered on, after performing some initial setup, it will load a bootloader into memory and run that code. The bootloader's main job is to find the operating system's binary

program, load that binary into memory, and run the operating system, which in our case in the Linux kernel

The bootloader is done at this point, and all of its code and data in RAM are usually overwritten by the operating system. The bootloader won't run again until the computer is reset, or power cycled again.

The bootloader in embedded systems is different from a typical laptop, desktop or server computer. A typical PC usually boots into what we call the BIOS first and then runs GRUB as the bootloader. Embedded Linux systems boot using Das-UBoot or U-Boot for short as the bootloader.

Once the bootloader loads the Linux kernel into memory and runs it, the kernel will begin running its startup code. This code will be responsible for the initialization of the hardware, the system critical data structures, the scheduler, all the hardware drivers, the filesystem drivers, mount the first filesystem, and launch the first program, and more.

2.3.2 Kernel

The Linux kernel's main job is to start applications and provide coordination among these applications (or programs, as they are commonly called in Linux). The Linux kernel cannot identify all the programs that are supposed to run, so the Linux kernel starts only one program and lets that program launch all the other needed programs. And this very first program is none other than the init program, or sometimes referred to as just 'init'. Note that this first program doesn't need to be in a file called 'init', but it often is.

If the kernel for any reason cannot find the init program, the kernel's purpose is gone and the kernel crashes.

The main takeaway in the Linux kernel for embedded systems is that it is built to run on a different CPU architecture. Otherwise, the way the kernel operates is the same as the typical PC, which is one if its main advantages.

2.3.3 Root filesystem

In Linux, the kernel loads the programs into memory separately, and the kernel expects these programs to be stored on some medium organized into files and directories. This organization is what is known as a filesystem. Linux akin to many operating systems has filesystems on media, which is the data stored on a storage medium, and filesystem drivers, which is the code that knows how to interrupt and update the filesystem data on the medium, which is often a hardware device like SD cards, or even flash memory.

The Linux kernel works hand in hand with what is called the root filesystem. This is the filesystem upon which the root directory can be mounted, and which contains the files necessary to bring the system to a state where other filesystems can be mounted and user space daemons and applications started. The directory structure for a root filesystem can be extremely minimal, as we'll see in a moment, or it can contain the usual set of directories including /dev, /bin, /etc, and /sbin, among others that you see in any desktop Linux distribution. The kernel boot process concludes with the

init code (see init/main.c) whose primary purpose is to create and populate an initial root filesystem with a set of directories and files. It then tries to launch the first user mode process to run an executable file found on this initial filesystem. This first process ("init") is always given process ID 1. There are three ways for the kernel to find the file that will be run by the init process. The first method is to use a file specified at boot time with the init= kernel parameter. If this parameter is not set, the kernel tries a series of locations to find a file named "init". These include /sbin/init, /etc/init, and /bin/init. If all these fail, the kernel tries to run any shell it finds at /bin/sh. If this last fallback is not found, the kernel will print an error saying that no init could be found.

Unlike Windows, Linux filesystem do not get associated with drive letters, they do get associated with a directory. More so, filesystems can be associated with any directory, even ones that are several layers down in a path, and this association is called ‘mounting’. Linux first starts with an empty directory called / or slash, then during startup, the top most filesystem gets associated with (or in other words, mounted to) this directory, and all the contents of that filesystem appear under / or slash. This topmost filesystem is called the root filesystem.

Linux systems expect the root filesystem to be laid out a certain way. So, this filesystem is special and can’t just be some random set of directories and files. This is where directories like bin, sbin and more come from.

Because embedded systems have different hardware constraints, often Linux embedded systems use special filesystem formats rather than the typical EXT3, EXT4, btrfs, or xfs used on desktop or laptop computers.

2.3.4 Services

When the kernel finds, loads and runs the init program, that program then is responsible for bringing up the rest of the system. At this point, the kernel is no longer actively running and remains to coordinate the sharing of hardware among all the running programs.

There are several init programs available. Regardless of which init program is chosen, this program will launch all of the necessary services and applications that are needed for the system to be useful. This set of services includes setting up networking, mounting additional filesystems, setting up graphical environment, and more.

Under Linux, services are just programs that run in the background. These services were once known as daemon or daemon program, but recently this terminology became less popular.

2.3.5 Application/Program

Embedded Linux enables us to run programs in higher level language than that of bare metal embedded. Languages like python, C/C++, and Rust and the most common. The init program is responsible for starting these programs.

Embedded Linux is used to develop care software, and many other examples such as network equipment, machine control, industrial automation, navigation equipment, spacecraft flight software, and medical instruments in general. Even Microsoft Windows has Linux components as

part of the Windows Subsystem for Linux or WSL. But perhaps the best example of an Embedded Linux application is Android, developed by Google.

2.4 Embedded Linux vs. Desktop Linux

Table 1 Comparison between Embedded Linux and Desktop Linux

Embedded Linux	Desktop Linux
Linux kernel running in the embedded system product/single board computer/development board.	Linux kernel running on Desktop/Laptop.
Real time Linux kernel is used, making the response time real time or deterministic.	Linux kernel running in the desktop or laptop is not real time, the kernel response is not deterministic for response against events.
Kernel used in Embedded Linux is the customized version of the original kernel. User configures the kernel as per target processor, components present on the board, need of driver, etc.	Complete version of the kernel is used with all possible drivers and libraries. Whenever any new device or protocol is released then its driver patch is provided by either Linux community or by vendor.
Embedded Linux kernel footprint is less, around 1MBs.	Desktop Linus kernel footprint is more, around 100 MBs.

2.5 Embedded Linux vs. Bare metal vs. RTOS

Table 2 Embedded Linux vs Bare metal vs RTOS

Embedded Linux	Bare-metal	RTOS
Large overhead compared to other technologies due to scheduler, memory management, background tasks, etc.	Little to no software overhead.	Scheduler overhead.
Requires a microprocessor with a memory management unit (MMU) and an external RAM.	Low power requirement.	Requires a more powerful microcontroller.
Low direct control of hardware (files or abstraction layers).	High control of hardware.	High control of hardware
Multiple threads and processes.	Single-purpose or simple applications, hardware-dependent.	Multi-threading.
Multiple complex tasks, like networking, filesystem, graphical interface, etc.	Strict timing.	Multiple tasks, like networking, user interface, etc.

Chapter 3: Embedded Linux Development

3.1 Introduction to Embedded Linux development

Embedded Linux is a wide field as it can be seen from the previous chapters, hence an Embedded Linux developer has many goals, targets and responsibilities, all which can be divided to:

- Develop and maintain C/C++, python, or rust codes.
- Develop and maintain Linux device drivers
- Develop and maintain Linux environment either through Yocto (most common), or build root, or any other method.

In this Chapter we will discuss the development process of an Embedded Linux Image.

3.2 SD card structure in Embedded Linux

Before we dive any deeper, let's first talk about the structure of the SD card we will be using on our target device.

Recall the Embedded Linux elements we discussed earlier:

- Toolchain.
- Bootloader.
- Kernel + Device tree binary (DTB).
- Rootfs
- Application.

Each element should be placed in a specific manner, but first we need to part out own SD card into two partitions:

- Boot, which is a small FAT partition, will contain our bootloader, the kernel, device tree binary (DTB), and the configurations.
- Rootfs, which is a large EXT4 partition, will contain our directories (/user, /bin, etc.).

Now, we need to understand a very important term that we'll see very often not only when dealing with Embedded Linux, but with Linux in general, and that is 'Mount'.

The data we place on our SD card is stored in binary, so how are these ones and zeros transformed into actual information? The kernel contains a virtual file system (VFS) responsible for mapping binaries into readable data (.txt, .img … etc.). This process is called 'Mounting', where a row of data is transformed into readable data. If we were to try to access or read the data without mounting, the output data would be garbage values.

3.3 The Traditional Method for building a Customized Linux Image

Rather than the traditional bare metal embedded software development, where we write a C code, build, and produce a hex file, then burn the hex file on our target hardware), we create a Linux image that is flashed on the development board through an SD card.

The traditional method for building a Linux image is as it may sound, more of a primitive approach where we must, as they say, roll up our sleeves and do all the work by ourselves. The development process can be broken down to the following steps:

1- Study the board of choice, and specifically the booting process.
2- Build or download a toolchain.
3- Download a bootloader, which is a special piece of software with one sole purpose and that is to load the kernel and hand over the control to it (U-boot is the most popular one).
4- Build the kernel, which is the heart of the operating system.
5- Build the filesystem, depending on our configurations.
6- Build our application.
7- Plug and play.

3.3.1 Studying the booting sequence

In the previous chapters, we covered the booting sequence of Raspberry Pi with all its steps and elements.

3.3.2 Build or download a Toolchain

First, what is meant by a toolchain, a toolchain is a set of tools that compiles source code into executable that can run on our target device (includes a compiler, kernel headers, binutils, a linker, and run-time libraries).

In order to discuss the importance of the toolchain, we must first understand an important concept, and that is the difference between a cross compiler and a native compiler.

Let’s say that we have a c file called test.c, we compile it using GCC compiler to output test.exe that we run on our host machine. In this case, GCC is called a native compiler because we used it to generate code for the same platform on which it runs.

Say that for the same c file called test.c, we used arm.gcc to compile and output an executable that will run on an ARM target device. In this case, arm.gcc is called cross compiler because it was used to generate an executable code for a platform other than the one on which the compiler is running.

Although we could simply install the compiler on RPI using sudo apt-get but we don’t want to do this, we instead want to have/build our own toolchain for better customization. Of course, we are not going to build everything from scratch, but instead we will be using tools like crosstool-ng.

As mentioned above, we will be using tools like crosstool-ng to build a toolchain. Let’s see how this process will go.

1- Clone the repository.
2- Search for suitable configurations.
3- Set the default configurations according to the board we're using.
4- Apply all the necessary edits through menuconfig (Kconfig, which is a GUI that will provide us with helpful tools to output the .config file).
5- Build.

Note that the above sequence will be used for developing all elements, not just the toolchain.

3.3.3 Building a bootloader (U-boot)

Let's first take a look at U-boot or the Universal Boot Loader and understand what it is. It supports a wide range of microprocessors like MIPS, ARM, PPC, Blackfin, AVR32 and x86. It also supports different methods of booting, which is neat, it can support booting from USB, SD card, NOR and NAND flash (non-volatile memory). It can also boot Linux kernel from the network using TFTP. U-boot also provides a command line interface which gives us very easy access to it and many different things.

As we discussed in the previous chapters, U-boot is the second stage bootloader, responsible for loading the kernel and giving it control. When U-boot is compiled, we get u-boot.img.

To use U-boot on our target device, we will follow the same steps as in the toolchain. First, we will download the source code, which is of course open source and available on GitHub. We will then identify then specify the architecture we will be using, which can be found either through the target device's name or the SOC in the config file. Now that we have specified the architecture, we also need to identify what family it's under (for example, if we were to use this method, our architecture goes under the ARM family).

Running the make command followed by the architecture and family will output the .config file. The next step is to configure the U-boot configurations through menuconfig. After applying all necessary configurations, all that is left now is to build. So, we simply run the build command followed by the number of cores to utilize through the build process.

After building, we should now have the u-boot.bin file.

3.3.4 Building a Linux Kernel

First of all, a kernel can be a zimage or a uimage but it shouldn't really matter for us as all of them are just different ways to store.

Recall the function of a kernel in Embedded Linux:

- Runs and schedules different processes.
- Manages resources.
- Manages memory.
- Handles interrupts and multiusers.
- Provides filesystem, networking and security.

The kernel is, as one could imagine, a bunch of code (written in C) compiled to binary. This binary output which resembles the kernel doesn't have any information about the hardware configurations. Instead, these configurations can be found in the Device Tree Binary (DTB). Note that DTB files were at first stored as Device Tree Source files (DTS) which were then compiled using a Device Tree Compile (DTC).

We will follow the same sequence we used in the toolchain and bootloader. First, we will download the source code through Github, as usual. Then we will specify the target device architecture, the family, and enable the cross compiler, all through the build command. All that is left now is to apply all the necessary configurations through menuconfig, as we did before, and to finally build.

3.3.5 Building the Filesystem

Up till now, we have discussed building a toolchain, bootloader (U-boot), Linux kernel, and now we will talk about the filesystem. We will be following the same sequence we used to build the prior elements.

For the first step, instead of downloading some source code through Github, we will be installing a tool called BusyBox that will aid us in generating a root file system.

BusyBox is open-source and licensed under the GPL. BusyBox is a collection of core UNIX utilities packaged as a single binary. This makes it ideal for resource-constrained environments, which is just a fancy way to describe an embedded system. The complete distribution has almost 400 of the most common commands.

BusyBox input and output can be used in a standard manner across different distributions and versions of Linux. This helps keep the system configuration files compatible between different versions of Linux and allows easy updating of the system without having to learn new commands or make modifications. In addition, it can also help automate tedious tasks so less time is spent maintaining the system, saving valuable resources and time.

BusyBox provides the everyday convenience commands that often feel like they're part of your shell. Although userland tools like `ls` and `cat` are ubiquitous, they actually reside in a separate utility package that's independent of your shell. Many Linux distributions deliver these commands via GNU's coreutils but others ship BusyBox instead.

3.3.6 Building the Application

Embedded Linux can run a wide variety of programming languages, it can run Python, C/C++, rust, etc. It all comes down to what application we want to develop. Let's for example look at our case, where we had used the traditional method, we would be running our code on a cross compiler to produce an executable that we'll run on our target device. Of course, the cross compiler is going to differ according to the target device, as for Raspberry Pi, it will be arm cross compiler.

3.4 Using Yocto to build The Image

After exploring the traditional approach to customizing a Linux image, we will delve into the details of The Yocto Project, an alternative method of customizing a Linux image that we employed in our project.

Figure 2 Yocto Project Logo

The Yocto Project is a comprehensive open-source embedded Linux build system that enables the rapid development of custom embedded Linux distributions and related components. It provides an intuitive user interface to easily configure, create, and deploy these distributions to multiple target platforms. With its wide array of tools and libraries, it simplifies the implementation of complex multi-level layers for product specific customization and board support packages (BSPs). Furthermore, its adaptability helps manufacturers reduce time-to-market while increasing efficiency and extensibility in products with integrated Yocto components. By using packages such as Poky and OpenEmbedded Core (OE-Core), developers have access to preconfigured as well as customizable software stacks that can be used across multiple architectures including ARM, PowerPC, MIPS, x86 or x86_64 processors. That makes Yocto one of the most accepted open-source embedded build systems around.

The Yocto Project provides a compliant framework for creating customized Linux distributions for embedded and IoT devices. Its components include the OpenEmbedded-Core layer, which is a set of core recipes and classes to support complex image customization operations; the Poky layer, which offers technology to construct any type of Linux distribution; the BitBake build system which is responsible for the integration process of all other layers; Configurations files, used to interact effectively with multiple host systems to ensure easy cross-compilation; and Templates, providing helpful utilities and hooks enabling users to create very specific customizations. All said components allow the Yocto Project to offer developers advanced flexibility in building custom solutions for their applications in the embedded market. In the following paragraphs, we will delve deeper into the structure of the Yocto Project and all of its components.

Using reference templates, layers, build tools and machine configurations, it facilitates software integration activities across different hardware architectures. The project utilizes a shared metadata that allows developers to easily manage and customize the more than 3,000 electronic components supported by its ecosystem. Uniquely among Linux embedded platforms, Yocto provides developers with unparalleled flexibility across I/O devices, peripherals, and boards. In addition to established drivers and frameworks such as Linux kernel modules, multimedia codecs and hardware adaptation capabilities relying on board support packages features from 3rd party vendors; all aspects of the development environment can be tailored or extended according to specific requirements.

The following figure illustrates the Yocto Project Environment.

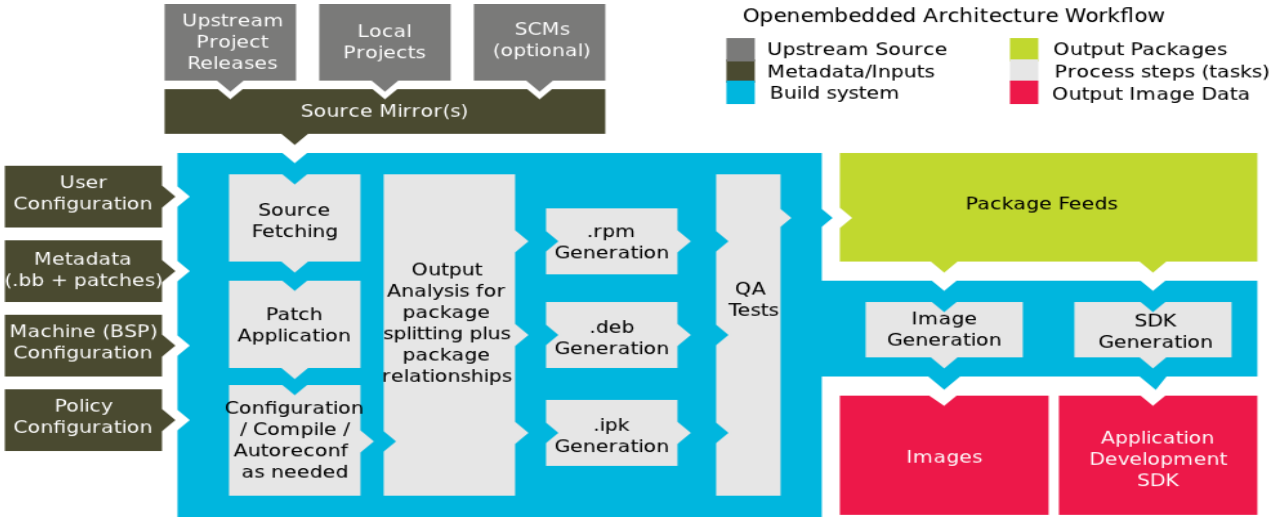

Figure 3 Yocto Project Environment

Now that we have untangled what the Yocto Project is, let's demonstrate the components and tools it's built upon.

4.4.1 Poky

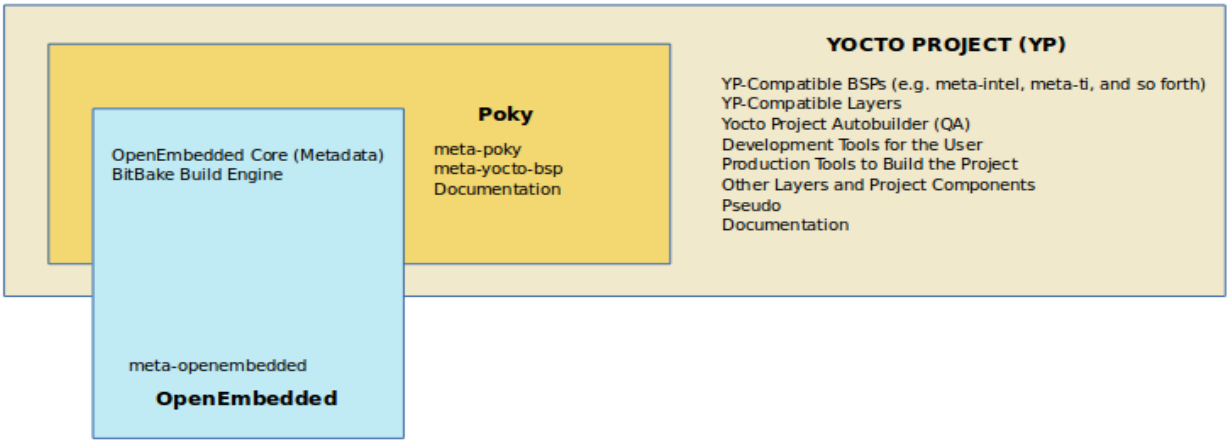

Figure 4 Poky Architecture

It is the reference distribution or the reference OS kit of the Yocto Project. It provides a standard set of metadata and tooling that serves as a foundation layer for other distributions and products in the project ecosystem. Poky includes routines to build Embedded Linux systems with cross-compiling support, as well as applications tailored to embedded environments such as BitBake and OE Core. While Poky is a "complete" distribution specification and is tested and put through QA, we cannot use it as a product "out of the box" in its current form.

3.4.2 Board Support Package (BSP)

It is a collection of packages and recipes that allow operating systems to be customized for embedded systems. The BSP package includes all necessary files for building a customized operating system for an embedded board, such as board-specific device support, graphics drivers, startup scripts, bootloaders, and kernel modules. Poky supports multiple BSPs by default like x86, Beaglebone, and Freescale platform.

3.4.3 Meta Data

It is basically the layers about a specific hardware or a software package. It abstracts away all the cross-compilation complexity and makes it easy for users to simply identify what packages are needed. This reduces development time by streamlining component selection, enabling faster delivery of products built around the Yocto Project technology stack. Meta data can be:

- Configurations Files (.conf) like Machine type, packages to be installed, etc.

- Recipes are the most common form of metadata. A recipe contains a list of settings and tasks (i.e. instructions) for building packages that are then used to build the binary image. A recipe describes where you get source code and which patches to apply. Recipes describe dependencies for libraries or for other recipes as well as configuration and compilation options. Related recipes are consolidated into a layer. Recipes (.bb or .bbappend) (BitBake) files are makefile-like recipes used within the Yocto Project to build packages. Each recipe file contains all the important information to build a component, including what source code is needed for the component, how it should be configured, and which dependencies must be satisfied before building it. Furthermore, bb files also contain instructions on how to package and install a component once it has been built. Additionally, since source code typically used in Yocto projects is often very complicated or infrequently used, bb files help developers focus on only the specific parts that really matter when dealing with them.

- Classes (.bbclass) files enable developers to create their own recipes using classes. Class files provide a means to share common code and configuration options between packages. This allows developers to reduce the number of lines of code they need to write, while also avoiding the duplication of code, which can lead to error-prone applications. Additionally, with class files, developers can quickly access settings and variables shared across multiple packages without having to look up each setting separately.

- Includes (.inc) files can be used to configure the kernel, add software packages, or modify system settings. Additionally, they can also store recipes in order to define how a specific package would be built during image generation.

3.4.4 Open-Embedded Core (OE-Core)

It provides developers with the necessary tools to build their own customized Linux distributions for embedded targets. OE-Core takes care of dependency management and cross compilation support. This ensures that platforms such as ARM can run sophisticated applications while still utilizing minimal system requirements. With its powerful yet easy to use feature set, Yocto's Open-Embedded Core makes using Linux on embedded devices easier than ever.

3.4.5 BitBake

It is a powerful tool used by the Yocto Project to support cross-platform, embedded Linux development. At its core, it takes project metadata and recipes as inputs, executes user commands and tasks, and outputs binary packages. Furthermore, it is a sophisticated build system that can handle dynamic runtime dependencies between tasks during execution via an internal dependency engine.

3.4.6 Layer

A collection of related recipes. Layers allow us to consolidate related metadata to customize our build. Layers also isolate information used when building for multiple architectures. Moreover, Layers are hierarchical in their ability to override previous specifications. We can include any number of available layers from the Yocto Project and customize the build by adding your layers after them. We can search the Layer Index for layers used within Yocto Project.

3.4.7 Packages

In the context of the Yocto Project, this term refers to a recipe's packaged output produced by BitBake (i.e., a "baked recipe"). A package is generally the compiled binaries produced from the recipe's sources. We "bake" something by running it through BitBake.

3.4.8 QEMU

It is an open-source virtual machine developed by the Linux Foundation in cooperation with Intel and is well-suited for use with the Yocto Project. By utilizing QEMU in an embedded system, it allows systems engineers a safe environment to conduct experiments without interfering with actual hardware or its real-time behavior. Simply, it acts as a simulator to run an image to test it and to know whether it is working or not and to know if there are any bugs.

To be able to use Yocto Project, Poky is downloaded to the Host Machine and its file structure contains:

Figure 5 Poky File Structure

- meta
  It is the Open-Embedded meta data.

- meta-poky
  It is the meta data of Yocto Project.

- meta-selftest
  Meta data used while building and testing the image, but it is not included in the image.

- meta-skeleton
  A template used as a layer example.

- meta-yocto-bsp
  It contains the BSPs supported by poky like x86, BeagleBone and FreeScale.

Chapter 4: Installing Yocto and Building the Image

4.1 Setting up the Yocto Environment

To use Yocto Project on our host machine, we must first make sure that the following requirements are met:

1. A host system with a minimum of 50 GB of free disk space running a supported Linux distribution (Ubuntu, Debian, Fedora, etc…)
2. The like git, gcc, python3, and more on the host which are essential for Yocto to generate the image.

For our project, we will be using Ubuntu 22.04 installed on Virtual Box.

The following command will install said packages based on an Ubuntu distribution:

```
$ sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit mesa-common-dev zstd liblz4-tool
```

Once the setup is complete, the next step is to install a copy of the Poky repository (which is a reference distribution of the Yocto Project that contains the OpenEmbedded Build System, BitBake, and OpenEmbedded Core, as well as a set of metadata to get you started building our own distro) on our build host using the following command:

```
$ git clone git://git.yoctoproject.org/poky
```

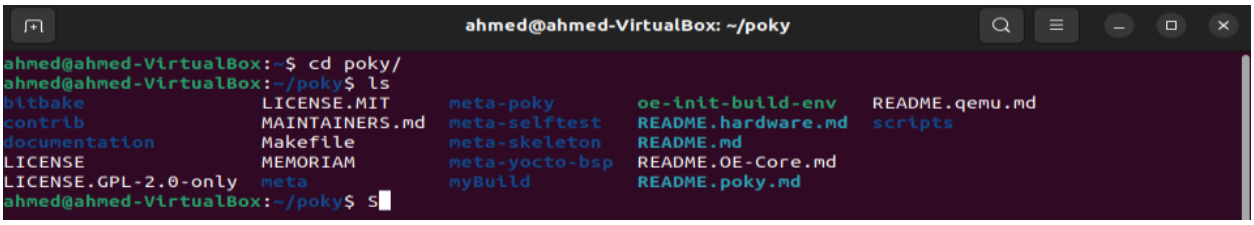

Figure 6 Poky Installation files

The last step in the installation process is to choose the release, which we will be working on, through their corresponding code name, each representing a branch from the Poky repository with its own support lifetime. For our project, we started by using Dunfell, then later, we had to switch to Kirkstone for reasons that will be discussed later.

By running the above commands, we now have set up our Yocto Project environment and are ready to create our own distro and layer.

4.2 Initialize the Build Environment

As mentioned above, Yocto can support any number of targeted builds for different embedded devices. Each of these builds however had to reside inside its own build directory. Yocto provides a script to set up and/or a user-specific build directory.

From within the poky directory, we run the oe-init-build-env environment setup script to define Yocto Project's build environment on our build host.

```
$ cd ~/yocto/poky
```

```
$ source oe-init-build-env myBuild
```

```
Oe-init-build-env
```

Is the environment setup script that will set up the local path and other variables for your build directory. `myBuild` will be the name of the build directory. Note that hadn't we specified the build directory name, it would have been set to `build` by default. Also note that we can source the script by also running `$ . oe-init-build-env myBuild`

The first call to the script with a new directory name will create the directory and its contents, all subsequent calls will only set the environment variables accordingly. We have to call source oe-

init-build-env once per terminal session, otherwise building and many other tools will not work. The source command runs the contents of the script as if typed onto the current shell. This includes directory changes, thus running the source oe-init-build-env script will leave you inside the build directory. Looking at the build directory we have, we'll find the following files:

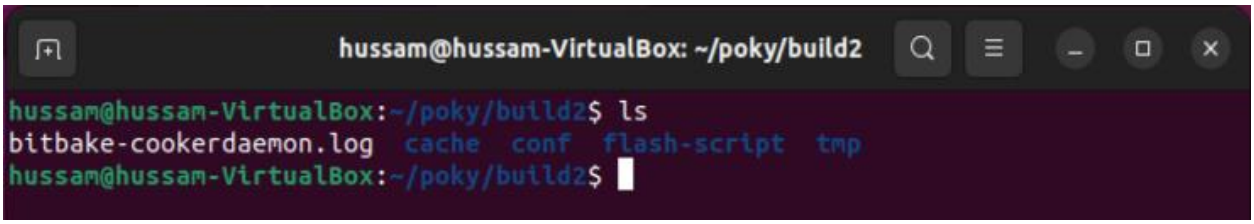

Figure 7 The Build Directory

Each file will be discussed in detail:

1) **Conf:** it contains the following 3 files:

Figure 8 Conf File Structure

- **Bblayers.conf:** before the OpenEmbedded build system can use our new layer, we need to enable it. And to enable it, we simply add our layer's path to the BBLAYERS variable in our conf/bblayers.conf file.

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""


BBLAYERS ?= " \
  /home/ahmed/poky/meta \
  /home/ahmed/poky/meta-poky \
  /home/ahmed/poky/meta-yocto-bsp \
  /home/ahmed/poky/meta-mylayer \
  /home/ahmed/poky/meta-raspberrypi \
  /home/ahmed/poky/meta-openembedded/meta-oe \
  /home/ahmed/poky/meta-openembedded/meta-python \
  /home/ahmed/poky/meta-openembedded/meta-networking \
  /home/ahmed/poky/meta-openembedded/meta-multimedia \
  /home/ahmed/poky/meta-atmel \
  "
```

Figure 9 bblayers,conf File Contents

Layers can also be added through a bitbake command:

```
$ bitbake-layers add-layer
```

- **Local.conf:** is a powerful tool that can configure almost every aspect of the building process. Through local.conf we can set the following. Tt contains the used bit-bake layers in

bblayers.conf and the build configurations like machine type, package classes, configuration version and many more in local.conf.

a. Set the machine which we will be working on (raspberrypi3 in our case). There is a various selection of emulated machine available beside the QEMU emulator.

b. Set where to place downloads directory. As during the first build the system will download many different source code tarballs from various upstream projects. This can take a while, particularly if the network connection is slow. These are all stored in DL_DIR. When wiping and rebuilding we can preserve this directory to speed up this part of subsequent builds. This directory is safe to share between multiple builds on the same machine too.

c. Set where to place the shared-state files directory. Since BitBake has the capability to accelerate builds based on previously built output. This is done using "shared state" files which can be thought of as cache objects and this option determines where those files are placed. We can wipe out TMPDIR leaving this directory intact and the build would regenerate from these files if no changes were made to the configuration. If changes were made to the configuration, only shared state files where the state was still valid would be used (done using checksums).

d. Set where to place the build output (tmp file). This option specifies where the bulk of the building work should be done and where BitBake should place its temporary files and output. Keep in mind that this includes the extraction and compilation of many applications and the toolchain which can use Gigabytes of hard disk space.

e. Set the distribution settings. The distribution setting controls which policy settings are used as defaults. The default value (Poky) is fine for general Yocto project use, at least initially. Ultimately when creating custom policy, people will likely end up sub classing these defaults.

f. Enable/Disable package management configurations.

g. Add extra packages to the generated images through `EXTRA_IMAGE_FEATURES`.

h. QEMU configurations in case we're using QEMU.

i. Optimization options for maximum build time.

In addition to many other features, these were the main ones.

The following figures shows what's inside the local.conf file:

```
# This sets the default machine to be qemux86-64 if no other machine is selected:
MACHINE ="raspberrypi3"

# The default is a downloads directory under TOPDIR which is the build directory.
DL_DIR ?= "${TOPDIR}/downloads"

# The default is a sstate-cache directory under TOPDIR.
SSTATE_DIR ?= "${TOPDIR}/sstate-cache"


# The default is a tmp directory under TOPDIR.
#TMPDIR = "${TOPDIR}/tmp"

#setting the distribution settings
DISTRO ?= "mydistro"

# Package Management configuration

# We default to rpm:
PACKAGE_CLASSES ?= "package_rpm"

#
# SDK target architecture

#SDKMACHINE ?= "i686"

#
# Extra image configuration defaults

# We default to enabling the debugging tweaks.
EXTRA_IMAGE_FEATURES ?= "debug-tweaks"

#IMAGE_FSTYPES += "wic.xz"

#
# Additional image features

#   - 'buildstats' collect build statistics
USER_CLASSES ?= "buildstats"

#
# Runtime testing of images

#TESTIMAGE_AUTO:qemuall = "1"

#
# Interactive shell configuration

# By default disable interactive patch resolution (tasks will just fail instead):
PATCHRESOLVE = "noop"

#
# Disk Space Monitoring during the build
#
```

Figure 10 local.conf File Contents 1

```
BB_DISKMON_DIRS ??= "\
    STOPTASKS,${TMPDIR},1G,100K \
    STOPTASKS,${DL_DIR},1G,100K \
    STOPTASKS,${SSTATE_DIR},1G,100K \
    STOPTASKS,/tmp,100M,100K \
    HALT,${TMPDIR},100M,1K \
    HALT,${DL_DIR},100M,1K \
    HALT,${SSTATE_DIR},100M,1K \
    HALT,/tmp,10M,1K"

#
# Shared-state files from other locations

#SSTATE_MIRRORS ?= "\
#file://.* https://someserver.tld/share/sstate/PATH;downloadfilename=PATH \
#file://.* file:///some/local/dir/sstate/PATH"

#
# Yocto Project SState Mirror

#BB_HASHSERVE_UPSTREAM = "hashserv.yocto.io:8687"
#SSTATE_MIRRORS ?= "file://.* http://sstate.yoctoproject.org/all/PATH;downloadfilename=PATH"

#
# Qemu configuration

#ASSUME_PROVIDED += "libsdl2-native"

# You can also enable the Gtk UI frontend, which takes somewhat longer to build, but adds
# a handy set of menus for controlling the emulator.
#PACKAGECONFIG:append:pn-qemu-system-native = " gtk+"

#
# Hash Equivalence

#BB_HASHSERVE = "auto"
#BB_SIGNATURE_HANDLER = "OEEquivHash"

#
# Memory Resident Bitbake
server will shut down.
#
#BB_SERVER_TIMEOUT = "60"

# CONF_VERSION is increased each time build/conf/ changes incompatibly and is used to
# track the version of this file when it was generated. This can safely be ignored if
# this doesn't mean anything to you.
CONF_VERSION = "2"
LICENSE_FLAGS_ACCEPTED = "commercial"
```

Figure 11 local.conf File Contents 2

2) **Downloads:** it contains the downloaded files and packages that will be installed to the custom image.

3) **Sstate-cache:** it contains the shared state cache. It will be used later to act as a base for multiple builds. This directory can be moved to another location and specify in the build configuration of the image where it is through the SSTATE_DIR variable.

4) **Tmp:** The OpenEmbedded build system creates and uses this directory for all build system's output. BitBake creates this directory if it does not exist. As a last resort, to clean up a build and start it from scratch (other than the downloads), we can remove everything in the tmp directory or get rid of the directory completely. Note that if we do so, we should also completely remove the state-cache. tmp also contains the following sub directories:

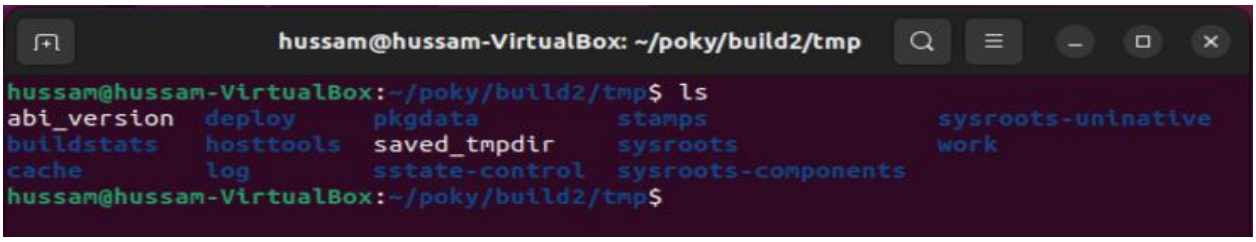

Figure 12 Contents of tmp Folder

We go through each file:

- **buildstats:** stores the build statistics.

- **cache:** after bitbake parses the metadata (recipes and configuration files), it caches the results tmp/cache to speed up future builds. During subsequent builds, Bitbake checks each recipe (together with, for example, any files included or appended to it) to see if they have been modified. Changes can be detected, for example, through file modification time (mtime) changes and hashing of file contents. If no changes to the file are detected, then the parsed result stored in the cache is reused. If the file has changed, it is reparsed.

- **deploy:** This directory contains any "end result" output from the OpenEmbedded build process. The DEPLOY_DIR variable points to this directory.

- deb: This directory receives any .deb packages produced by the build process. The packages are sorted into feeds for different architecture types.

- rpm: This directory receives any .rpm packages produced by the build process. The packages are sorted into feeds for different architecture types.

- ipk: This directory receives .ipk packages produced by the build process.

- licenses: This directory receives package licensing information. For example, the directory contains sub-directories for bash, busybox, and glibc (among others) that in turn contain appropriate COPYING license files with other licensing information.

- Images: This directory is populated with the basic output objects of the build (think of them as the "generated artifacts" of the build process), including things like the boot loader image, kernel, root filesystem, and more. If we want to flash the resulting image from a build onto a device, look here for the necessary components. Note that when deleting files in this directory. We can safely delete old images from this directory (e.g. core-image-*). However, the kernel (*zImage*, *uImage*, etc.), bootloader, and other supplementary files might be deployed here prior to building an image. Because these files are not directly produced from the image, if we delete them they will not be automatically re-created when we build the image again.

- sdk: The OpenEmbedded build system creates this directory to hold toolchain installer scripts which, when executed, install the sysroot that matches your target hardware.

- sstate-control: The OpenEmbedded build system uses this directory for the shared state manifest files. The shared state code uses these files to record the files installed by each sstate task so that the files can be removed when cleaning the recipe or when a newer version is about to be installed. The build system also uses the manifests to detect and produce a warning when files from one task are overwriting those from another.

- sysroots-components: This directory is the location of the sysroot contents that the task do_prepare_recipe_sysroot links or copies into the recipe-specific sysroot for each recipe listed in DEPENDS. Population of this directory is handled through shared state, while the path is specified by the COMPONENTS_DIR variable. Apart from a few unusual circumstances, handling of the sysroots-components directory should be automatic, and recipes should not directly reference build/tmp/sysroots-components.

- sysroots: Previous versions of the OpenEmbedded build system used to create a global shared sysroot per machine along with a native sysroot.

- stamps: This directory holds information that BitBake uses for accounting purposes to track what tasks have run and when they have run. The directory is sub-divided by architecture, package name, and version.

- work: This directory contains architecture-specific work sub-directories for packages built by BitBake. All tasks are executed from the appropriate work directory. For example, the source for a particular package is unpacked, patched, configured, and compiled all within its own work directory. Within the work directory, organization is based on the package group and version for which the source is being compiled as defined by the WORKDIR.

- work-shared: For efficiency, the OpenEmbedded build system creates and uses this directory to hold recipes that share a work directory with other recipes. In practice, this is only used for gcc and its variants (e.g. gcc-cross, libgcc, gcc-runtime, and so forth).

4.3 Building the minimal image using QEMU

The OpenEmbedded build system provides several example images to satisfy different needs. When we issue the bitbake command and provide a “top-level” recipe that essentially begins the build for the type of image wanted.

From within the Poky Git repository, we can use the following command to display the list of directories within the source directory that contain image recipe files.

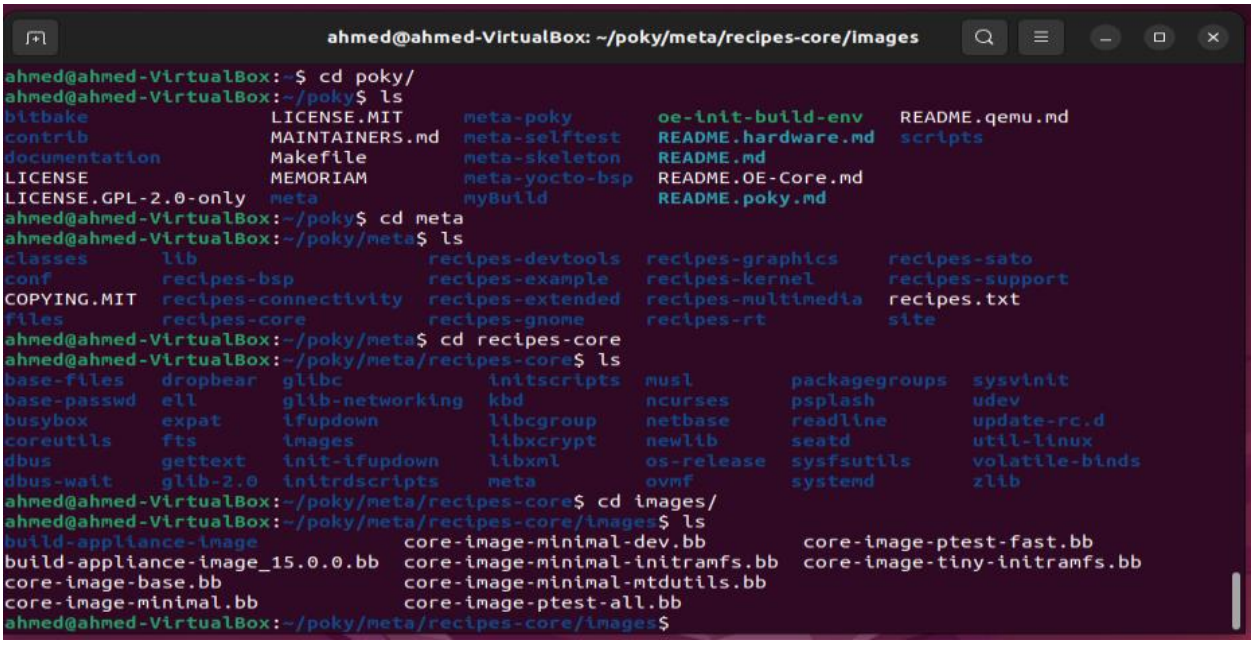

Figure 13 List of Core Images provided by Poky

For this example, we will be building core-image-minimal, which is a small image, just enough for allowing a device to boot. To do so, we go to our Poky Git repository, source our build environment through the command

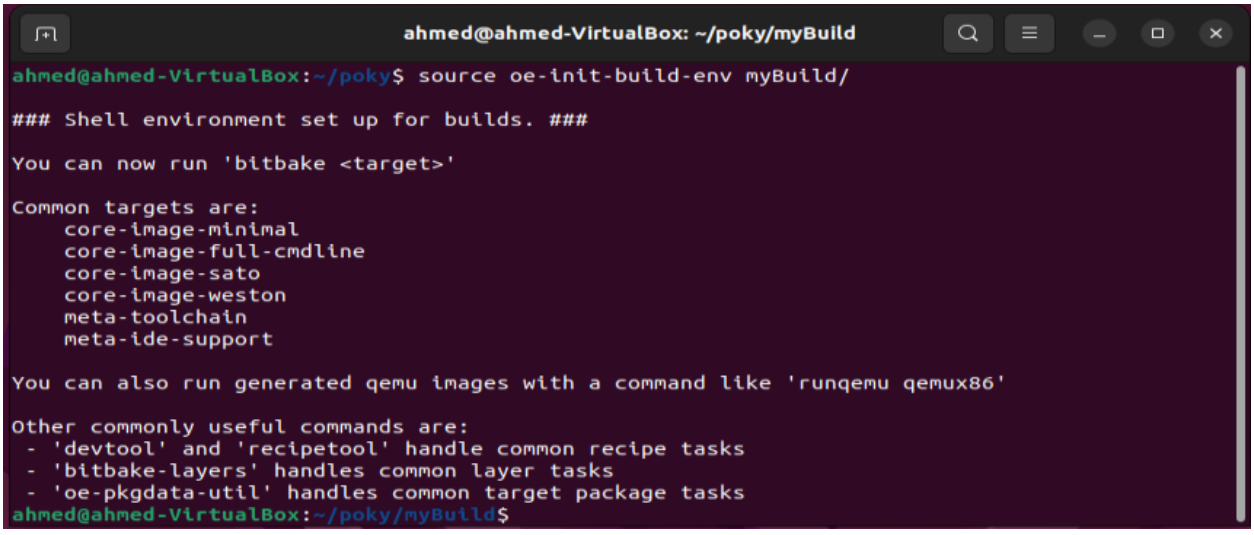

Figure 14 Sourcing Yocto Environment

Then we finally run bitbake.

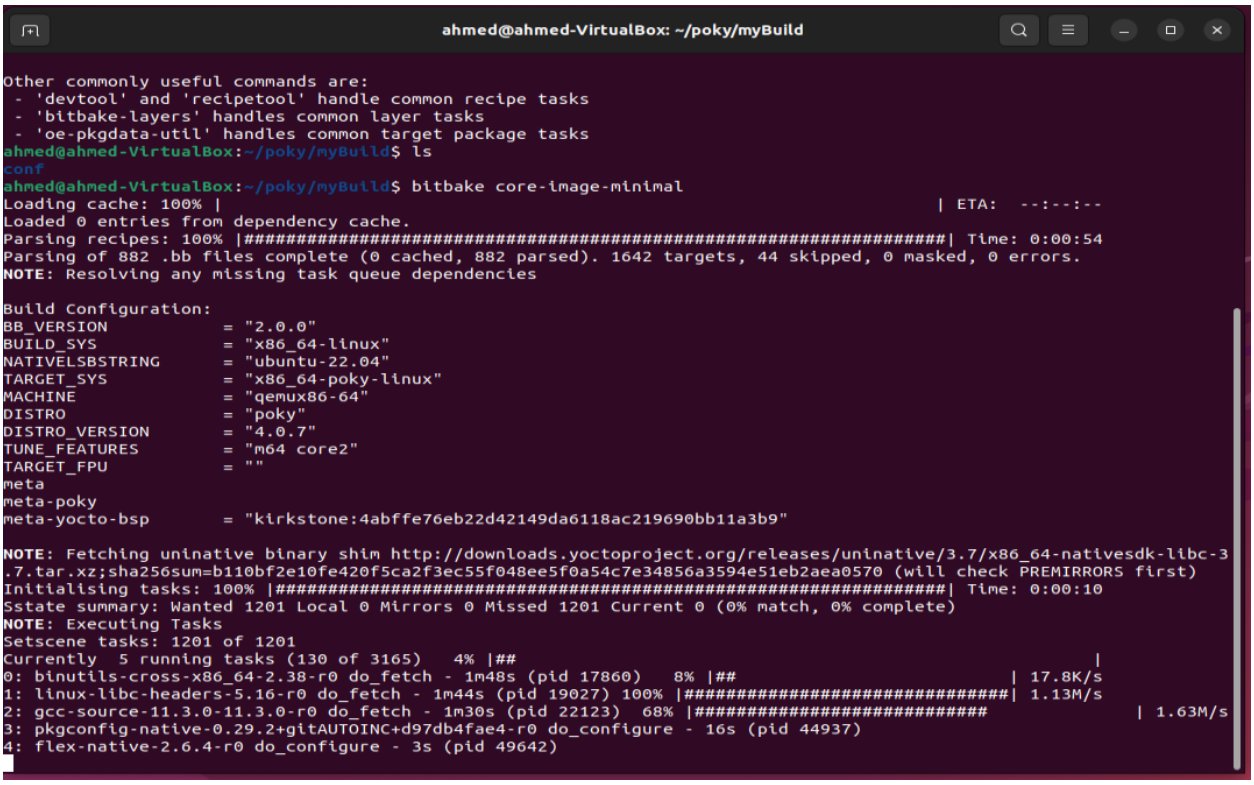

Figure 15 Bitbaking the recipes

Let's discuss what QEMU is. QEMU is a generic and open-source machine emulator and virtualizer. When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). Now, let's run the image using QEMU.

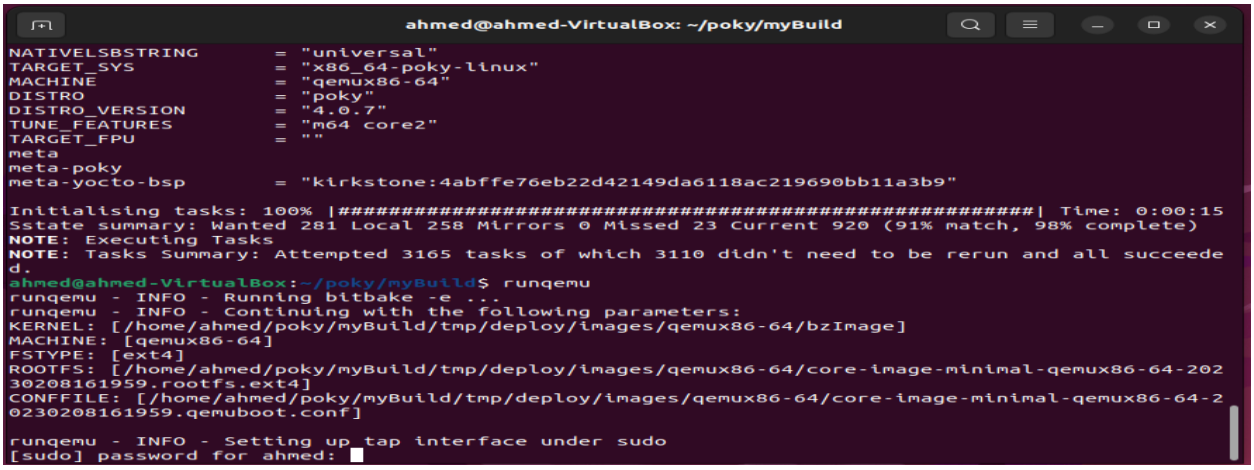

Figure 16 Running QEMU

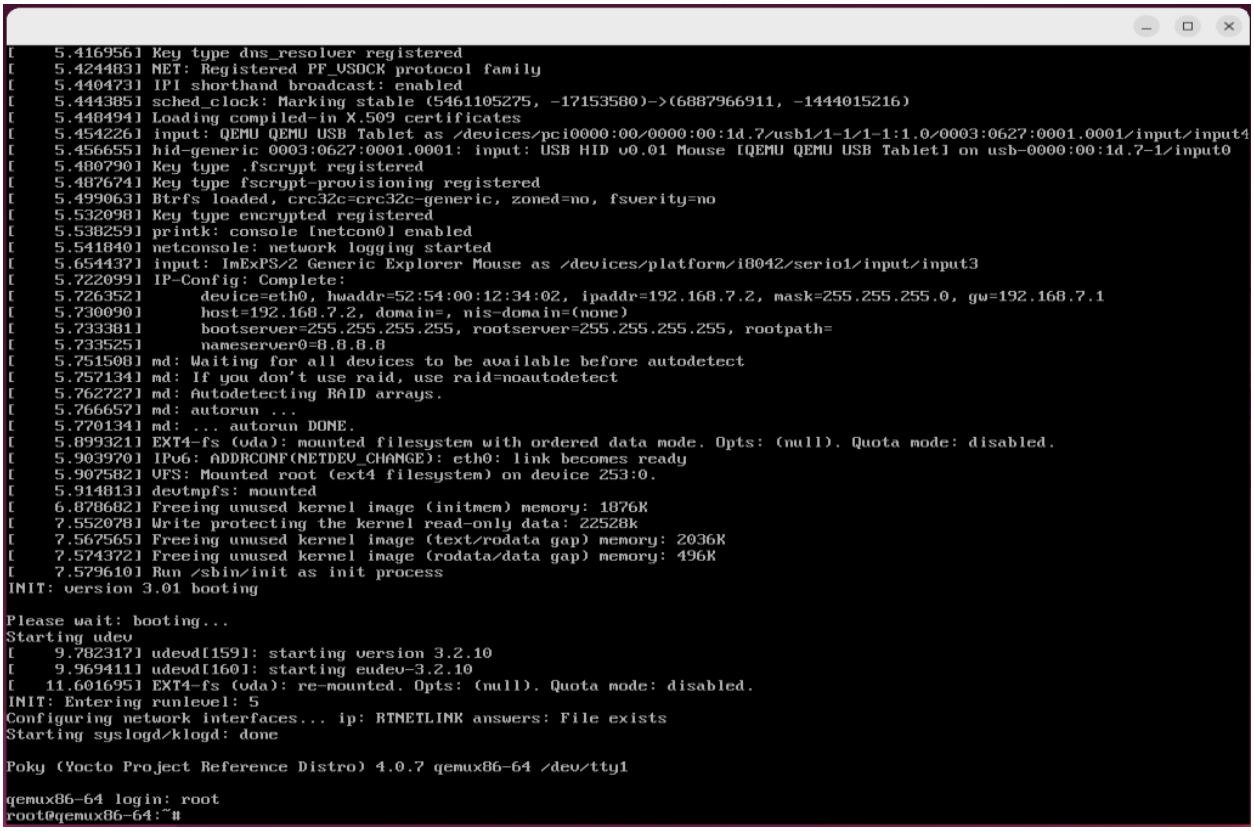

Figure 17 QEMU on action

4.4 Creating our own Layer

Instead of adding the required packages and applications to our image in local.conf file, present the build directory, which is not considered a good practice. As by doing so, any image will be built with these added packages which is not what we want, we will create a layer of our own to be able to add specific packages and applications to it. Whenever an image is built, we can add this layer to the bblayers.conf file which includes all the added layers to the image. This way the added packages and applications are abstracted so that whenever they are needed, we can just include it in the bblayers.conf file.

There are various ways to create a layer:

- The manual way is to use the meta-skeleton folder which acts like an example of a new layer and provides some examples of the recipes to be used.
- The automated way is to use the bitbake to automatically create a layer:

```
$ bitbake-layers create-layer meta-mylayer
```

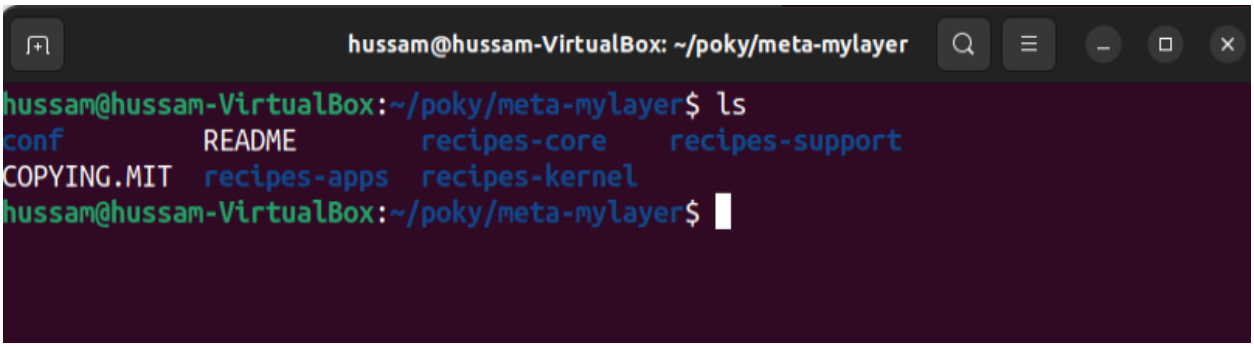

Figure 18 Contents of our custom layer

We used the automated way to create our own layer. This layer contains:

- Multiple recipes which are organized so that each set of recipes with the same description or configuration are in one folder as:
    - recipes-core: It contains recipes related to the image.
    - recipes-kernel: It contains recipes related to the kernel.
    - recipes-apps: It contains recipes that are used to build our custom-made applications.
    - recipes-support: It contains some recipes which add some configuration to an already made recipes inside another layer.
- conf folder: It contains the layer configurations and our own custom-made distribution.
    - layer.conf: This file contains the layer configurations like the compatible Yocto releases, the location of the recipes (.bb) files and some other configurations.
    - distro folder: This folder contains the recipes that is responsible for the custom-made distro which will be discussed later.

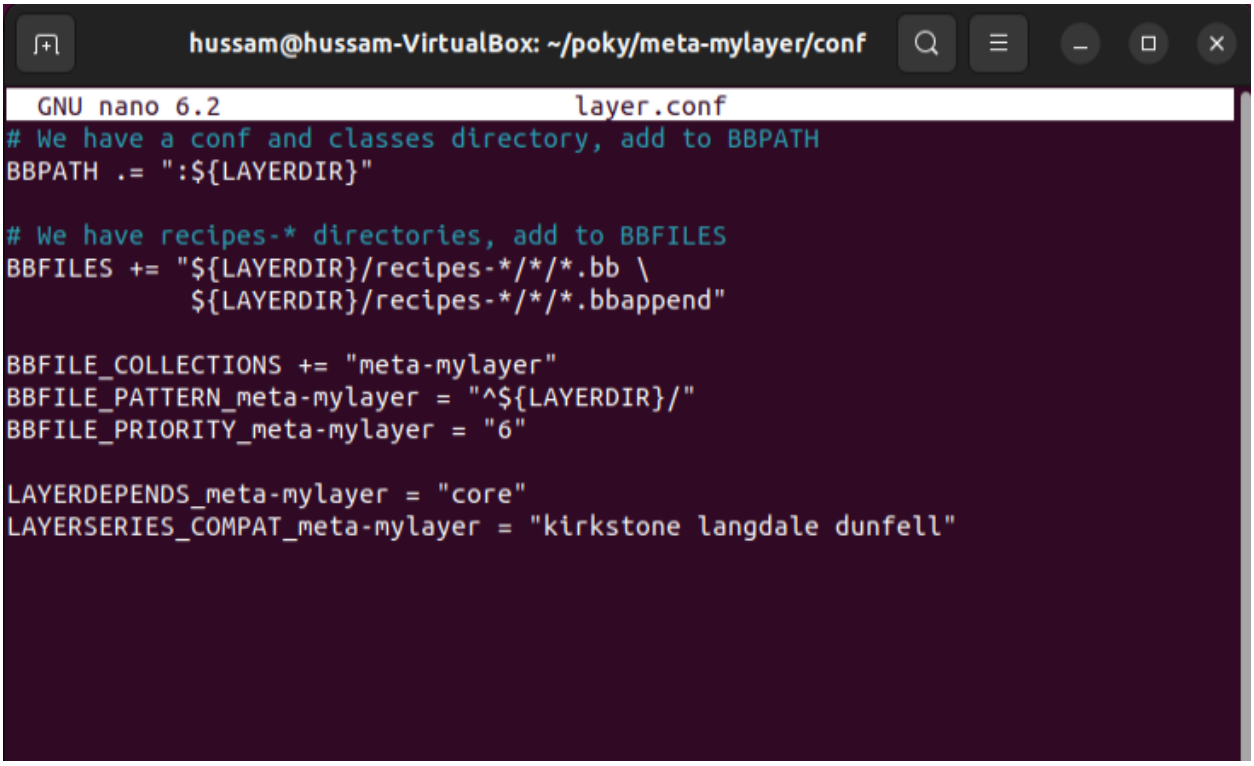

Figure 20 Config file of our custom layer

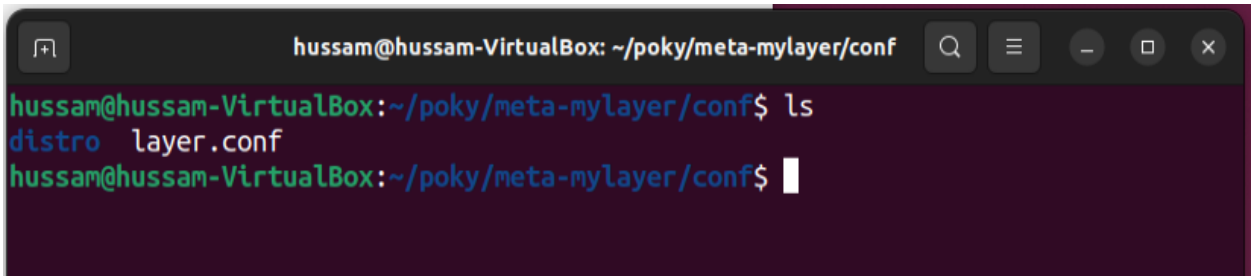

Figure 19 Contents of config folder

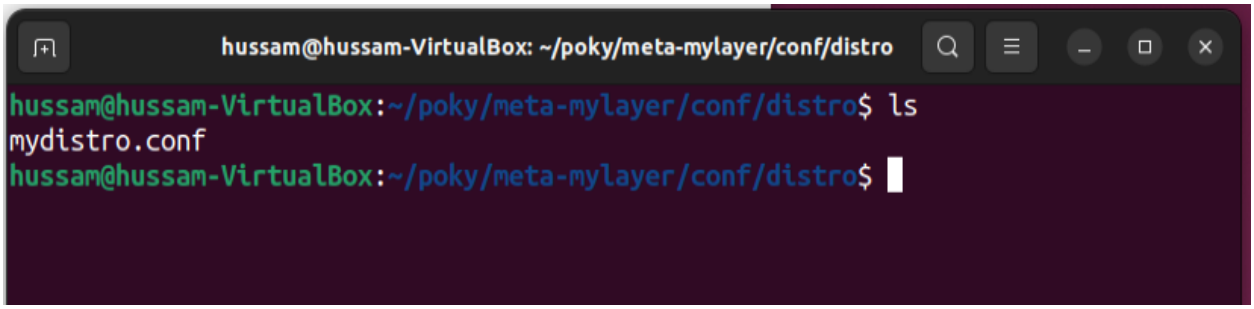

Figure 21 Contents of the distro folder inside the custom layer

4.5 Creating our own Linux Distribution

Now, we will create our own Linux Distribution (often shortened to 'Linux Distro') which is basically a version of the open-source Linux operating system that is packed with other components, such as installation programs, management tools, and additional software. Our distribution will be based on Yocto's Poky distribution.

The custom distro will contain some extra information about the distro itself as (Name, Version, Code Name, SDK Vendor, SDK Version, Maintainer and Target Vendor) each of these information are already present in the Yocto's Poky Distro so by adding these info in our distro, it overwrites the data already present inside Poky distribution.

In addition to the basic distro information, we will be adding some extra configurations to be included in the distro, like systemd, wifi, opengl, etc.

The distribution configuration file needs to be created in the conf/distro directory of our layer. We will be naming it using our distribution name (e.g. mydistro.conf).

The DISTRO variable in our local.conf, present inside the build directory, file determines the name of our distribution.

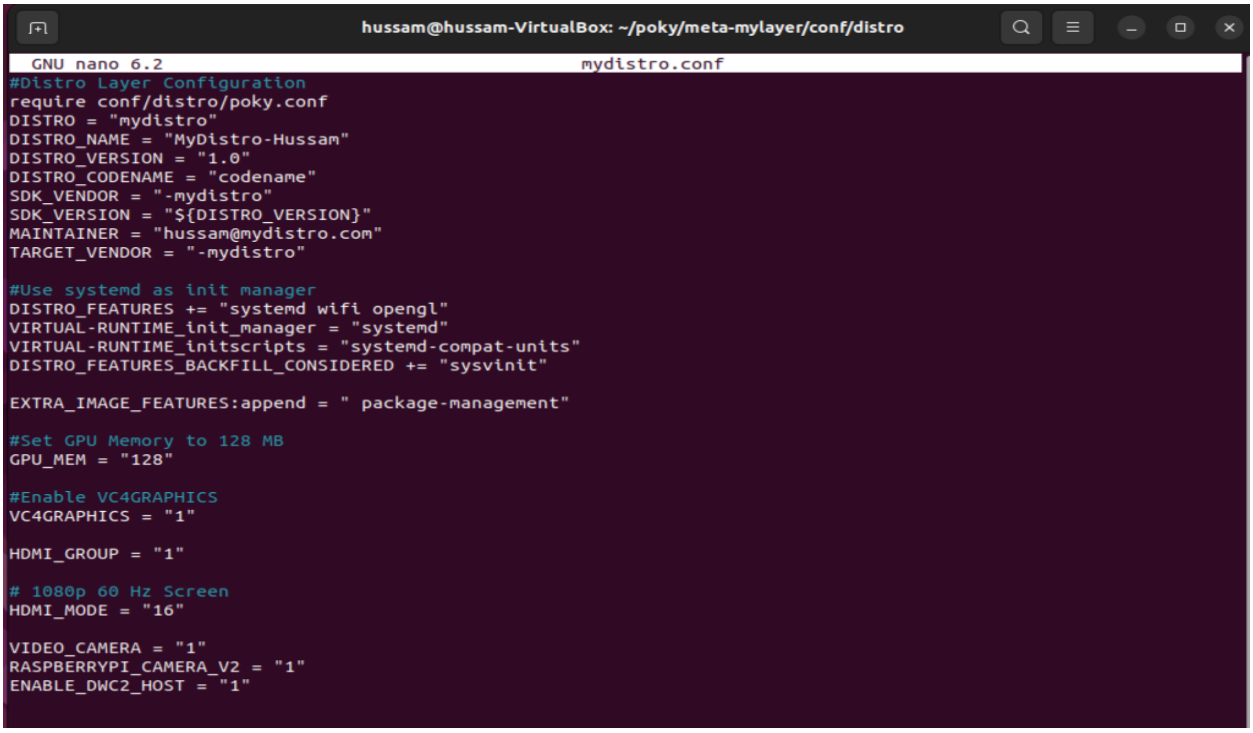

Figure 22 Contents of distro.conf file

4.6 Configuring our Layer

We will configure the files created in section 5.4 inside the layer that we created.

- **recipes-core:** It contains the recipe which is responsible for building our own image inside images directory and it includes the packages to be included in our image. The image recipe is called “myimage-gp.bb”.

The following tree shows the files inside:

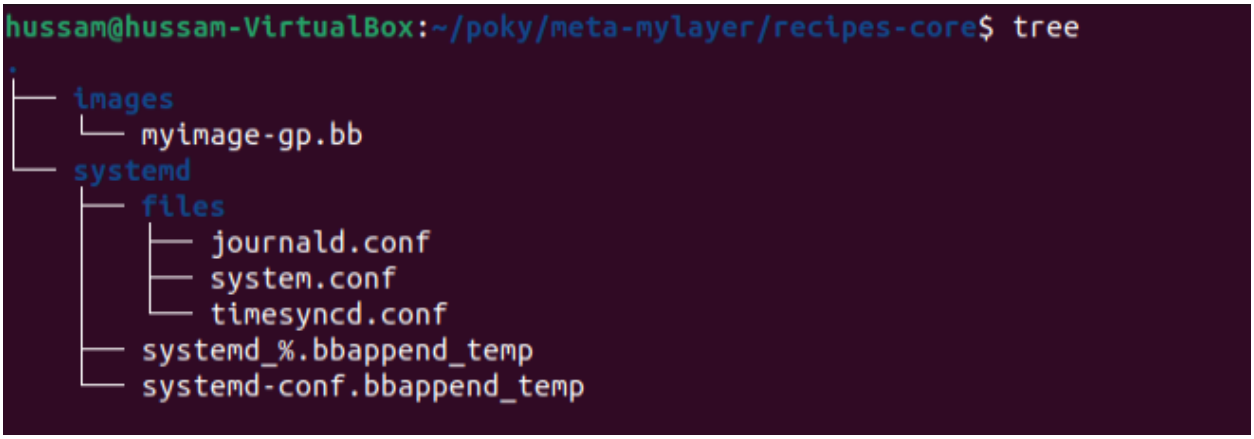

Figure 23 Files inside recipes-core

- **recipes-kernel:** It contains recipes related to the kernel like device drivers. In our case, we added a kernel module to display a simple message to the kernel when the module runs.

The following tree shows the files inside:

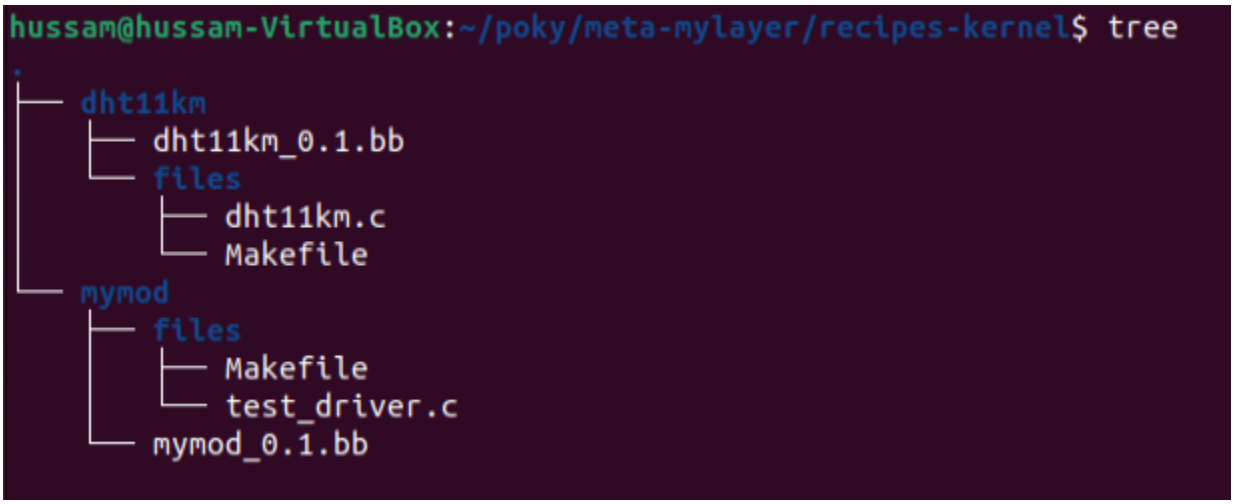

Figure 24 Files inside recipes-kernel

- **recipes-apps**: It contains recipes that are used to build our custom-made applications.

The following tree shows files inside the recipes-apps:

Figure 25 Files inside recipes-apps

Here we will have our applications, some of them are written in C++ like LaneDetection, some written in Python like VehiclesDetection, TrafficSignsDetection, TrafficLightsDetection and Pedestrian Detections, and others written in Qt C++ like CarDashboard.

- **recipes-support:** It contains some recipes which add some configuration to an already made recipe inside another layer. In our case, we wanted to enable a certain package configuration which

is disabled by default. However, we can not go into the original recipe and edit it. Instead we create an extension (.bbappend) which includes only the extra configuration and during building the image. The original and the .bbappend recipes are considered as one. It is simple to do that, we look for the original recipe's location so that we can mimic the same location in our layer. For example, in our case we want to modify the OpenCV recipe to enable that package configuration. OpenCV recipe is located in meta/recipe-support/opencv and according to that we added the same location in our meta layer and the .bbappend contains the package configuration to be enabled which is "dnn" as shown. These details will be discussed later as the reason behind the "dnn" package configuration being used.

Figure 26 Files inside recipes-support

4.7 Bitbake

4.7.1 Introduction

Fundamentally, BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints. One of BitBake's main users, OpenEmbedded, takes this core and builds embedded Linux software stacks using a task-oriented approach.

Conceptually, BitBake is like GNU Make in some regards but has significant differences:

- BitBake executes tasks according to the provided metadata that builds up the tasks. Metadata is stored in recipe (.bb) and related recipe "append" (.bbappend) files, configuration (.conf) and underlying include (.inc) files, and in class (.bbclass) files. The metadata provides BitBake with instructions on what tasks to run and the dependencies between those tasks.

- BitBake includes a fetcher library for obtaining source code from various places such as local files, source control systems, or websites.

- The instructions for each unit to be built (e.g. a piece of software) are known as "recipe" files and contain all the information about the unit (dependencies, source file locations, checksums, description and so on).

- BitBake includes a client/server abstraction and can be used from a command line or used as a service over XML-RPC and has several different user interfaces.

4.7.2 Concepts

4.7.2.1 Recipes

BitBake Recipes, which are denoted by the file extension .bb, are the most basic metadata files. These recipe files provide BitBake with the following:

- Descriptive information about the package (author, homepage, license, and so on).
- The version of the recipe.
- Existing dependencies (both build and runtime dependencies).
- Where the source code resides and how to fetch it.
- Whether the source code requires any patches, where to find them, and how to apply them.
- How to configure and compile the source code.
- How to assemble the generated artifacts into one or more installable packages.
- Where on the target machine to install the package or packages created.

Within the context of BitBake, or any project utilizing BitBake as its build system, files with the .bb extension are referred to as recipes.

4.7.2.2 Configuration File

Configuration files, which are denoted by the .conf extension, define various configuration variables that govern the project's build process. These files fall into several areas that define machine configuration, distribution configuration, possible compiler tuning, general common configuration, and user configuration. The main configuration file is the sample bitbake.conf file, which is located within the BitBake source tree conf directory.

4.7.2.3 Classes

Class files, which are denoted by the .bbclass extension, contain information that is useful to share between metadata files. The BitBake source tree currently comes with one class metadata file called base.bbclass. You can find this file in the classes directory. The base.bbclass class files is special since it is always included automatically for all recipes and classes. This class contains definitions for standard basic tasks such as fetching, unpacking, configuring (empty by default), compiling (runs any Makefile present), installing (empty by default) and packaging (empty by default). These tasks are often overridden or extended by other classes added during the project development process.

4.7.2.4 Layers

Layers allow you to isolate different types of customizations from each other. While you might find it tempting to keep everything in one layer when working on a single project, the more modular your metadata, the easier it is to cope with future changes.

To illustrate how you can use layers to keep things modular, consider customizations you might make to support a specific target machine. These types of customizations typically reside in a special layer, rather than a general layer, called a Board Support Package (BSP) layer. Furthermore, the machine customizations should be isolated from recipes and metadata that

support a new GUI environment, for example. This situation gives you a couple of layers: one for the machine configurations and one for the GUI environment. It is important to understand, however, that the BSP layer can still make machine-specific additions to recipes within the GUI environment layer without polluting the GUI layer itself with those machine-specific changes. You can accomplish this through a recipe that is a BitBake append (.bbappend) file.

4.7.2.5 Append Files

Append files, which are files that have the .bbappend file extension, extend or override information in an existing recipe file.

BitBake expects every append file to have a corresponding recipe file. Furthermore, the append file and corresponding recipe file must use the same root filename. The filenames can differ only in the file type suffix used (e.g. formfactor_0.0.bb and formfactor_0.0.bbappend).

Information in append files extends or overrides the information in the underlying, similarly named recipe files.

When you name an append file, you can use the "%" wildcard character to allow for matching recipe names. For example, suppose you have an append file named as follows:

- busybox_1.21.%.bbappend

That append file would match any busybox_1.21.x.bb version of the recipe. So, the append file would match the following recipe names:

- busybox_1.21.1.bb
- busybox_1.21.2.bb
- busybox_1.21.3.bb

If the busybox recipe was updated to busybox_1.3.0.bb, the append name would not match. However, if you named the append file busybox_1.%.bbappend, then you would have a match.

In the most general case, you could name the append file something as simple as busybox_%.bbappend to be entirely version independent.

4.7.3 The BitBake Command

The bitbake command is the primary interface to the BitBake tool.

4.7.4 Execution

The primary purpose for running BitBake is to produce some kind of output such as a single installable package, a kernel, a software development kit, or even a full, board-specific bootable Linux image, complete with bootloader, kernel, and root filesystem. Of course, you can execute the bitbake command with options that cause it to execute single tasks, compile single recipe files, capture or clear data, or simply return information about the execution environment.

4.7.5 Parsing the Base Configuration Metadata

The first thing BitBake does is parse base configuration metadata. Base configuration metadata consists of your project's bblayers.conf file to determine what layers BitBake needs to recognize, all necessary layer.conf files (one from each layer), and bitbake.conf. The data itself is of various types:

- Recipes: Details about particular pieces of software.
- Class Data: An abstraction of common build information (e.g. how to build a Linux kernel).
- Configuration Data: Machine-specific settings, policy decisions, and so forth. Configuration data acts as the glue to bind everything together.

The layer.conf files are used to construct key variables such as BBPATH and BBFILES. BBPATH is used to search for configuration and class files under the conf and classes directories, respectively. BBFILES is used to locate both recipe and recipe append files (.bb and .bbappend). If there is no bblayers.conf file, it is assumed the user has set the BBPATH and BBFILES directly in the environment.

Next, the bitbake.conf file is located using the BBPATH variable that was just constructed. The bitbake.conf file may also include other configuration files using the include or require directives.

Prior to parsing configuration files, BitBake looks at certain variables, including:

- BB_ENV_PASSTHROUGH
- BB_ENV_PASSTHROUGH_ADDITIONS
- BB_PRESERVE_ENV
- BB_ORIGENV
- BITBAKE_UI

The first four variables in this list relate to how BitBake treats shell environment variables during task execution. By default, BitBake cleans the environment variables and provides tight control over the shell execution environment. However, through the use of these first four variables, you can apply your control regarding the environment variables allowed to be used by BitBake in the shell during execution of tasks. See the "Passing Information Into the Build Task Environment" section and the information about these variables in the variable glossary for more information on how they work and on how to use them.

The base configuration metadata is global and therefore affects all recipes and tasks that are executed.

BitBake first searches the current working directory for an optional conf/bblayers.conf configuration file. This file is expected to contain a BBLAYERS variable that is a space-delimited list of 'layer' directories. Recall that if BitBake cannot find a bblayers.conf file, then it is assumed the user has set the BBPATH and BBFILES variables directly in the environment.

For each directory (layer) in this list, a conf/layer.conf file is located and parsed with the LAYERDIR variable being set to the directory where the layer was found. The idea is these files automatically set up BBPATH and other variables correctly for a given build directory.

BitBake then expects to find the conf/bitbake.conf file somewhere in the user-specified BBPATH. That configuration file generally has include directives to pull in any other metadata such as files specific to the architecture, the machine, the local environment, and so forth.

Only variable definitions and include directives are allowed in BitBake .conf files. Some variables directly influence BitBake's behavior. These variables might have been set from the environment depending on the environment variables previously mentioned or set in the configuration files. The "Variables Glossary" chapter presents a full list of variables.

After parsing configuration files, BitBake uses its rudimentary inheritance mechanism, which is through class files, to inherit some standard classes. BitBake parses a class when the inherit directive responsible for getting that class is encountered.

The base.bbclass file is always included. Other classes that are specified in the configuration using the INHERIT variable are also included. BitBake searches for class files in a classes subdirectory under the paths in BBPATH in the same way as configuration files.

4.7.6 Locating and Parsing Recipes

During the configuration phase, BitBake will have set BBFILES. BitBake now uses it to construct a list of recipes to parse, along with any append files (.bbappend) to apply. BBFILES is a space-separated list of available files and supports wildcards. An example would be:

```
BBFILES = "/path/to/bbfiles/*.bb /path/to/appends/*.bbappend"
```

BitBake parses each recipe and append file located with BBFILES and stores the values of various variables into the datastore.

For each file, a fresh copy of the base configuration is made, then the recipe is parsed line by line. Any inherit statements cause BitBake to find and then parse class files (.bbclass) using BBPATH as the search path. Finally, BitBake parses in order any append files found in BBFILES.

4.7.7 Dependencies

Each target BitBake builds consists of multiple tasks such as fetch, unpack, patch, configure, and compile. For best performance on multi-core systems, BitBake considers each task as an independent entity with its own set of dependencies.

At a basic level, it is sufficient to know that BitBake uses the DEPENDS and RDEPENDS variables when calculating dependencies.

4.8 SDK

4.8.1 Introduction

SDK stands for Software Development Kit. It is a collection of software development tools that enable developers to create software applications for a specific platform or operating system.

An SDK typically includes tools like libraries, documentation, sample code, debugging tools, and a software emulator or simulator. These tools help developers to create software applications that can interact with the platform or operating system on which they are intended to run.

For example, if a developer wants to create a mobile application for iOS, they would use the iOS SDK provided by Apple. The SDK would provide the necessary tools and resources to create an application that can run on an iPhone or iPad. Similarly, if a developer wants to create an application for Windows, they would use the Windows SDK provided by Microsoft.

SDKs are used by developers to streamline the software development process and make it easier to create applications that are optimized for a specific platform or operating system.

All SDKs consist of the following:

- Cross-Development Toolchain: This toolchain contains a compiler, debugger, and various associated tools.
- Libraries, Headers, and Symbols: The libraries, headers, and symbols are specific to the image (i.e. they match the image against which the SDK was built).
- Environment Setup Script: This *.sh file, once sourced, sets up the cross-development environment by defining variables and preparing for SDK use.

You can use an SDK to independently develop and test code that is destined to run on some target machine. SDKs are completely self-contained. The binaries are linked against their own copy of libc, which results in no dependencies on the target system. To achieve this, the pointer to the dynamic loader is configured at install time since that path cannot be dynamically altered. This is the reason for a wrapper around the populate_sdk and populate_sdk_ext archives.

Another feature of the SDKs is that only one set of cross-compiler toolchain binaries are produced for any given architecture. This feature takes advantage of the fact that the target hardware can be passed to gcc as a set of compiler options. Those options are set up by the environment script and contained in variables such as CC and LD. This reduces the space needed for the tools. Understand, however, that every target still needs its own sysroot because those binaries are target-specific.

The SDK development environment consists of the following:

The self-contained SDK, which is an architecture-specific cross-toolchain and matching sysroots (target and native) all built by the OpenEmbedded build system (e.g. the SDK). The toolchain and sysroots are based on a Metadata configuration and extensions, which allows you to cross-develop on the host machine for the target hardware. Additionally, the extensible SDK contains the devtool functionality.

The Quick EMUlator (QEMU), which lets you simulate target hardware. QEMU is not literally part of the SDK. You must build and include this emulator separately. However, QEMU plays an important role in the development process that revolves around use of the SDK.

4.8.2 The Cross-Development Toolchain

The Cross-Development Toolchain consists of a cross-compiler, cross-linker, and cross-debugger that are used to develop user-space applications for targeted hardware; in addition, the extensible SDK comes with built-in devtool functionality. This toolchain is created by running a SDK installer script or through a Build Directory that is based on your metadata configuration or extension for your targeted device. The cross-toolchain works with a matching target sysroot.

4.8.3 Sysroots

The native and target sysroots contain needed headers and libraries for generating binaries that run on the target architecture. The target sysroot is based on the target root filesystem image that is built by the OpenEmbedded build system and uses the same metadata configuration used to build the cross-toolchain.

4.8.4 SDK Development Model

Fundamentally, the SDK fits into the development process as follows:

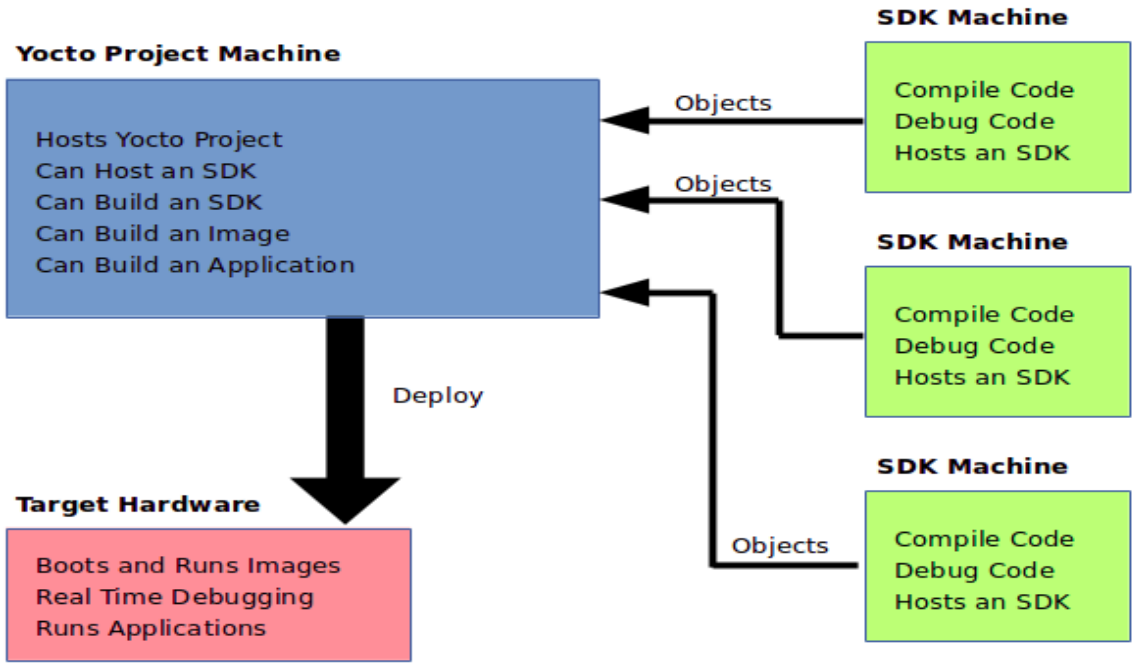

Figure 27 SDK Development Model

The SDK is installed on any machine and can be used to develop applications, images, and kernels. The fundamental concept is that the machine that has the SDK installed does not have to be associated with the machine that has the Yocto Project installed. A developer can independently compile and test an object on their machine and then, when the object is ready for integration into an image, they can simply make it available to the machine that has the Yocto Project. Once the object is available, the image can be rebuilt using the Yocto Project to produce the modified image.

You just need to follow these general steps:

- Install the SDK for your target hardware.
- Download or Build the Target Image: The Yocto Project supports several target architectures and has many pre-built kernel images and root filesystem images.
- Develop and Test your Application

4.8.5 Building an SDK Installer

As an alternative to locating and downloading an SDK installer, you can build the SDK installer. Follow these steps:

1. Set Up the Build Environment: Be sure you are set up to use BitBake in a shell.

2. Clone the ``poky`` Repository: You need to have a local copy of the Yocto Project Source Directory (i.e. a local poky repository).

3. Initialize the Build Environment: While in the root directory of the Source Directory (i.e. poky), run the oe-init-build-env environment setup script to define the OpenEmbedded build environment on your build host.

   ```
   $ source oe-init-build-env
   ```

   Among other things, the script creates the Build Directory, which is build in this case and is located in the Source Directory. After the script runs, your current working directory is set to the build directory.

4. Make Sure You Are Building an Installer for the Correct Machine: Check to be sure that your MACHINE variable in the local.conf file in your Build Directory matches the architecture for which you are building.

5. Make Sure Your SDK Machine is Correctly Set: If you are building a toolchain designed to run on an architecture that differs from your current development host machine (i.e. the build host), be sure that the SDKMACHINE variable in the local.conf file in your Build Directory is correctly set.

6. Build the SDK Installer: To build the SDK installer for a standard SDK and populate the SDK image, use the following command form. Be sure to replace image with an image (e.g. “core-image-minimal”):

   ```
   $ bitbake image -c populate_sdk
   ```

These commands produce an SDK installer that contains the sysroot that matches your target root filesystem.

When the bitbake command completes, the SDK installer will be in tmp/deploy/sdk in the Build Directory.

7. Run the Installer: You can now run the SDK installer from tmp/deploy/sdk in the Build Directory. Following is an example:

```
$ cd poky/build/tmp/deploy/sdk
$ ./poky-glibc-x86_64-core-image-minimal-core2-64-toolchain-ext-4.1.999.sh
```

During execution of the script, you choose the root location for the toolchain.

The following shows the directory where the SDK is install files are installed:

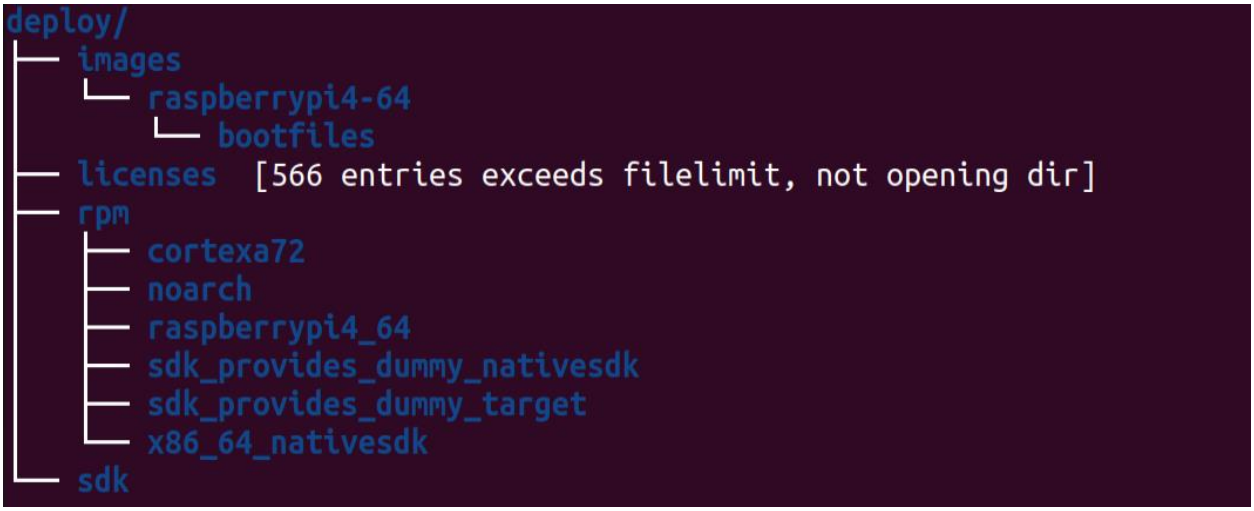

Figure 28 Contents of deploy folder

The following shows the contents of the SDK installation folder:

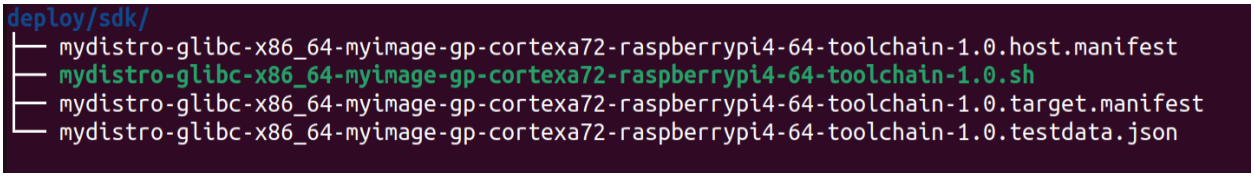

Figure 29 SDK Installation Folder

The script called "mydistro-glibc-x86_64-myimage-gp-cortexa72-raspberrypi4-64-toolchain-1.0.sh" is the SDK installation script.

The following shows the directory where the SDK is installed and the source file is:

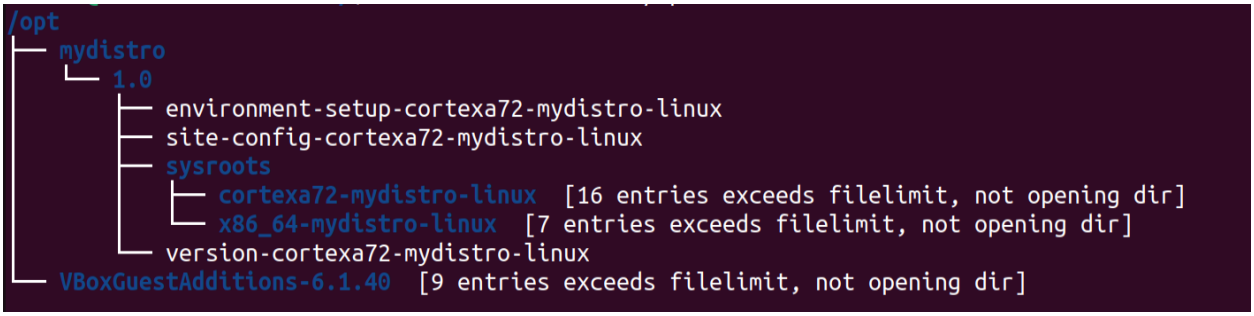

Figure 30 SDK Installation Directory

The script called "environment-setup-cortexa72-mydistro-linux" is the source file which adds the installed SDK to the environment variable of the currently working terminal. This can be done by:

```
$ ./environment-setup-cortexa72-mydistro-linux
```

4.9 Summary

In this section, we will go through the steps that we did in the previous sections in a brief manner.

First, in **4.1**, we checked that our machine can support the Yocto Project, then we updated and upgraded the specified application, and we installed the Yocto project.

Second, in **4.2**, we navigated to the Yocto Project directory that we installed in the previous step, then we initialized the environment of the terminal we're using by sourcing a specific script (not that this step will be used each time we want to access the Poky environment or issue any BitBake command). This script also created our first build directory which we explored in the same section.

In **4.3**, we used QEMU, which is an emulating tool provided by Yocto Project, to a small image called minimal-core-image as an exercise to get familiar with Yocto.

In **4.4**, we explored the idea of layers in Yocto, then we created our own layer and created some additional files inside that we will populate in later steps. These files consisted of:

- Recipes-core
- Recipes-kernel
- Distro (under the conf directory)
- Recipes-support
- Recipes-apps

In **4.5**, we created our Linux Distro which was based on Yocto's Poky.

In **4.6**, we went through the purpose behind each of the files we created in **4.4**, populated them with new files, and explored them.

In **4.7**, we explained what BitBake is, what it consists of, and how it performs.

In we explored what an SDK is, and how we are going to use it.

Chapter 5: Running an image on Raspberry Pi

Now that we have generated the image, it is time to flash it on the SD Card and run it on Raspberry Pi. To do that, follow these steps:

1. Insert the SD Card into the host machine, unmount and format it.

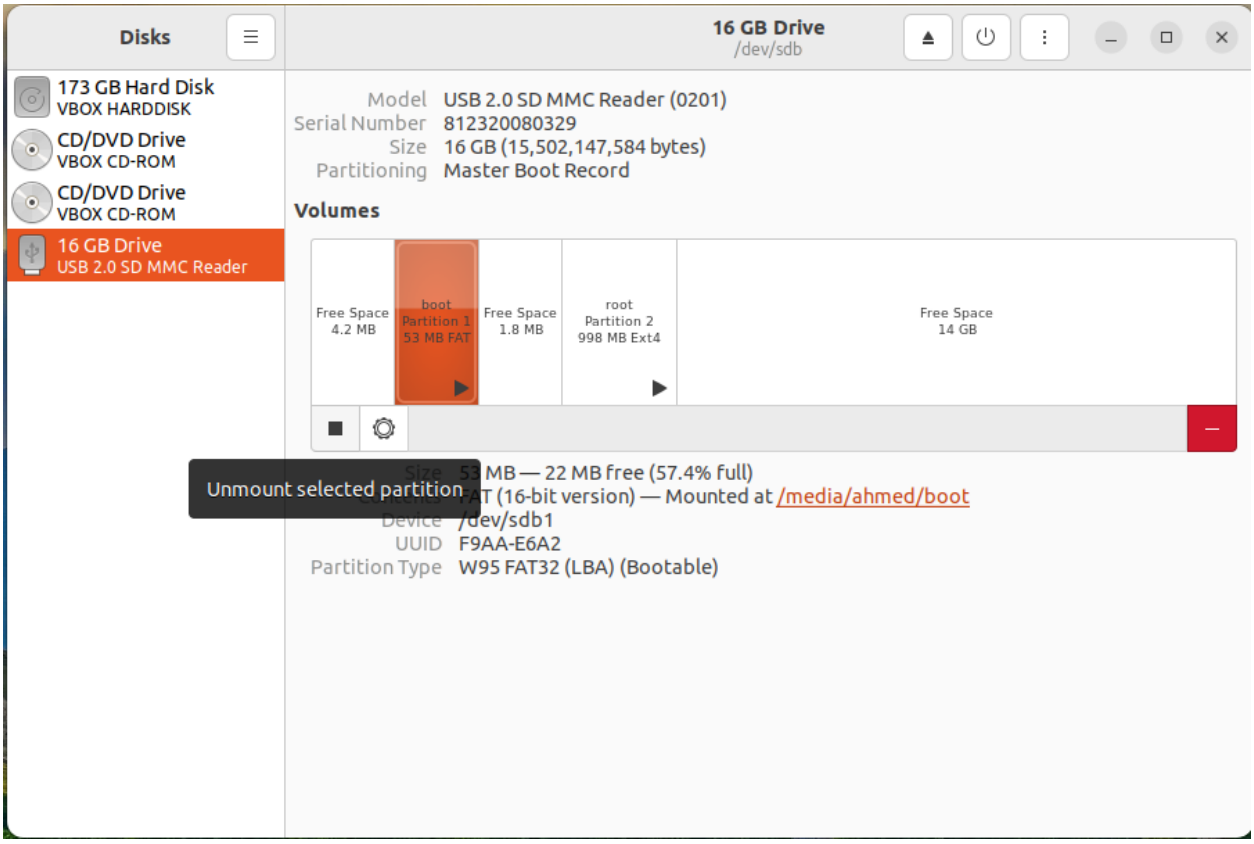

Figure 31 Unmounting the SD Card

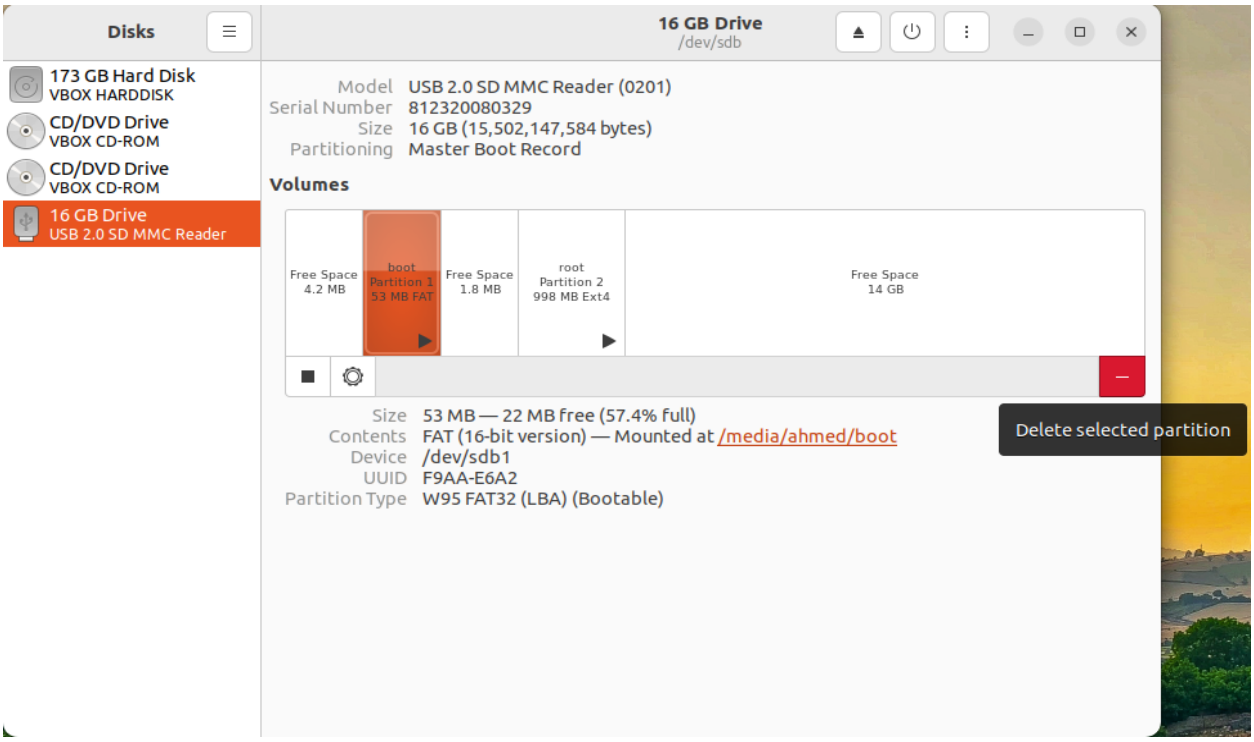

Figure 32 Removing the partitions

2. Go to the directory where the image is located which is in: myBuild/tmp/deploy/images/raspberrypi4-64. “raspberrypi4-64” is the name of the target which is specified in the local.conf in the MACHINE variable.

3. Search for the “image_name_rootfs.wic.tar” file and extract it.

4. This extracted .wic file is the image itself which will be flashed on the SD Card.

5. An application called “dd” in Linux will be used to flash the image on the SD Card through this command sudo dd if=./core-image-minimal-raspberrypi3-20221017134421.rootfs.wic of=/dev/sdb bs=1M iflag=fullblock status=progress

   “core-image-minimal-raspberrypi3-20221017134421.rootfs.wic” is an example for the .wic file which will be replaced with the current image.

   This process takes a while as it writes the image files onto the SD Card and creates two partitions which are boot and rootfs.

6. Once the flashing has ended, the SD Card is now ready to be inserted into the Raspberry Pi.

7. Insert the power cable into the Raspberry Pi and let it boot up.

8. There are two ways to display the image after running on Raspberry Pi:
   a. By connecting an external monitor through the HDMI cable in RPi.

b. Headless: Without using an external monitor. By using an application called “ssh” which can be used on the host machine to connect to the terminal of the created image on RPi.

Note that every time the image is edited by adding more applications or features or packages or configurations, the previous process has to be repeated exactly as it is. Obviously, this is a time consuming and exhausting process to do especially when there are some tests to be run on the image to check its functionality.

Therefore, the process of flashing the image on SD Card will be automated by using a simple bash script that will automatically do the whole process. This bash script will be discussed later in more detail.

Once we have successfully flashed the image on Raspberry Pi, we have several approaches on how to boot the image, or two to be frank. The first is to connect the necessary hardware from monitors to mouses and keyboards. The second one is to connect remotely through a secure connection using what is known as a secure shell or more commonly SSH.

As the first method sounds, it’s just straight forward, we connect the screen and the keyboard and hook up the Raspberry Pi with a strong enough power supply and then we are done. By doing so we have booted Raspberry Pi and have access to a terminal which we can navigate using the connected hardware.

The second method is not as simple, but it does provide us with remote access that can come in handy, especially for testing. The connection is made by using SSH or secure shell which is basically a network communication protocol that enables two computers or two devices to communicate and share data. To establish an SSH connection, we can use an SSH client app called PuTTY on windows or use the command ssh on Linux. Either way, we will need to provide a bit of information before we can begin using it fully. We will need to provide:

- Hostname or IP address.
- Port number.

Raspberry Pi doesn’t connect to WiFi by default, so we’ll need to do some adjustments to bypass this issue. For that, we will use a tool called nmtui or Network Manager Text User Interface.

Nmtui is a command-line tool used in Linux operating systems for configuring network connections. We will be using its simple and easy-to-use text-based interface to apply all the necessary configurations. To access nmtui, we will simply type ‘nmtui’ in the terminal, and this will launch the text-based interface.

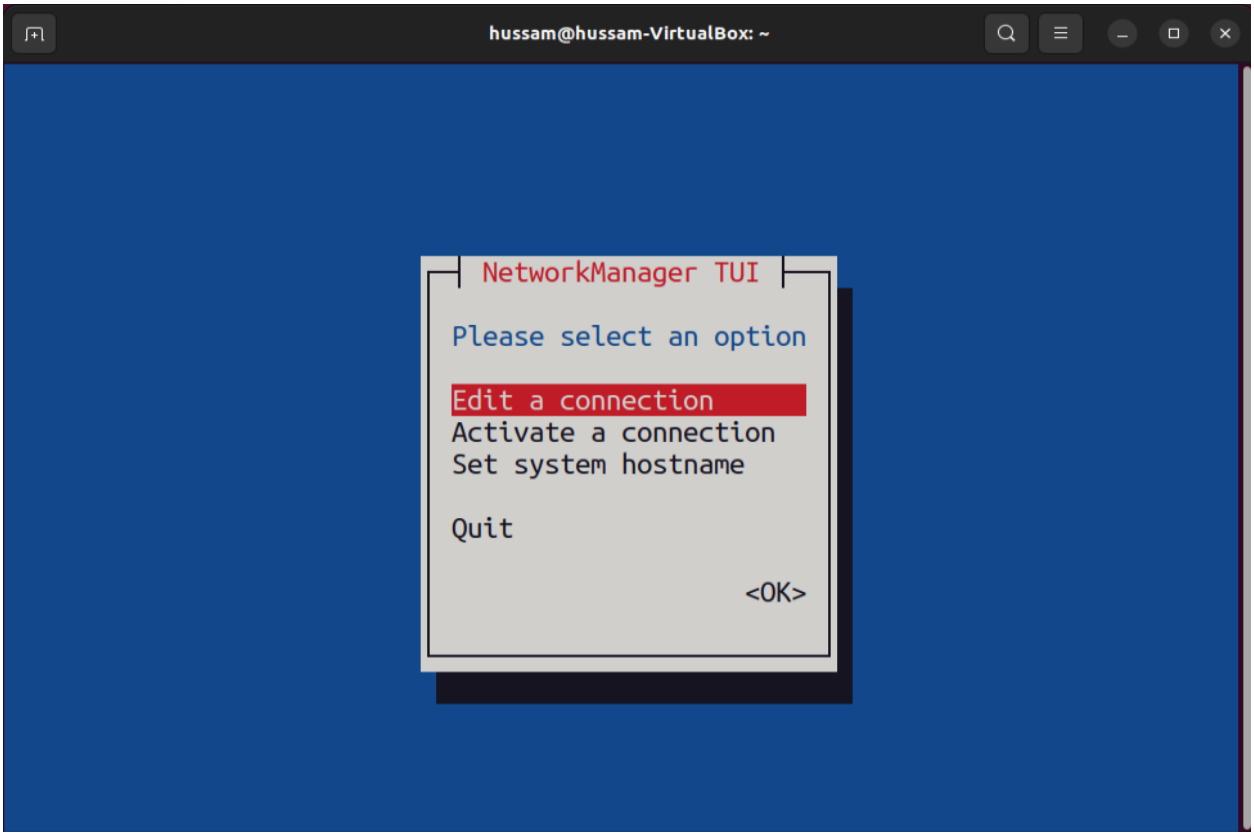

Figure 33 NmtUI Interface

Go to Edit a connection, choose the option to add a WiFi connection, add all the details, and check connect automatically.

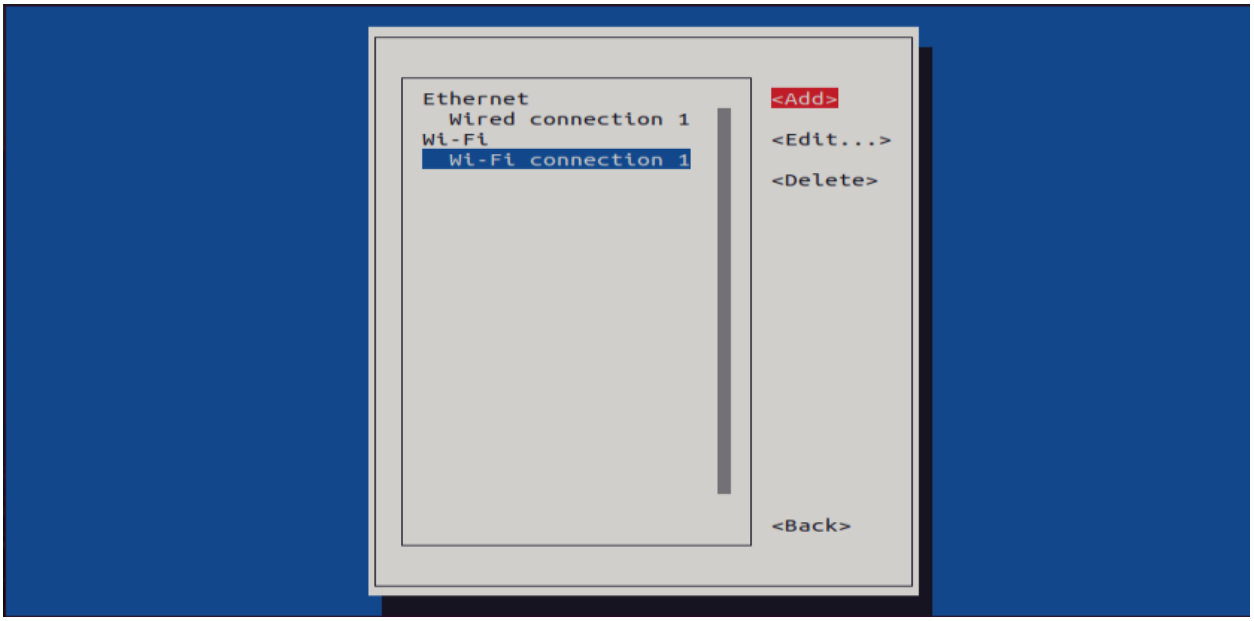

Figure 34 Choosing the WIFI Network

Now that Raspberry Pi connects automatically to our WiFi network, we need to obtain its IP address which can be done in many ways.

Status	Name	IP	Manufacturer	MAC address
	192.168.1.1	192.168.1.1	HUAWEI TECHNOLOGIES CO.,LTD	D8:29:18:BD:FA:52
	192.168.1.2	192.168.1.2	Cloud Network Technology (Samoa) Limited	28:3A:4D:3B:4E:21
	192.168.1.3	192.168.1.3		C2:6E:7D:3F:AD:A8
	192.168.1.5	192.168.1.5	Samsung Electronics Co.,Ltd	68:E7:C2:97:CD:B1
	raspberrypi3	192.168.1.6	Raspberry Pi Foundation	B8:27:EB:6A:D3:0E
	192.168.1.7	192.168.1.7	SAMSUNG ELECTRO-MECHANICS(THAILAND)	24:18:1D:76:BA:68
	192.168.1.9	192.168.1.9	GUANGDONG OPPO MOBILE TELECOMMUNICATIONS CORP.,LTD	D4:67:D3:EA:40:BB
	192.168.1.11	192.168.1.11	GUANGDONG OPPO MOBILE TELECOMMUNICATIONS CORP.,LTD	F4:D6:20:14:86:D3
	192.168.1.12	192.168.1.12	GUANGDONG OPPO MOBILE TELECOMMUNICATIONS CORP.,LTD	C4:FE:5B:9F:A8:1B
	LAPTOP-RMT97NPJ.home	192.168.1.13	LCFC(HeFei) Electronics Technology co., ltd	E8:6A:64:61:92:5A
	raspberrypi3	192.168.1.16	Raspberry Pi Foundation	B8:27:EB:3F:86:5B
	raspberrypi3	192.168.1.50	Raspberry Pi Foundation	B8:27:EB:6A:D3:0E
	LAPTOP-RMT97NPJ	192.168.56.1		0A:00:27:00:00:17
	LAPTOP-RMT97NPJ	192.168.131.1	VMware, Inc.	00:50:56:C0:00:01
	192.168.131.254	192.168.131.254	VMware, Inc.	00:50:56:F5:0E:6F
	LAPTOP-RMT97NPJ	192.168.230.1	VMware, Inc.	00:50:56:C0:00:08
	192.168.230.254	192.168.230.254	VMware, Inc.	00:50:56:F2:D1:59

Figure 35 Getting the IP of the Raspberry Pi

Now that we have the IP, we can issue the SSh command on Linux or use PuTTY

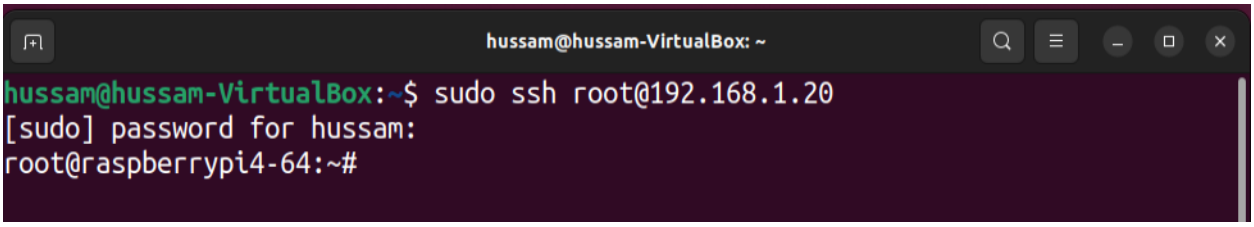

Figure 36 Getting into Raspberry Pi using SSH

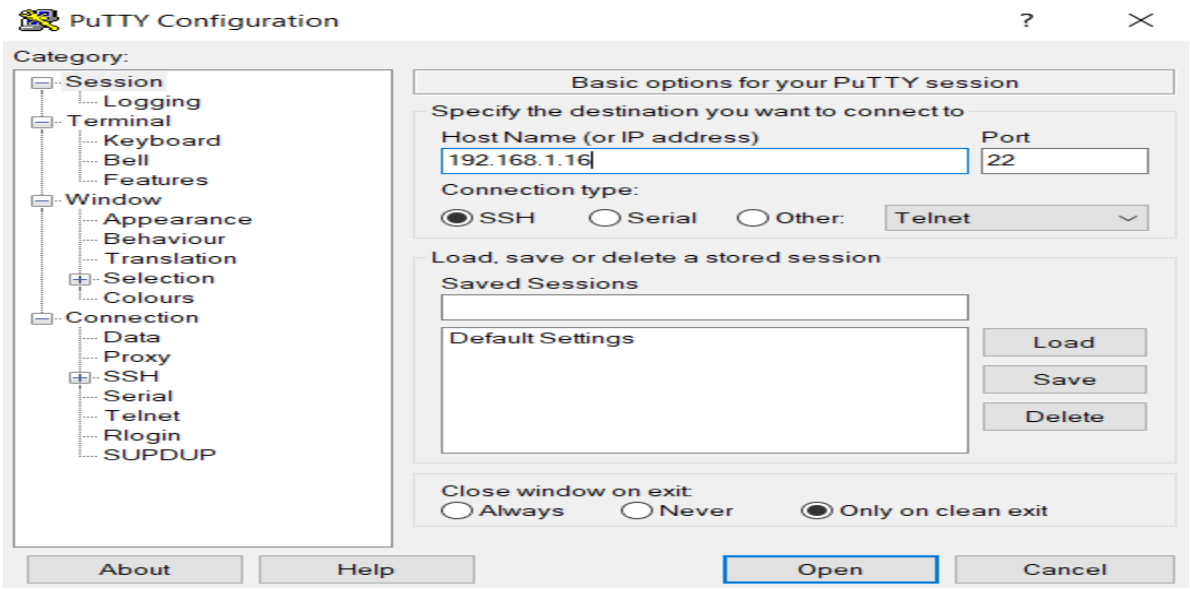

Figure 37 Getting into Raspberry Pi using PuTTy

Using PuTTy, We login as root

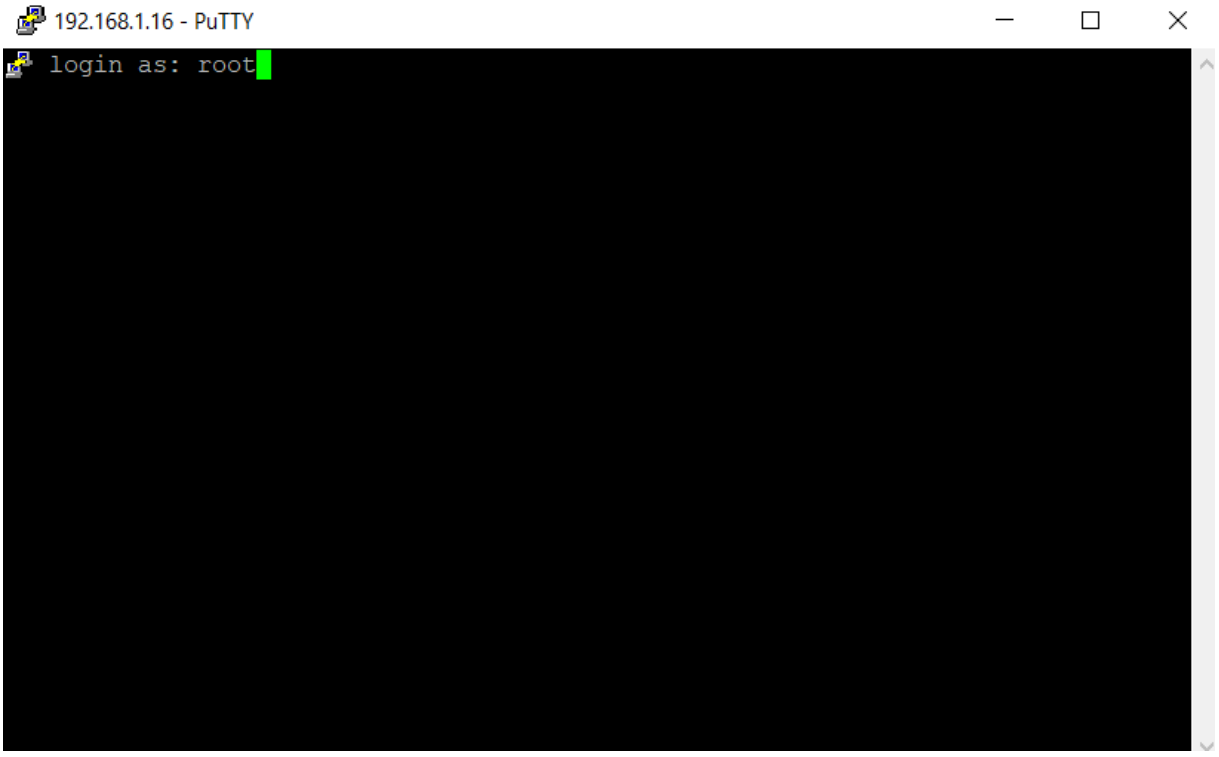

Figure 38 Logging in as root user

And finally, we now have access to our image and its files.

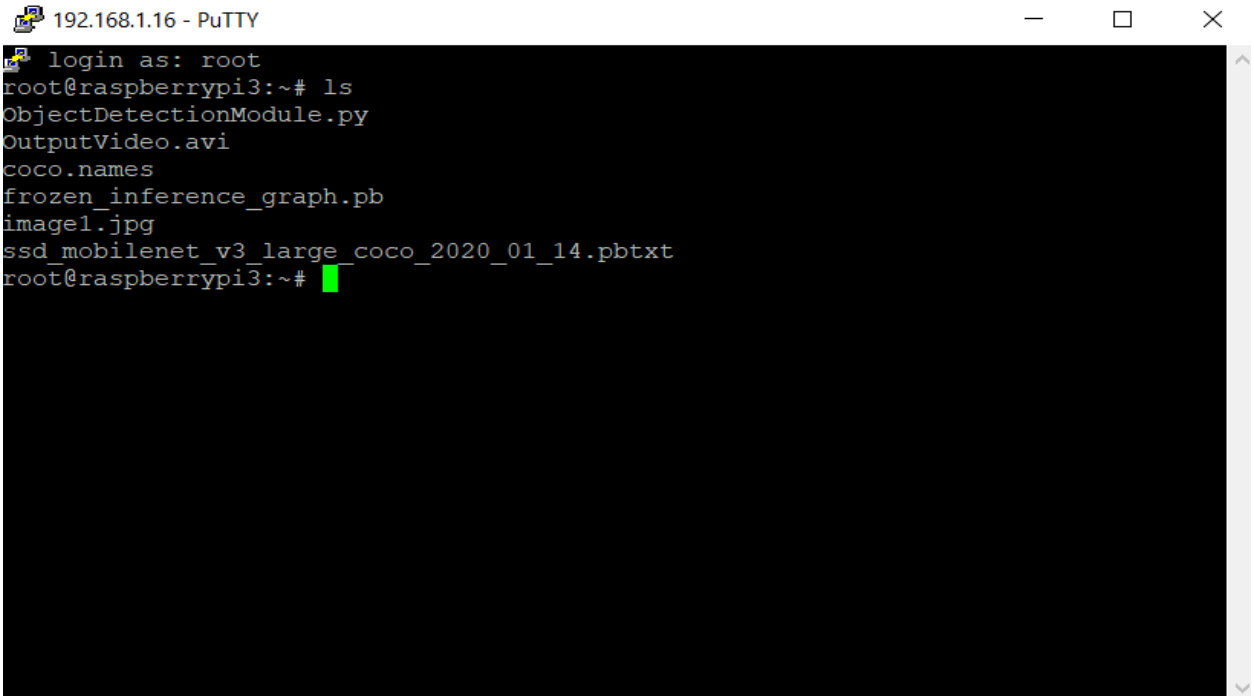

Figure 39 Accessing the image using PuTTy

Chapter 6: Applications

For the models in the project, 5 were trained to satisfy different requirements and different needs in order to simulate multiple features that could be installed on an ADAS. Those models being:

1. Vehicle detection.
2. Traffic signs detection and classification.
3. Traffic lights detection and classification.
4. Pedestrian detection.
5. Drowsiness detection.
6. Lane detection.

6.1 Drowsiness detection

If you have driven before, you've been drowsy at the wheel at some point. It's not something we like to admit but it's an important problem with serious consequences that needs to be addressed. 1 in 4 vehicle accidents are caused by drowsy driving and 1 in 25 adult drivers report that they have fallen asleep at the wheel in the past 30 days. The scariest part is that drowsy driving isn't just falling asleep while driving. Drowsy driving can be as small as a brief state of unconsciousness when the driver is not paying full attention to the road. Drowsy driving results in over 71,000 injuries, 1,500 deaths, and $12.5 billion in monetary losses per year. Due to the relevance of this problem, we believe it is important to develop a solution for drowsiness detection, especially in the early stages to prevent accidents**.**

Our solution to this problem is to build a detection system that identifies key attributes of drowsiness and triggers an alert when someone is drowsy before it is too late.

6.1.1 Data source and preprocessing

For our training and test data, we used the Real-Life Drowsiness Dataset created by a research team from the University of Texas at Arlington specifically for detecting multi-stage drowsiness. The end goal is to detect not only extreme and visible cases of drowsiness but allow our system to detect softer signals of drowsiness as well. The dataset consists of around 30 hours of videos of 60 unique participants. From the dataset, we were able to extract facial landmarks from 44 videos of 22 participants. This allowed us to obtain a sufficient amount of data for both the alert and drowsy state.

For each video, we used OpenCV to extract 1 frame per second starting at the 3-minute mark until the end of the video.

Each video was approximately 10 minutes long, so we extracted around 240 frames per video, resulting in 10560 frames for the entire dataset.

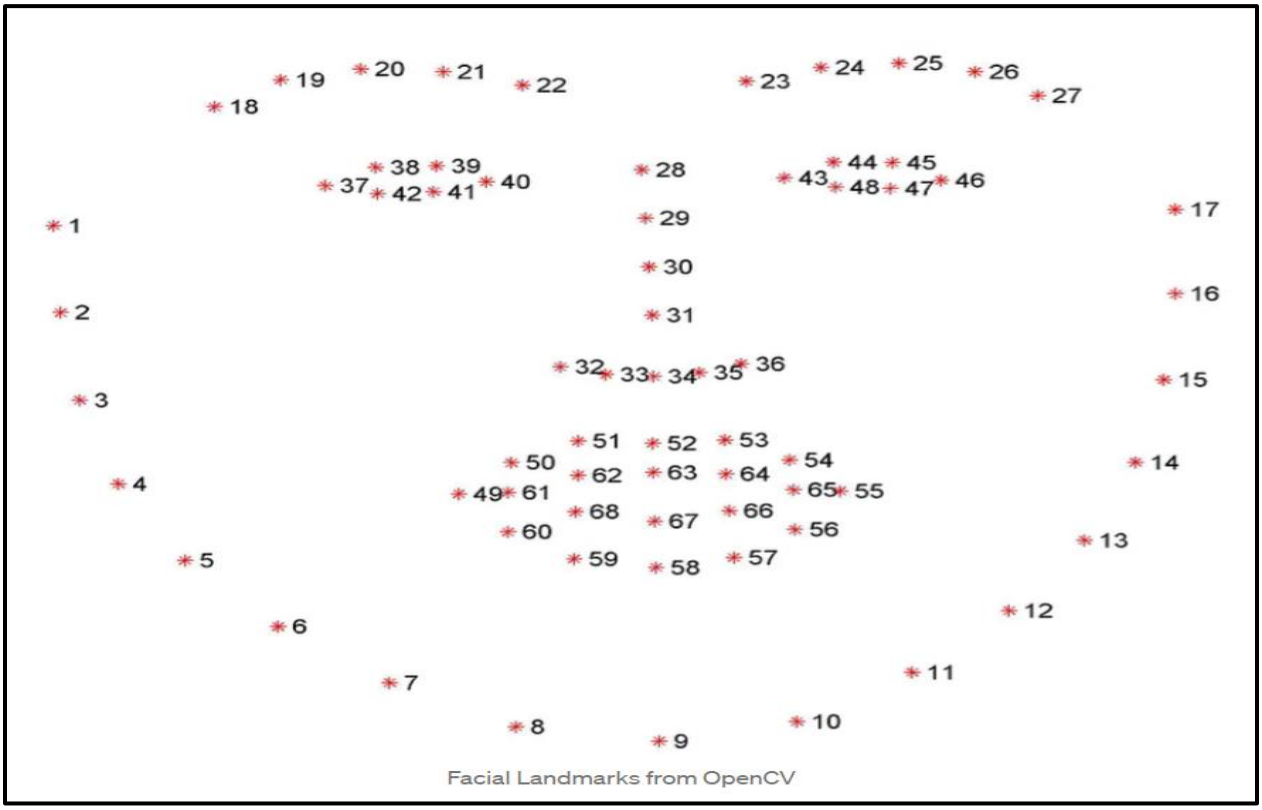

Figure 40 Facial Landmark from OpenCV

There were 68 total landmarks per frame but we decided to keep the landmarks for the eyes and mouth only (Points 37–68). These were the important data points we used to extract the features for our model.

6.1.2 Feature extraction

As briefly alluded to earlier, based on the facial landmarks that we extracted from the frames of the videos, we ventured into developing suitable features for our classification model. While we hypothesized and tested several features, the four core features that we concluded on for our final models were eye aspect ratio, mouth aspect ratio, pupil circularity, and finally, mouth aspect ratio over eye aspect ratio.

- **Eye Aspect Ratio (EAR)**

EAR, as the name suggests, is the ratio of the length of the eyes to the width of the eyes. The length of the eyes is calculated by averaging over two distinct vertical lines across the eyes as illustrated in the figure below.

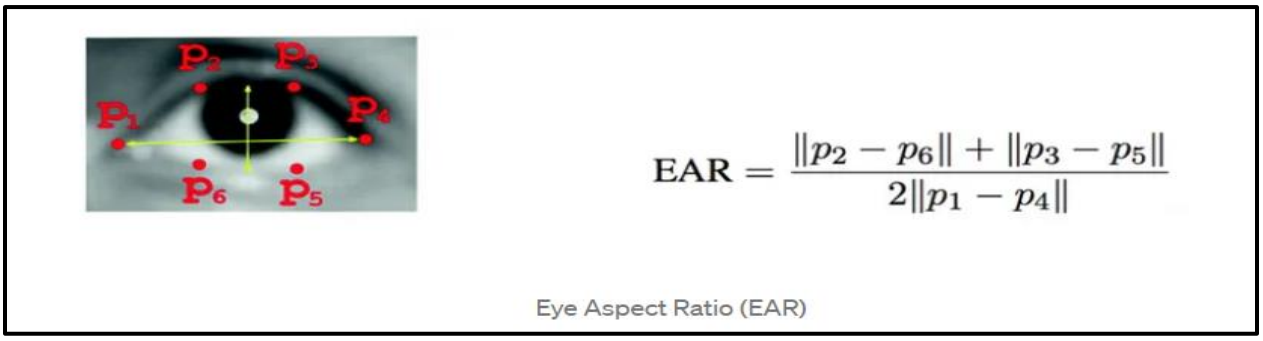

Figure 41 Eye Aspect Ratio (EAR)

Our hypothesis was that when an individual is drowsy, their eyes are likely to get smaller and they are likely to blink more.

- **Mouth Aspect Ratio (MAR)**

Computationally similar to the EAR, the MAR, as you would expect, measures the ratio of the length of the mouth to the width of the mouth. Our hypothesis was that as an individual becomes drowsy, they are likely to yawn and lose control over their mouth, making their MAR to be higher than usual in this state.

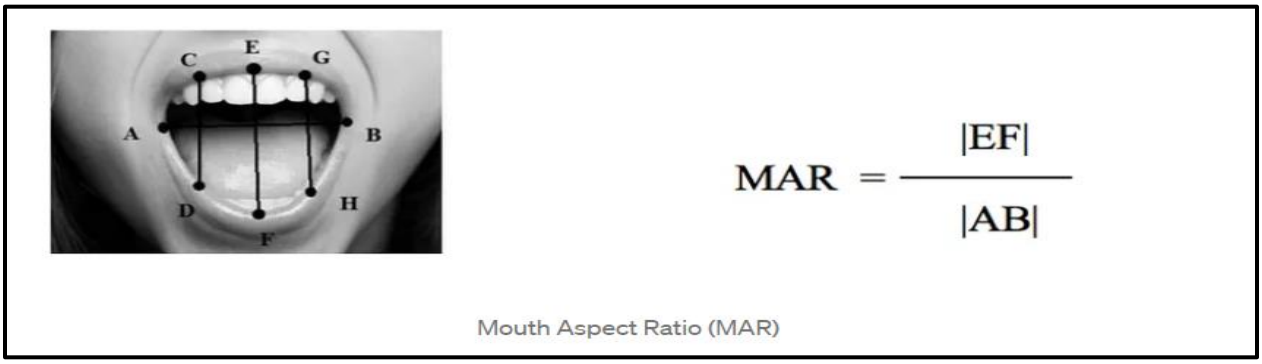

Figure 42 Mouth Aspect Ratio (MAR)

- **Pupil Circularity (PUC)**

PUC is a measure complementary to EAR, but it places a greater emphasis on the pupil instead of the entire eye.

$$Circularity = \frac{4 * \pi * Area}{perimeter^2} \qquad Area = \left(\frac{\text{Distance(p2, p5)}}{2}\right)^2 * \pi$$

$$Perimeter = Distance(p1, p2) + Distance(p2, p3) + Distance(p3, p4) + Distance(p4, p5) + Distance(p5, p6) + Distance(p6, p1)$$

Pupil Circularity

Figure 43 Pupil Circularity

For example, someone who has their eyes half-open or almost closed will have a much lower pupil circularity value versus someone who has their eyes fully open due to the squared term in the denominator. Similar to the EAR, the expectation was that when an individual is drowsy, their pupil circularity is likely to decline.

- **Mouth aspect ratio over Eye aspect ratio (MOE)**

Finally, we decided to add MOE as another feature. MOE is simply the ratio of the MAR to the EAR.

$$MOE = \frac{MAR}{EAR}$$

Mouth Over Eye Ratio (MOE)

Figure 44 Mouth Over Eye Ratio (MOE)

The benefit of using this feature is that EAR and MAR are expected to move in opposite directions if the state of the individual changes. As opposed to both EAR and MAR, MOE as a measure will be more responsive to these changes as it will capture the subtle changes in both EAR and MAR and will exaggerate the changes as the denominator and numerator move in opposite directions. Because the MOE takes MAR as the numerator and EAR as the denominator, our theory was that as the individual gets drowsy, the MOE will increase.

While all these features made intuitive sense, when tested with our classification models, they yielded poor results in the range of 55% to 60% accuracy which is only a minor improvement over the baseline accuracy of 50% for a binary balanced classification problem. Nonetheless, this disappointment led us to our most important discovery: the features weren't wrong, we just weren't looking at them correctly.

6.1.3 Feature normalization

When we were testing our models with the four core features discussed above, we witnessed an alarming pattern. Whenever we randomly split the frames in our training and test, our model would yield results with accuracy as high 70%, however, whenever we split the frames by individuals (i.e. an individual that is in the test set will not be in the training set), our model performance would be poor as alluded to earlier.

This led us to the realization that our model was struggling with new faces and the primary reason for this struggle was the fact that each individual has different core features in their default alert state. That is, person A may naturally have much smaller eyes than person B. If a model is trained on person B, the model, when tested on person A, will always predict the state as drowsy because it will detect a fall in EAR and PUC and a rise in MOE even though person A was alert. Based on this discovery, we hypothesized that normalizing the features for each individual is likely to yield better results and as it turned out, we were correct.

To normalize the features of each individual, we took the first three frames for each individual's alert video and used them as the baseline for normalization. The mean and standard deviation of each feature for these three frames were calculated and used to normalize each feature individually for each participant. Mathematically, this is what the normalization equation looked like:

$$\textbf{Normalised Feature}_{n,m} = \frac{\textbf{Feature}_{n,m} - \mu_{n,m}}{\sigma_{n,m}}$$

where:
n is the feature
m is the person
$\mu_{n,m}$ *and* $\sigma_{n,m}$ *are taken from the first 3 frames of the "Alert" state*

Normalization Method

Figure 45 Normalization Method

Now that we had normalized each of the four core features, our feature set had eight features, each core feature complemented by its normalized version. We tested all eight features in our models and our results improved significantly.

6.1.4 Classification Methods and Results

After we extracted and normalized our features, we wanted to try a series of modeling techniques, starting with the most basic classification models like logistic regression and Naive Bayes, moving on to more complex models containing neural networks and other deep learning approaches. It's important to note the performance-interpretability tradeoff here. Although we prioritize top-performing models, interpretability is also important to us if we were to commercialize this solution and present its business implications to stakeholders who are not familiar with the machine learning lingo. In order to train and test our models, we split our dataset into data from 17 videos and data from 5 videos respectively. As a result, our training dataset contains 8160 rows and our test dataset contains 2400 rows.

- **KNN**

We tried, K-Nearest Neighbor (kNN, k = 9) had the highest out-of-sample accuracy of 90.1%.

- **Convolutional Neural Network (CNN)**

Convolutional Neural Networks (CNN) are typically used to analyze image data and map images to output variables. However, we decided to build a 1-D CNN and send in numerical features as sequential input data to try and understand the spatial relationship between each feature for the two states. Our CNN model has 5 layers including 1 convolutional layer, 1 flatten later, 2 fully connected dense layers, and 1 dropout layer before the output layer. The flatten layer flattens the output from the convolutional layer and makes it linear before passing it into the first dense layer. The dropout layer randomly drops 20% of the output nodes from the second dense layer in order to prevent our model from overfitting to the training data. The final dense layer has a single output node that outputs 0 for alert and 1 for drowsy. It had the second highest accuracy of 85% and a worse response time of 170 ms.

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_9 (Conv1D)            (None, 6, 64)             256
_________________________________________________________________
flatten_9 (Flatten)          (None, 384)               0
_________________________________________________________________
dense_25 (Dense)             (None, 32)                12320
_________________________________________________________________
dense_26 (Dense)             (None, 16)                528
_________________________________________________________________
dropout_5 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_27 (Dense)             (None, 1)                 17
=================================================================
Total params: 13,121
Trainable params: 13,121
Non-trainable params: 0
```

CNN Model Design

Figure 46 CNN Model Design

Activation Function	Relu/Sigmoid
Optimizer	Adam
Loss Function	Binary Crossentropy
Number of Epochs	100
Learning Rate	0.00001

CNN Parameters

Figure 47 CNN Parameters

6.1.5 Results

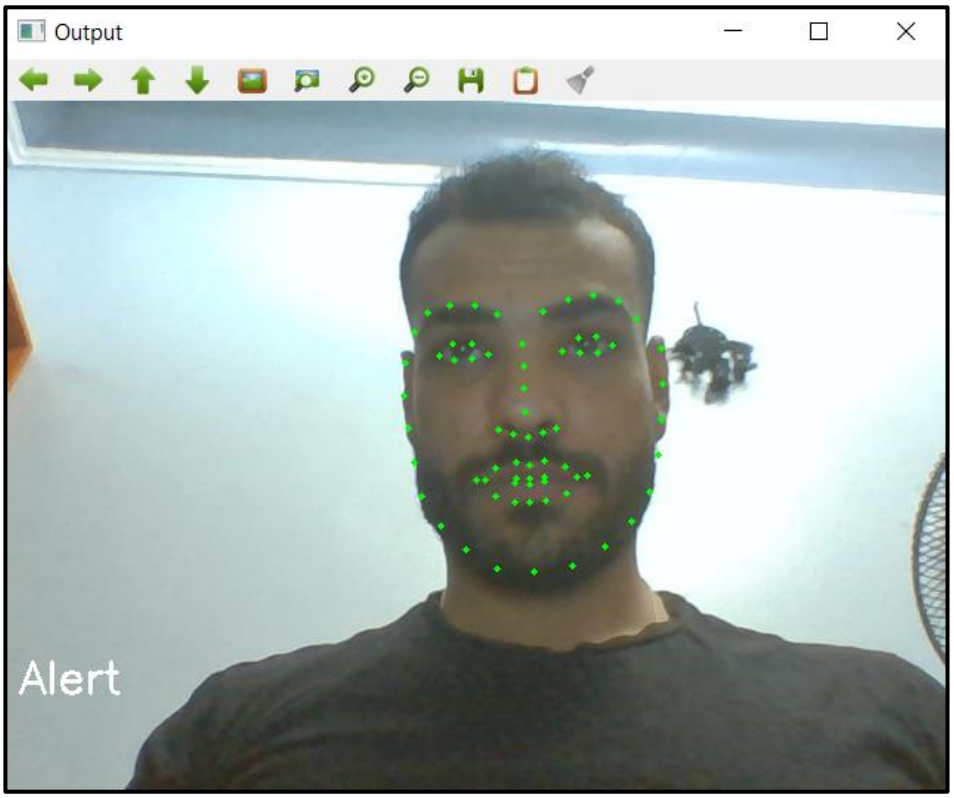

Figure 48 Testing Drowsiness Detection: Alert

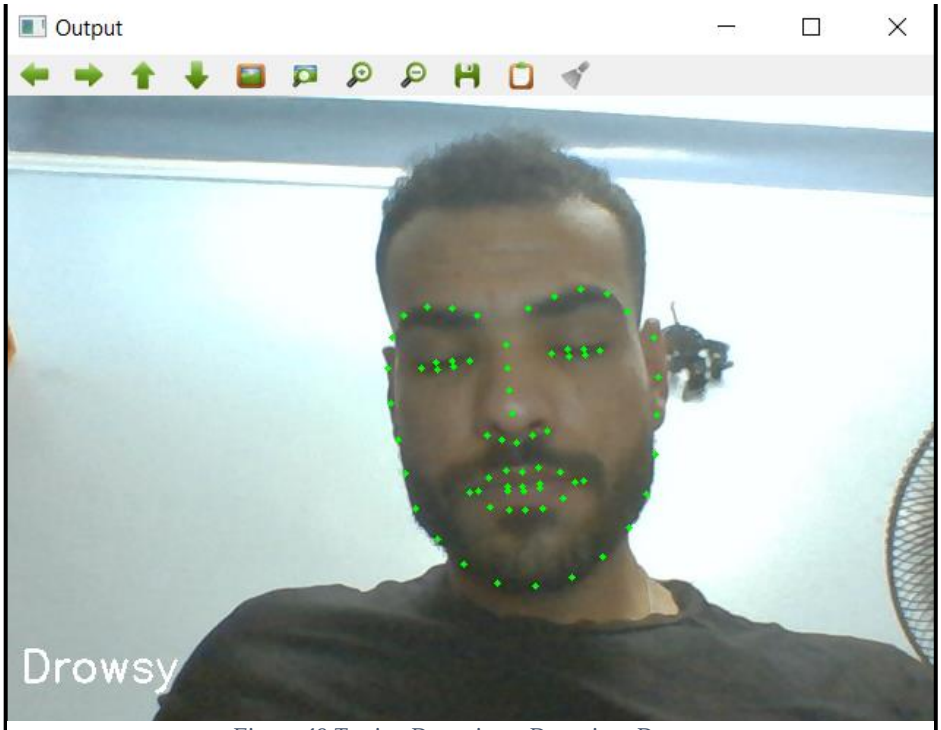

Figure 49 Testing Drowsiness Detection: Drowsy

6.2 Lane detection

As the name suggests, in the field of self-driving and autonomous vehicles lane detection refers to the identification or classification of lines on the road. A common use case is to confirm whether the car is driving in its own lane or has moved into another lane. While this is a simple use case the complex use case can be complete path identification for (self) driving to a particular destination by the autonomous vehicle.

The objective of Lane Detection is essentially to find the two straight lines of the lane, i.e. to find the straight lines in the image and this can be done using the Hough Transform technique and Canny Edge Detection.

- **Hough Space**

Let us assume we have a straight line and it can be represented as y=mx+c which is quite a common high school maths knowledge. But it can also be represented with the below equation

$$\rho = x*\cos(\theta) + y*\sin(\theta)$$

where –

- p (called rho) represents the distance from the origin
- θ (called theta) is the angle formed by the perpendicular line and horizontal axis measured in the counter-clockwise direction

This straight line can actually be represented with just two values (ρ, θ) and can be plotted in a space known as Hough Space.

As we can see in the illustration below, the straight line that existed in the regular cartesian space is represented with just (ρ, θ) coordinates in the Hough Space.

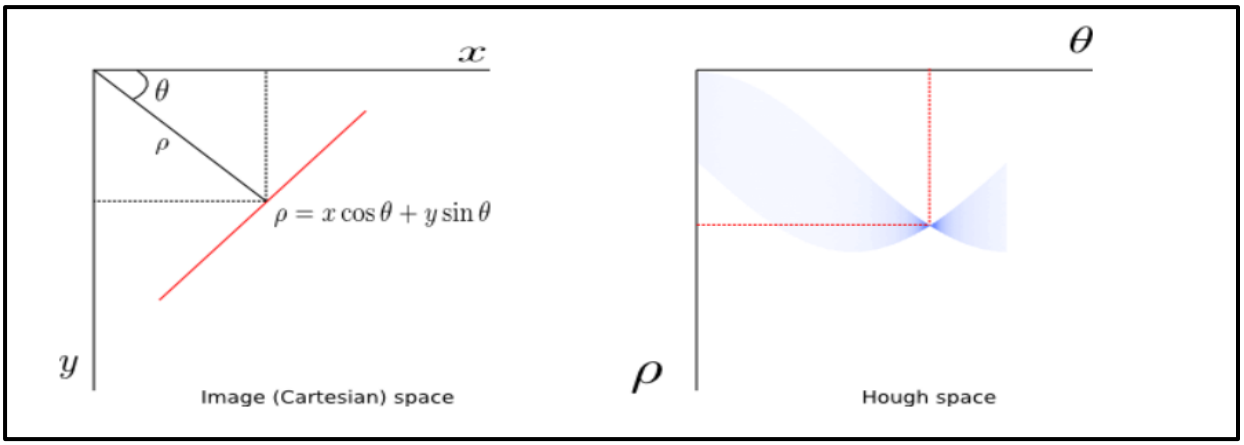

Figure 50 Hough Space

Every line in the original image is converted into the parametric form and represented using only these two values (ρ, θ) in the Hough Space. But why we need this Hough Space?

The intuition behind this is that while scanning the image, the same value of (ρ, θ) will occur many times for a straight line. The occurrences of (ρ, θ) can be accumulated as votes, and finally, when the scanning of the image is done, the (ρ, θ) value which got a high number of votes are identified as a line and can be reconstructed to its actual straight-line form to represent on the image.

To get the maximum performance, it is usually required to pass the image to the edge detector first before applying Hough Transform.

- **Lane Detection Pipeline**

1- Filter out pixels that are not yellow or white (or gray if video is taken during the night/evening).

Figure 51 Lane Detection Pipeline Stage 1

2- Convert image to grayscale and apply Gaussian blur. This is an important step since it reduces noise, and the algorithm for detecting edges (Canny Edge Detection) is susceptible to noise.

Figure 52 Lane Detection Pipeline Stage 2

3- Use the Canny Edge Detection algorithm to detect edges. We're interested in the edges of the lanes.

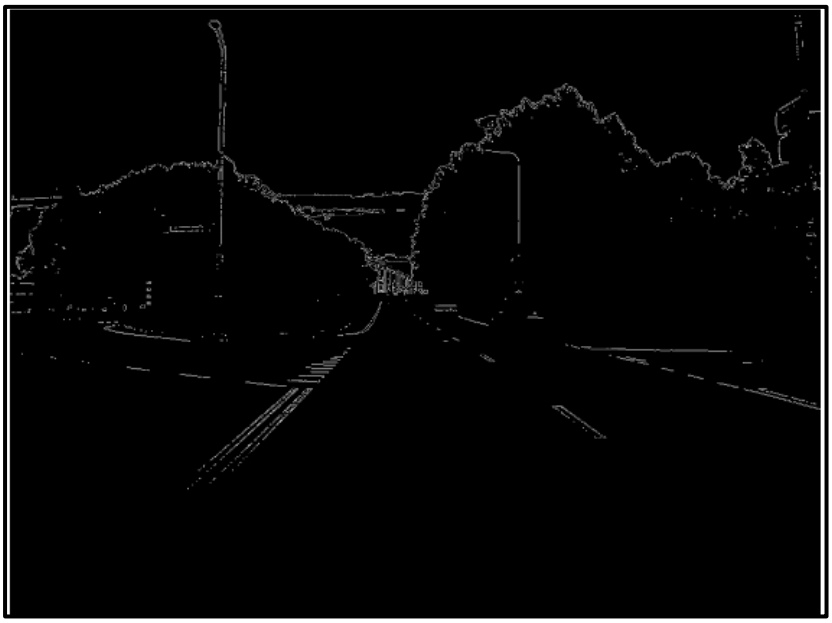

Figure 53 Lane Detection Pipeline Stage 3

4- Find the region of interest. For this program that means creating a mask with the use of two trapezoids (a big one and a smaller one that goes inside the big one). We get the following result after applying the mask on the canny image

Figure 54 Lane Detection Pipeline Stage 4

5- Detect lines using the Hough Line Transform. HLT is used to detect straight lines. Every detected line is made up of two points (starting and ending point). Here are the detected lines drawn

Figure 55 Lane Detection Pipeline Stage 5

6- As you can see in the picture above, the right and left lane consist of multiple detected lines and as said before, a line is just two points. Our goal is to form two lines from those points, one line for the right lane and one for the left lane. This brings us to the fun part: using linear regression to find the line of best fit through the points. To do that, we first filter out the points that form a line with a slope below a certain threshold, then we isolate the left points from the right points and finally we use linear regression to obtain two well-fitting lines. And to make it a bit more clear, we fill in the space between the lines with a lighter color. Result:

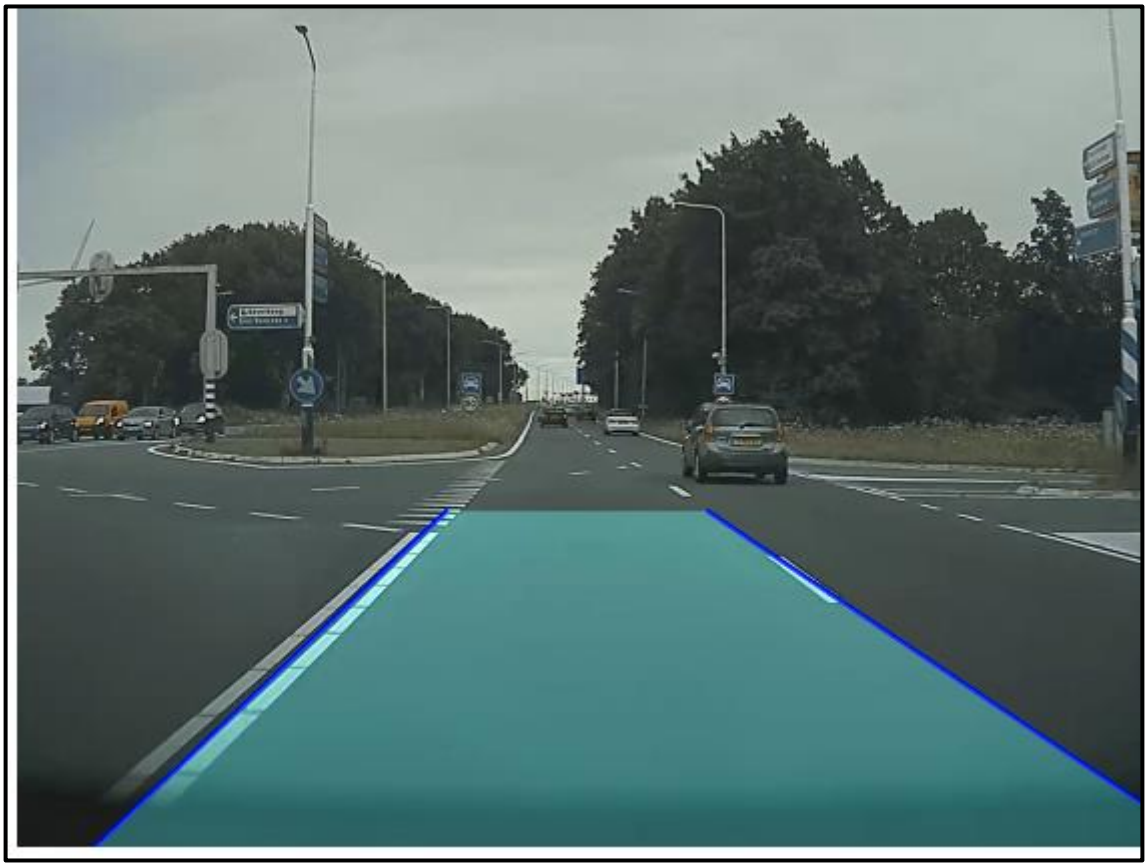

Figure 56 Lane Detection Pipeline Final Stage: output

6.3 Vehicle detection

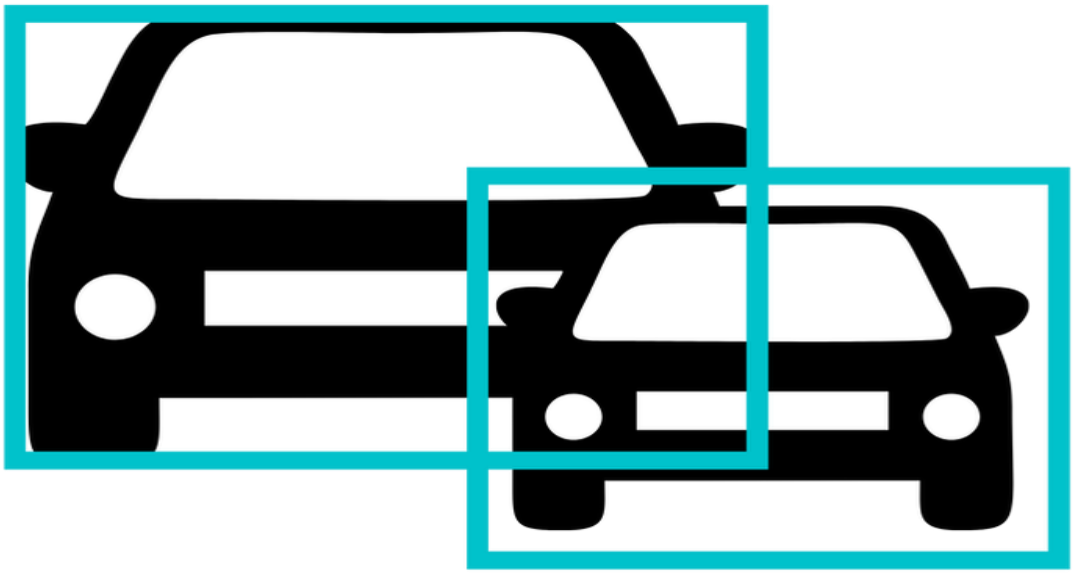

Figure 57 Vehicle Detection animation

Vehicle detection is a critical component of an ADAS which aim to improve the safety and convenience of driving by using technology to assist drivers. It might be considered the most important aspect of ADAS because it allows the system to detect other vehicles on the road and track their movements. By accurately detecting and tracking vehicles in the surrounding environment, ADAS can provide a range of features and can help avoid many catastrophes and accidents. In our project we rely on cameras and machine learning models to do the work.

In this section, we present a model for car detection based on the MobileNetSSD v2 FPN 320x320 architecture. This model is a variant of the popular Single Shot MultiBox Detector (SSD) architecture, which is designed for real-time object detection.

6.3.1 MobileNetSSD

MobileNetSSD is a popular object detection model that uses a lightweight convolutional neural network (CNN) architecture called MobileNet as its backbone network. The goal of MobileNet is to provide a lightweight, low-latency, and low-power CNN architecture that can be deployed on mobile and embedded devices with limited computational resources which in our case is RPI.

MobileNet achieves this by using depthwise separable convolutions, which factorize a standard convolution into a depthwise convolution and a pointwise convolution. The depthwise convolution applies a single filter to each channel of the input, while the pointwise convolution applies a 1x1 convolution to combine the outputs of the depthwise convolution. This reduces the number of parameters and computation required compared to standard convolutions.

Furthermore, MobileNetSSD adds an object detection head on top of the MobileNet backbone network to predict the presence and location of objects in an input image. The object detection head consists of a set of convolutional layers that predict a set of bounding boxes and associated class probabilities for each object in the image. The output of the object detection head is a set of candidate object detections that are filtered and refined using non-maximum suppression (NMS).

In addition, MobileNetSSD has several advantages over other object detection models. It is lightweight and can be deployed on mobile and embedded devices with limited computational resources. It is also fast and can achieve real-time object detection on high-resolution images and videos. Additionally, MobileNetSSD is accurate and can achieve state-of-the-art performance on various object detection benchmarks.

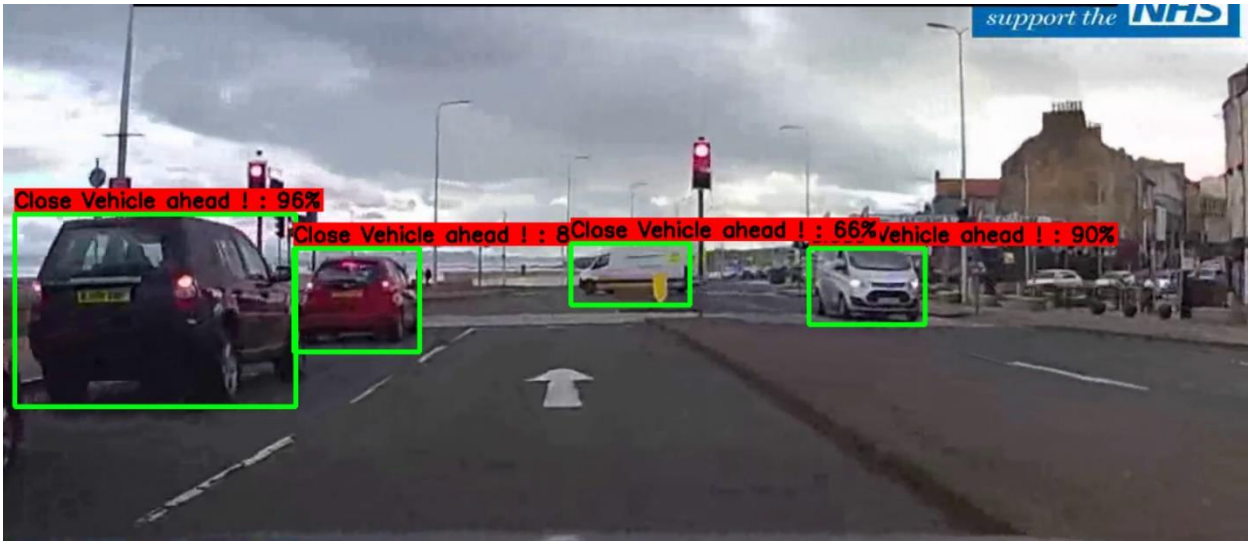

Figure 58 Vechicle detection on dash cam footage 1

What allows MobileNetSSD to perform like so is its unique architecture. Which is based on two key concepts: depthwise separable convolution and feature pyramid network.

Depthwise separable convolution is a type of convolutional layer that is used in MobileNetSSD to reduce the number of parameters and computational resources required for training and inference. In a depthwise separable convolutional layer, the input is first convolved with a depthwise convolution filter, which applies a single filter to each input channel separately. This is followed by a pointwise convolution, which applies a 1x1 filter to the output of the depthwise convolution. By separating the spatial and channel-wise convolutions, depthwise separable convolution reduces the number of parameters and computational resources required, while still achieving good performance.

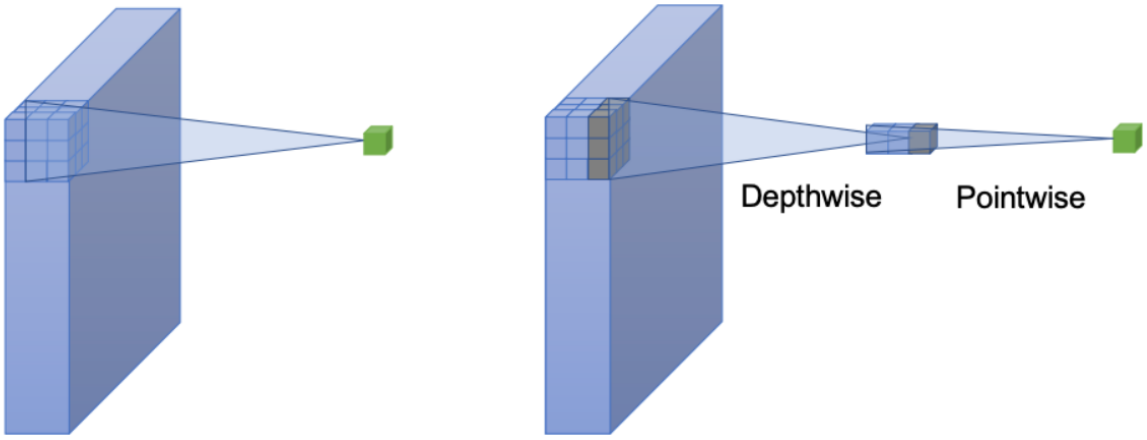

Figure 59 Depth wise seperable convolution

However, depthwise convolution has some limitations compared to regular convolution. One limitation is that it may not be able to capture complex spatial relationships between different channels. Because each channel is processed independently by a separate filter, depthwise convolution may not be able to capture interactions between different channels, which can limit its ability to extract complex features.

6.3.2 Dataset and Tools

For this model, we used the TensorFlow API which provides a lot of utilities like tensorboard which provides a live demo graph of the model's progress through training and most importantly it provides us with the pre-trained checkpoints that we use to fine tune the model and tailor it to our design. The TensorFlow API also allows us to manipulate hyper parameters and that include the learning rate, batch size, number of steps, etc…

We train and evaluate our model on Traffic-Sign-Localization-Detection-SSD-Annotated. This dataset consists of images and videos captured from a driving car. This dataset includes more than 16000 images of different car models and under different conditions.The images in this data set are especially good for our task because other than being all from dash cam and traffic surveillance footages, they are all captured at a certain distancing that is only considered alarming.We also evaluate the model's performance on videos to assess its real-time performance.

We assess the accuracy, precision, recall, and F1 score of our model and compare it with other state-of-the-art car detection models. We also investigate the limitations of our model, such as its sensitivity to lighting conditions and occlusion. The results show that our model achieves high accuracy and outperforms other models in terms of speed and efficiency. Our model can detect cars in real-time scenarios with high accuracy, making it suitable for applications such as autonomous driving and traffic monitoring.

6.3.3 TFLite

The model is then changed into TensorFlow lite of TFLite format, which is a lightweight version of TensorFlow, Google's open-source machine learning framework. TFLite is designed to run machine learning models on mobile and embedded devices with limited computational resources, such as smartphones, tablets, microcontrollers, and IoT devices. TFLite provides a set of tools and libraries for model optimization, conversion, and deployment. In addition, it also supports a wide range of machine learning models, including image classification, object detection, natural language processing, and speech recognition. TFLite provides several advantages over other machine learning frameworks. It is lightweight, fast, and efficient, enabling real-time inference on mobile and embedded devices. It also supports hardware acceleration using GPUs, CPUs, and specialized accelerators, such as Google's Edge TPU.

TFLite provides several tools for model optimization, such as quantization, pruning, and weight clustering. These techniques help reduce the size and complexity of machine learning models, making them more suitable for deployment on mobile and embedded devices. TFLite also provides a model converter that can convert models from other frameworks, such as TensorFlow and PyTorch, to TFLite format. And this format is widely used in various applications, such as object detection, image classification, speech recognition, and natural language processing. It is also used in Google's products, such as Google Lens, Google Assistant, and Google Translate. TFLite has a growing community of developers and researchers, who contribute to its development and share their models and applications.

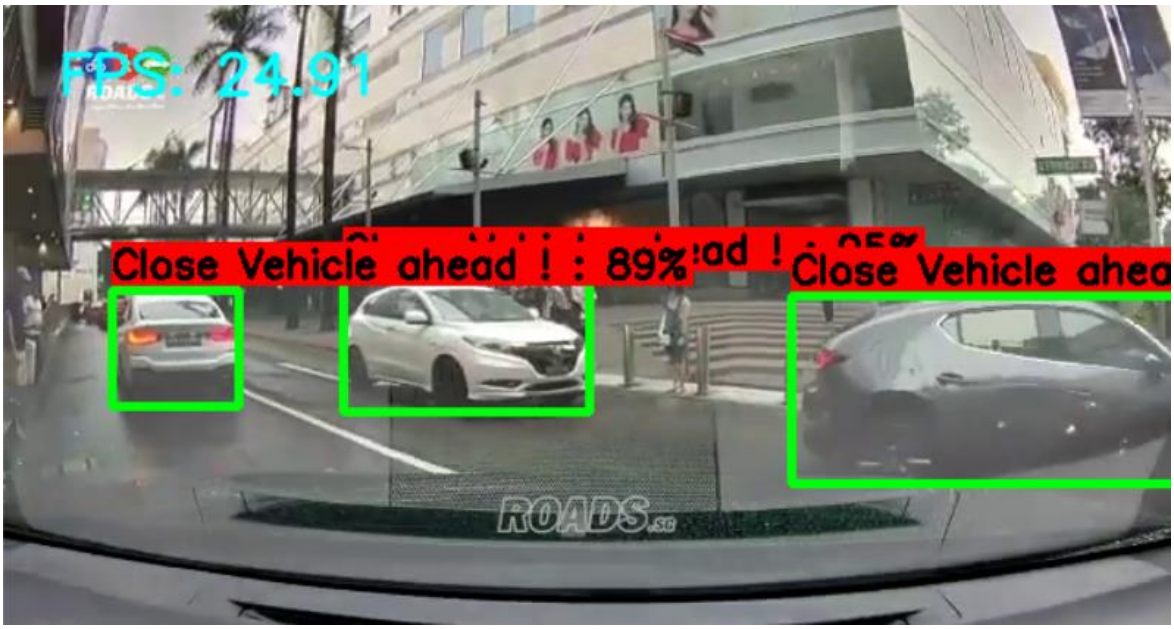

Figure 60 Vechicle detection on dash cam footage 2

6.3.4 Results

The code that we use to run the models includes some of the TensorFlow lite modules including mainly the interpreted in order to load the models that were saved in TFLite format and run them as efficiently as possible.

The TFLite Interpreter is a runtime engine that runs optimized TensorFlow Lite models on mobile and embedded devices. It is designed to be lightweight and efficient, allowing it to run on devices with limited computational resources.

The code also uses OpenCV not for any of the detection mechanisms but for the bounding boxes and alerts that are created on each frame to output a friendly interface and also include some useful information that can come in helpful when testing the model on real world scenarios like the images above.

Most notably, both the code and the model supports the addition of a TPU which are devices that are used to enhance and accelerate the processing power of hardware to tasks like video processing which can very much improve the capability and the speed of the model on targets like RPI as they lack a dedicated Graphical Processing Unit

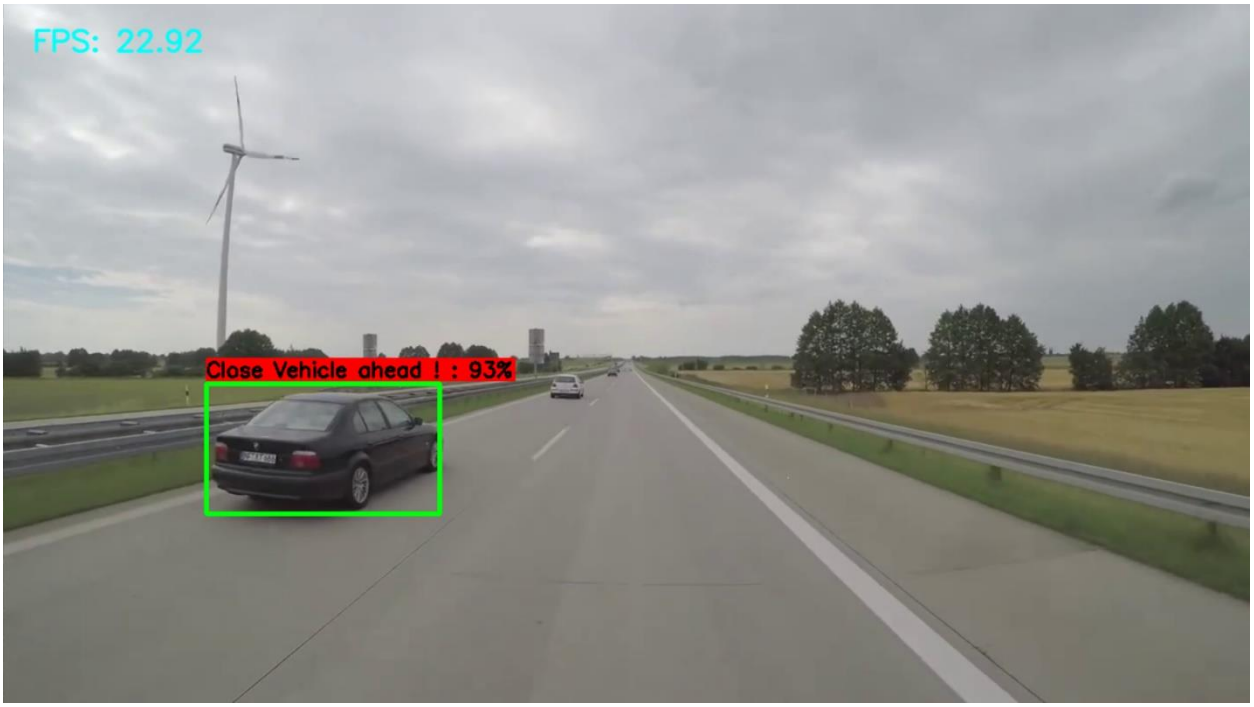

Figure 61 Vechicle detection on dash cam footage 3

6.4 Traffic Signs Detection

Figure 62 Traffic signs detection animation

Traffic sign detection is a crucial aspect of modern transportation systems. Their importance is a result of their capability to provide essential information to drivers, such as speed limits, stop signs, and warning signs, which help them navigate roads safely and efficiently.

A lot of the times these signs can become foggy or hidden which might affect the driver's ability to recognize them, and many other times the driver might fail to recognize or identify them for any other reason that could include fatigue, inattention etc....

Accurate detection of these signs can help improve road safety, reduce traffic congestion, and enhance the overall driving experience. For example, alerting the driver to the speed limit in a specific road might help him avoid a speeding ticket or harm anyone.

By providing timely and accurate information to drivers, traffic sign detection can help prevent accidents and improve overall road safety. It can also improve traffic flow by providing real-time information to drivers about road conditions, such as construction zones, detours, and traffic congestion.

Moreover, traffic sign detection is a critical component of autonomous driving systems, as it enables the safe and reliable operation of autonomous vehicles on roads. Traffic sign detection can

also support smart city initiatives by providing data on traffic patterns, road conditions, and other factors that can be used to optimize transportation systems.

6.4.1 YOLO v5

In this section we will talk about the traffic signs detection and classification model that we created using YOLOv5.

While TFLite did provide a lot of utility and was a great choice for vehicle detection, it falls off when the objects we are trying to detect objects like traffic signs.

Let's now talk about YOLOv5. YOLOv5 is a state-of-the-art object detection algorithm that has gained popularity in the computer vision community due to its high accuracy and fast inference speed. YOLOv5 stands for You Only Look Once version 5, which is one of the later version of the YOLO family of object detection algorithms.

YOLOv5, much like MobileNetSSD, is based on a one-stage object detection approach, which means that it performs object detection and classification in a single pass through the network. This makes it faster and more efficient compared to two-stage object detection approaches, such as Faster R-CNN and Mask R-CNN.

The architecture of YOLOv5 consists of a backbone network, neck network, and head network. The backbone network is a convolutional neural network (CNN) that extracts features from the input image. The neck network is a set of convolutional layers that combine the features from the backbone network to generate a feature map. The head network is a set of convolutional layers that perform object detection and classification on the feature map.

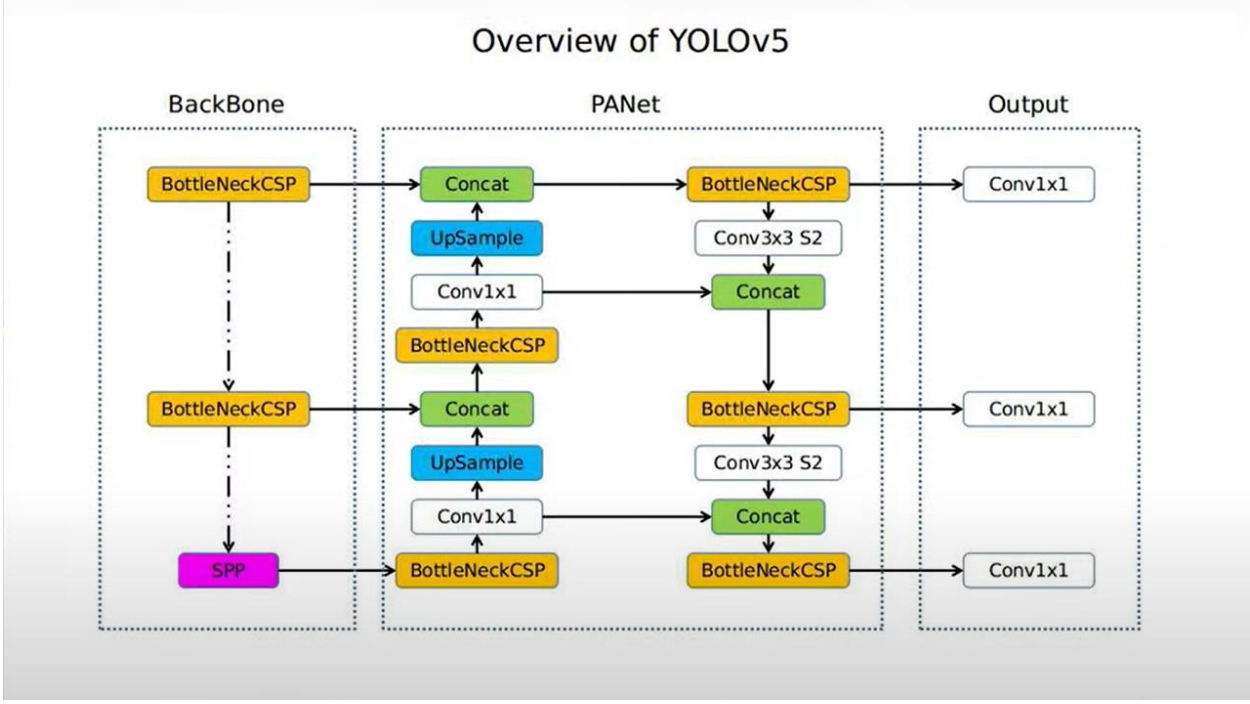

Figure 63 overview of YOLOv5

YOLOv5 uses a technique called anchor-based object detection, where the object detection and classification are performed by predicting a set of bounding boxes and class probabilities for each anchor box. Anchor boxes are predefined bounding boxes of different sizes and aspect ratios that are used to predict the location and size of the objects in the image.

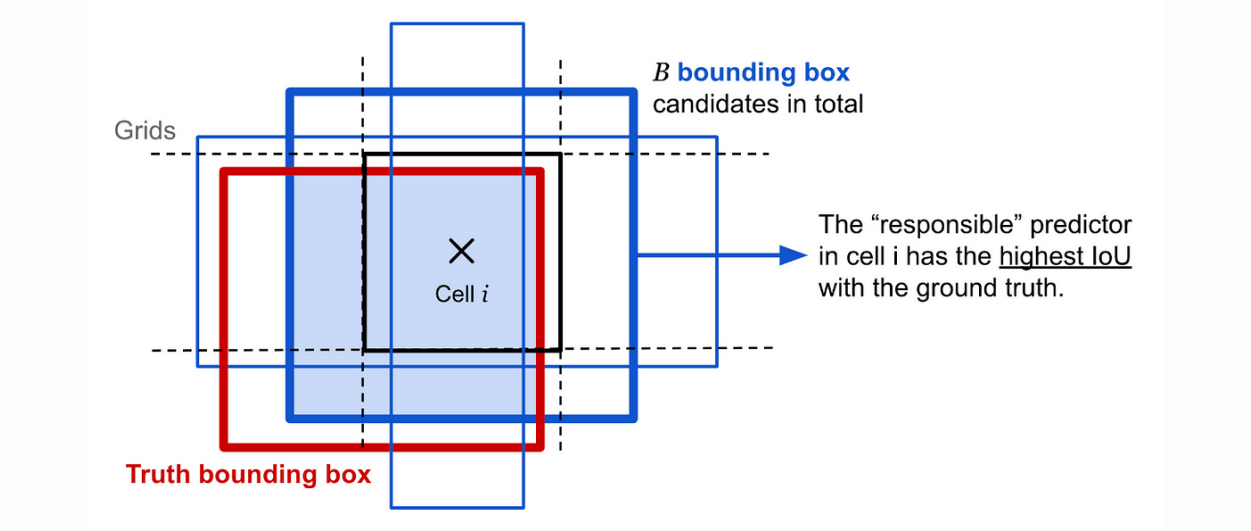

Figure 64 Truth bounding box

One of the key features of YOLOv5 is its high accuracy in object detection tasks. YOLOv5 achieves state-of-the-art results on several benchmark datasets, such as COCO and PASCAL VOC. Additionally, YOLOv5 is optimized for fast inference speed, making it suitable for real-time or near real-time applications.

In addition to its accuracy and speed, YOLOv5 is also highly customizable. It is open-source, allowing developers to fine-tune the model for specific use cases, such as traffic sign detection. This means that you can train the model on your own dataset of traffic signs or adapt it to detect other objects besides traffic signs.

Figure 65 Traffic signs detection on dash cam footage 1

The version that we will be using for YOLO V5 is the Nano version. The YoloV5 Nano (YoloV5n) variant is specifically designed for deployment on resource-constrained devices, such as mobile phones, embedded systems, and edge devices. This model has a smaller size and fewer parameters than other YoloV5 models, making it easier to deploy on devices with limited storage and memory. Additionally, the YoloV5n model has a lower inference latency and reduced power consumption than other YoloV5 models, which is important for real-time applications and battery-powered devices. Despite its smaller size, the YoloV5n model can still achieve high accuracy and good performance on object detection tasks, making it a good choice for many applications where resource constraints are a concern. Therefore, if you are deploying object detection on resource-constrained devices, the YoloV5n variant may be the best choice for your use case.

6.4.2 Dataset and tools

For the dataset, we also used Traffic-Sign-Localization-Detection-SSD-Annotated as it contains annotated images of traffic signs captured from dash-cam videos. The dataset contains over 16,000 images of traffic signs, captured in various lighting and weather conditions, and contains annotations for both traffic sign detection and localization.

The annotations in the dataset include the class label of the traffic sign, as well as the bounding box coordinates of the sign in the image. The dataset contains 23 different classes of traffic signs, including speed limit signs, stop signs, yield signs, and more.

The Traffic-Sign-Localization-Detection-SSD-Annotated dataset is divided into training, validation, and test sets, with 12,319 images in the training set, 3,181 images in the validation set, and 1,568 images in the test set. The dataset also includes a set of negative images, which do not contain any traffic signs, to aid in training and testing of the models.

The classes that are in this data set include:

1. Speed Limit 100
2. Right Curve Ahead
3. Speed Limit 120
4. Speed Limit 80
5. Speed Limit 50
6. Speed Limit 70
7. Stop
8. Road Work
9. Speed Limit 60
10. No Entry
11. Go Straight
12. No Over Taking Trucks
13. Turn Right
14. Turn Left
15. Give Way
16. Pedestrian
17. No Over Taking
18. Slippery Road
19. Huddle Road
20. Danger Ahead
21. Snow Warning Sign

As we mentioned in the section before, advantage of the Traffic-Sign-Localization-Detection-SSD-Annotated dataset is that it contains images captured from real-world scenarios, which makes it more representative of the challenges faced in traffic sign detection in real-world applications. Additionally, the annotations provided in the dataset make it easier to train and evaluate traffic sign detection models.

However, one limitation of the dataset is that it contains a relatively small number of images compared to other datasets, which may limit the accuracy and robustness of the models trained on this dataset. Nevertheless, the Traffic-Sign-Localization-Detection-SSD-Annotated dataset is a valuable resource for researchers and developers working on traffic sign detection and localization.

One of the challenges in training a traffic sign detection model is dealing with classes that are very similar to each other, such as speed limit signs. These signs may have similar shapes and colors, which can make it difficult for the model to distinguish between them.

An approach that we used to address this challenge is to use image augmentation techniques, which involve applying transformations to the training images to create new, slightly different images. By generating more diverse training data, image augmentation can help to improve the robustness and accuracy of the model, particularly for classes that are similar to each other.

In our case, image augmentation techniques used were rotation, translation, and scaling to create variations of the training images of speed limit signs, which would help the model learn to recognize them more accurately. Similarly, techniques such as color jittering, brightness adjustment, and contrast enhancement were used to create variations of the training images with different color and lighting conditions, which would help the model to generalize better to different lighting conditions in the real world.

By using image augmentation techniques, we can create a more diverse and robust training dataset, which can help to improve the performance of the traffic sign detection model, particularly for classes that are similar to each other.

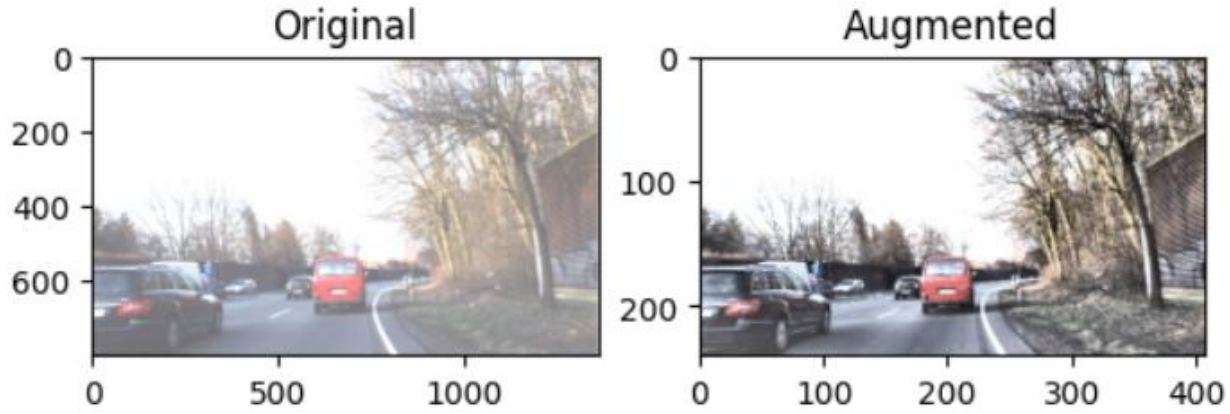

Figure 66 Augmentation

As it can be seen from the above image, we can see the augmentation technique's effect on the image. The size of the image shrunk to 0.4 of its original size and the brightness and color saturation was also altered.

After configuring the dataset to provide the highest accuracy and efficiency possible, we now have to change the model's format to match the requirements of YOLOv5 architecture and that is to have text files include the data of the images instead of the xml files used for SSD.

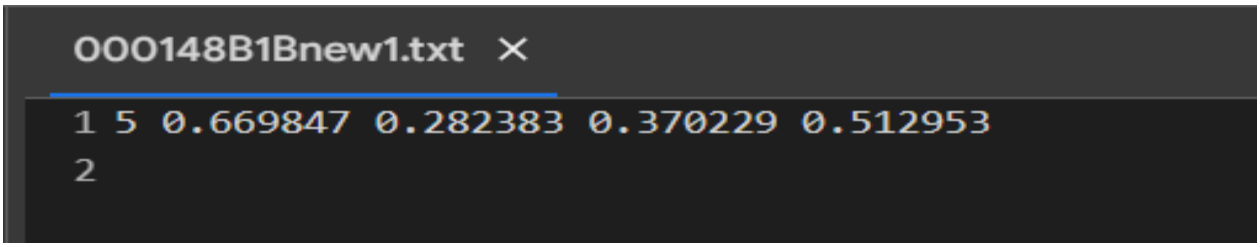

Figure 67 Example of a label file

6.4.3 Results

Finally for the training process, we load YOLOv5n weights file which is a pre-trained model file for the YOLOv5n object detection architecture that contains the learned parameters of the neural network, which were obtained through training on a large dataset of annotated images. The used hyper parameters:

- lr0 (initial learning rate) = 0.01.
- lrf (final learning rate) = 0.01.
- Momentum = 0.937.
- Weight decay = 0.0005
- Warm-up epochs=3.0
- Warm-up momentum=0.8

And the training continued for 30 epochs lasting over five hours and the results came as follow:

Table 3 Mean average percision for each class

Class	Images	Instances	P (TP/TP+FP)	R (TP/TP+FN)	mAP50	mAP50-95
all	2906	3365	0.93	0.898	0.931	0.719
Speed Limit 100	2906	242	0.985	1	0.995	0.878
Right Curve Ahead	2906	86	0.939	1	0.995	0.897
Speed Limit 120	2906	241	0.981	0.992	0.995	0.87
Speed Limit 80	2906	234	0.977	0.996	0.995	0.845
Speed Limit 50	2906	478	0.977	0.992	0.995	0.912
Speed Limit 70	2906	353	0.97	1	0.995	0.885
Stop	2906	48	0.864	1	0.983	0.674
Road Work	2906	84	0.956	0.94	0.986	0.759

Speed Limit 60	2906	342	0.993	1	0.995	0.926
No Entry	2906	276	0.986	1	0.995	0.862
Go Straight	2906	25	0.887	0.92	0.983	0.599
No Over Taking Trucks	2906	123	0.998	1	0.995	0.657
Turn Right	2906	92	0.881	0.891	0.942	0.639
Turn Left	2906	19	0.93	0.737	0.902	0.623
Give Way	2906	110	1	0.96	0.981	0.637
Pedestrian	2906	31	0.876	1	0.995	0.674
No Over Taking	2906	77	0.947	0.926	0.99	0.73
Slippery Road	2906	45	0.933	1	0.985	0.716
Huddle Road	2906	20	1	0.937	0.995	0.622
Danger Ahead	2906	64	0.887	0.858	0.935	0.639
Snow Warning Sign	2906	37	0.974	1	0.995	0.707

Figure 68 testing traffic signs detection on images

Figure 69 testing traffic signs detection on images 2

6.5 Traffic Lights Detection

Figure 70 traffic lights

Traffic light recognition is a crucial component of modern transportation systems, as it plays a critical role in ensuring the safety of drivers, pedestrians, and cyclists on the road. Traffic lights are used to regulate the flow of traffic and to prevent accidents at intersections, pedestrian crossings, and other locations where vehicles and pedestrians interact.

Accurate traffic light recognition is essential for many important transportation applications, including autonomous vehicles, traffic flow management, and pedestrian safety. For example, autonomous vehicles must be able to accurately detect and respond to traffic lights to operate safely and efficiently on public roads. Similarly, traffic flow management systems rely on accurate traffic light recognition to optimize traffic flow and reduce congestion.

In addition, accurate traffic light recognition is important for pedestrian safety, as it enables the development of more effective pedestrian crossing systems. For example, some traffic light recognition systems can detect the presence of pedestrians in the crosswalk and adjust the timing of the traffic signal, accordingly, giving pedestrians more time to cross safely.

YOLO v5 Nano was also used to detect traffic lights as it has shown great results in traffic signs detection compared to other models. YOLO v5 is, as we mentioned, known for its high accuracy, fast inference speed, and ease of use. We chose to create a separate model for traffic lights to simulate the features that an ADAS could carry.

Figure 71 testing traffic lights detection on surveilance footage

6.5.1

6.5.1 Dataset and tools

We trained YOLO v5 Nano to detect green and red lights through feeding it 4000+ pictures having more than 2000 images for each class. The images ranged from dash cam images to images from surveillance tapes to give the highest accuracy in both detection and classification.

Figure 72 samples from dataset

6.5.2 Results

The dataset was distributed among the training, testing, and validation in a 80%, 10%, and 10% fashion as it proved to be the best. And the training of the model was done for 100 epochs and with the following hyperparamters:

- lr0 (initial learning rate) = 0.01.
- lrf (final learning rate) = 0.01.
- Momentum = 0.937.
- Weight decay = 0.0005
- Warm-up epochs=3.0
- Warm-up momentum=0.8

And the results came as follows Scoring a mAP (mean average precision) of 0.959:

Figure 73 testing traffic lights detection on images 2

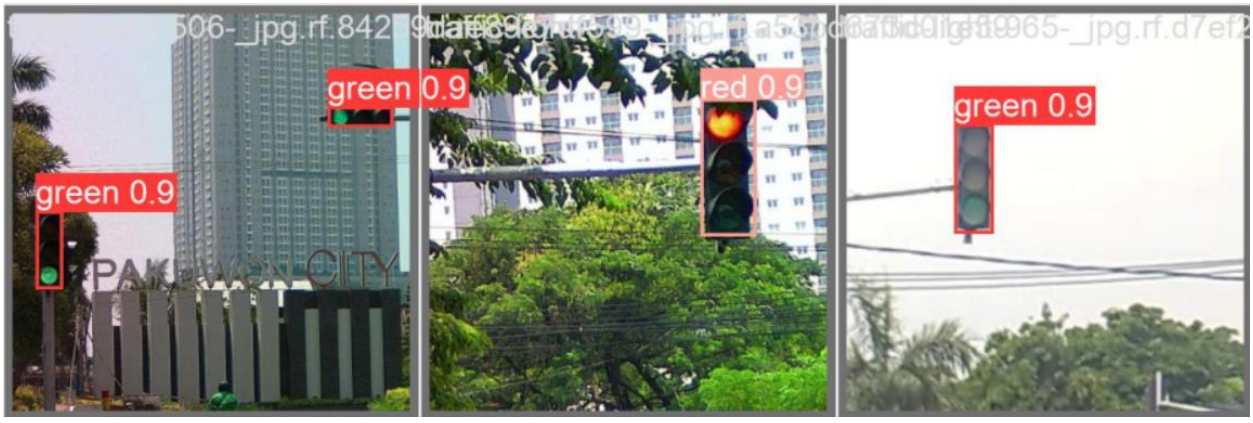

Figure 74 testing traffic lights detection on images 3

Figure 75 testing traffic lights detection on images 4

6.6 Pedestrian Detection

Pedestrian detection is another critical component of Advanced Driver Assistance Systems (ADAS) that aim to improve the safety and convenience of driving by using technology to assist drivers. As we mentioned before, our ADAS uses cameras and machine learning algorithms to detect and analyze the environment around the vehicle and provide real-time feedback to the driver.

Moreover, Pedestrian detection is an important aspect of ADAS because it allows the system to detect pedestrians in the surrounding environment and track their movements. By accurately detecting and tracking pedestrians, we can mitigate a lot of rather catastrophic scenarios. All in all, we conclude that an ADAS would not be complete without a Pedestrian Detection mechanism.

As a result, in this section we introduce our own custom Pedestrian Detection, which similar to the two previous models (Traffic signs and Traffic lights), uses YOLO v5 Nano, which by now should not need any introduction, as the model of choice.

6.6.1 Dataset

For pedestrian detection, the dataset used was one from Roboflow which is a popular online platform for creating and managing custom datasets for computer vision tasks. This dataset we used from Roboflow was a collection of images and annotations from various sources, such as street cameras, surveillance cameras, dash cams, and public datasets.

The dataset contained over 3000 images that mostly contained noise in the form of distortion, color scaling, salt and pepper noise. All the images were annotated with bounding boxes around pedestrians. These annotations were provided in the form of txt files, which is the YOLO format similar to the previous models, including the dimensions of the bounding boxes for the pedestrians.

Most importantly, this dataset was diverse in terms of image quality, lighting conditions, and pedestrian poses, which made it suitable for training a robust pedestrian detection model that could perform well in a variety of real-world scenarios.

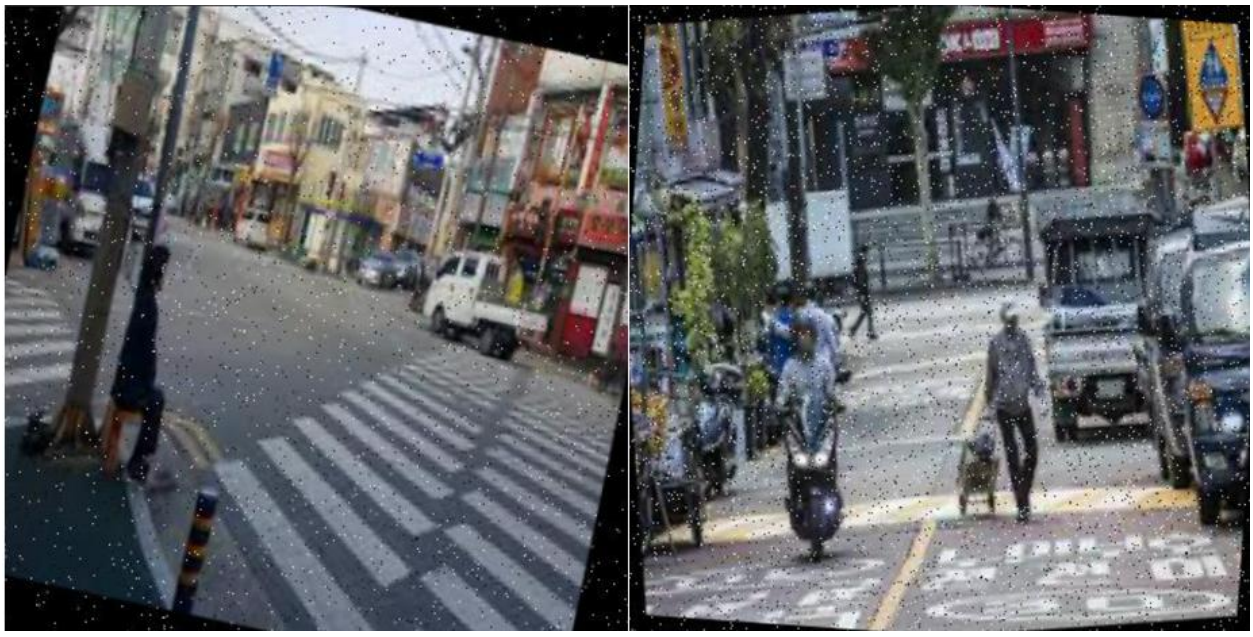

Figure 76 samples from data set

6.6.2 Training

For the training of YOLO v5 Nano, we trained the model for 100 epochs using the same hyper parameters as in the previous models as they proved to be the best for our task:

- lr0 (initial learning rate) = 0.01.
- lrf (final learning rate) = 0.01.
- Momentum = 0.937.
- Weight decay = 0.0005
- Warm-up epochs=3.0
- Warm-up momentum=0.8

And followed the same distribution between training testing and validation datasets to be 80% for training and the rest distributed evenly among the test and validation.

6.6.3 Results

As we mentioned in the above section, the model was trained for 100 epochs as it did net better results than any other, we tried throughout our testing and resulted in a mAP of 0.966

Figure 78 testing pedestrian detection model on images 1

Figure 77 testing pedestrian detection model on images 2

Figure 79 testing pedestrian detection model on images 3

Chapter 7: GUI using Qt

7.1 Famous GUI platforms

7.1.1 Matlab

aMatlab provides a graphical user interface (GUI) for creating and editing code, exploring data, and visualizing results. The GUI is designed to provide an interactive and user-friendly environment for working with Matlab, and includes a range of tools, menus, and widgets for creating and customizing Matlab code and visualizations.

In terms of visualization, Matlab offers a wide range of visualization tools, including 2D and 3D plots, animations, and interactive graphics. Matlab's visualization capabilities are highly regarded in the scientific and engineering communities, and its plotting functions are widely used for data analysis and presentation.

In terms of ease of use, Matlab's GUI is generally regarded as intuitive and user-friendly, with many built-in functions and a comprehensive help system that makes it easy for users to learn and use the software. Additionally, Matlab provides a range of features and tools for automating tasks and streamlining workflows, which can save time and effort for users.

However, Matlab is a proprietary software and requires a license to use. This is a barrier to us as we are working on an open-source or a non-commercial project. Additionally, Matlab can be quite heavy and resource-intensive, which limits its use on small embedded systems or devices with limited resources as we are working on raspberrypi. Matlab's size and resource requirements can also make it challenging to deploy and distribute Matlab applications.

In conclusion, Matlab's GUI provides a powerful and user-friendly environment for working with Matlab code and data, and its visualization capabilities are widely regarded as some of the best in the industry. However, Matlab's proprietary nature, heaviness, and size can be barriers to us.

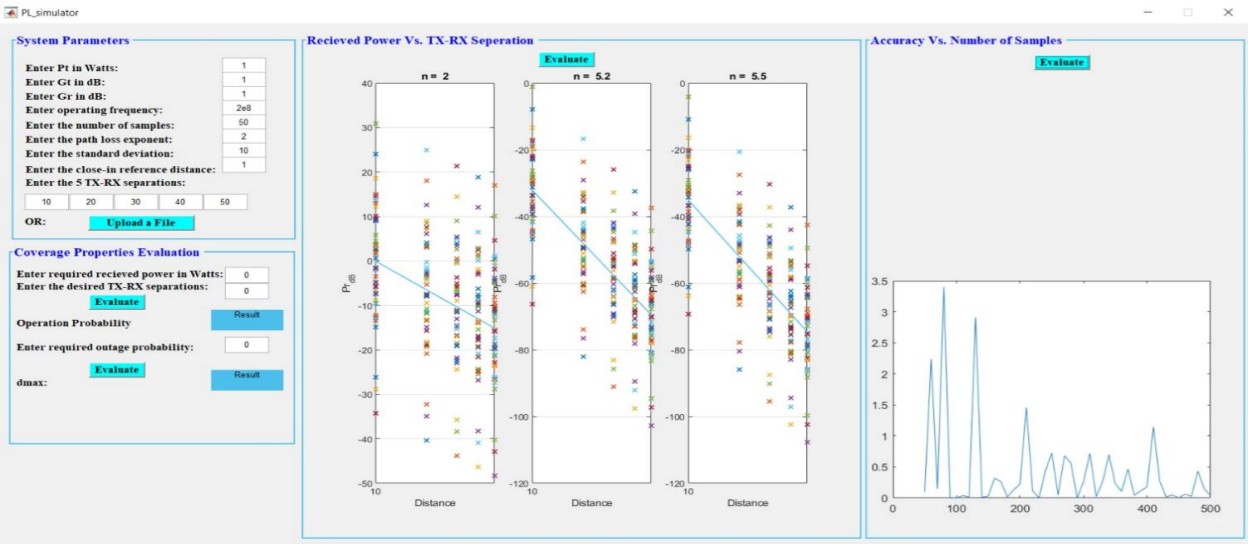

Figure 80 Matlab application example

7.1.2 Tkinter

Tkinter is a Python library for creating graphical user interfaces (GUIs) that is included with Python and does not require any additional installation. Tkinter provides a range of tools and widgets for creating desktop applications with a graphical user interface.

In terms of visualization, Tkinter's visualization capabilities are generally regarded as more basic compared to other GUI frameworks like Qt or MATLAB, but it still provides a range of tools for creating simple graphical user interfaces. Tkinter provides a range of widgets for creating buttons, labels, menus, and other GUI components, but it may require additional libraries or modules to create more complex visualizations.

In terms of ease of use, Tkinter is generally regarded as easy to use and learn, especially for Python developers who are already familiar with the language. Tkinter provides a simple and intuitive interface for creating GUI applications, and the Python language itself is known for its simplicity and ease of learning.

Tkinter is an open-source library that is distributed under the Python Software Foundation License. This means that it can be freely used and modified by anyone, including for commercial or non-commercial projects.

In terms of heaviness and size, Tkinter is generally considered to be lightweight and efficient, with a small footprint and low memory requirements. This makes it a great choice for creating GUI applications on small embedded systems or devices with limited resources.

In conclusion, Tkinter is a lightweight and efficient option that is well-suited for small embedded systems or devices with limited resources. It has an open-source nature and ease of use that make it the best option to those who are new to GUI programming. It provides a simple and easy-to-use GUI framework for creating desktop applications with a graphical user interface, but its visualization capabilities are more basic compared to other GUI frameworks, so this is not just we’re looking for as we are looking for a powerful source of visualization and the best performance.

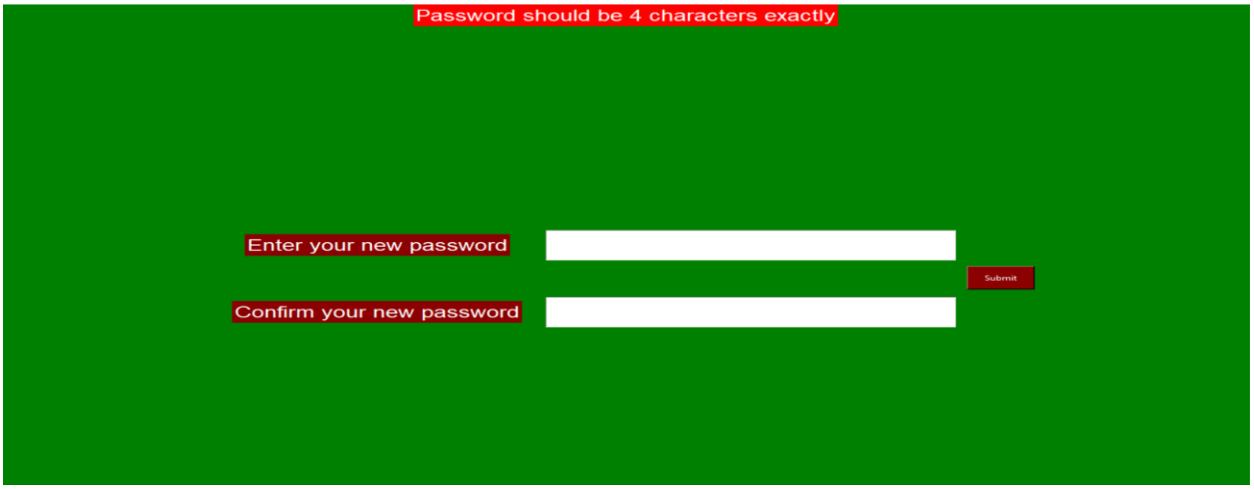

Figure 81 Tkinter application example

7.1.3 Qt

Qt is a cross-platform application and UI framework that provides a wide range of tools and libraries for creating high-quality visualizations. Qt provides a comprehensive set of tools and widgets for creating desktop applications with a graphical user interface (GUI).

In terms of visualization, Qt provides a wide range of visualization tools, including advanced 2D and 3D graphics, animations, and interactive visualizations. Qt's visualization capabilities are highly regarded in the scientific and engineering communities, and its plotting functions are widely used for data analysis and presentation. Qt also provides support for touch screens and other input devices, as well as support for hardware acceleration and graphics rendering.

In terms of ease of use, Qt is generally regarded as easy to use and learn, especially for developers who are already familiar with C++ programming language. Qt provides a range of features and tools for automating tasks and streamlining workflows, which can save time and effort for users. Additionally, Qt provides a comprehensive set of documentation and examples that can help developers get started quickly.

Qt is an open-source framework that is distributed under the GNU Lesser General Public License (LGPL). This means that it can be freely used and modified by anyone, including for commercial or non-commercial projects.

In terms of heaviness and size, Qt provides a lightweight runtime environment that is well-suited for small embedded systems with limited resources. However, writing an application directly in Qt C++ can offer better performance for certain tasks, as C++ is a lower-level language that provides more control over memory management and fine-grained details of the application.

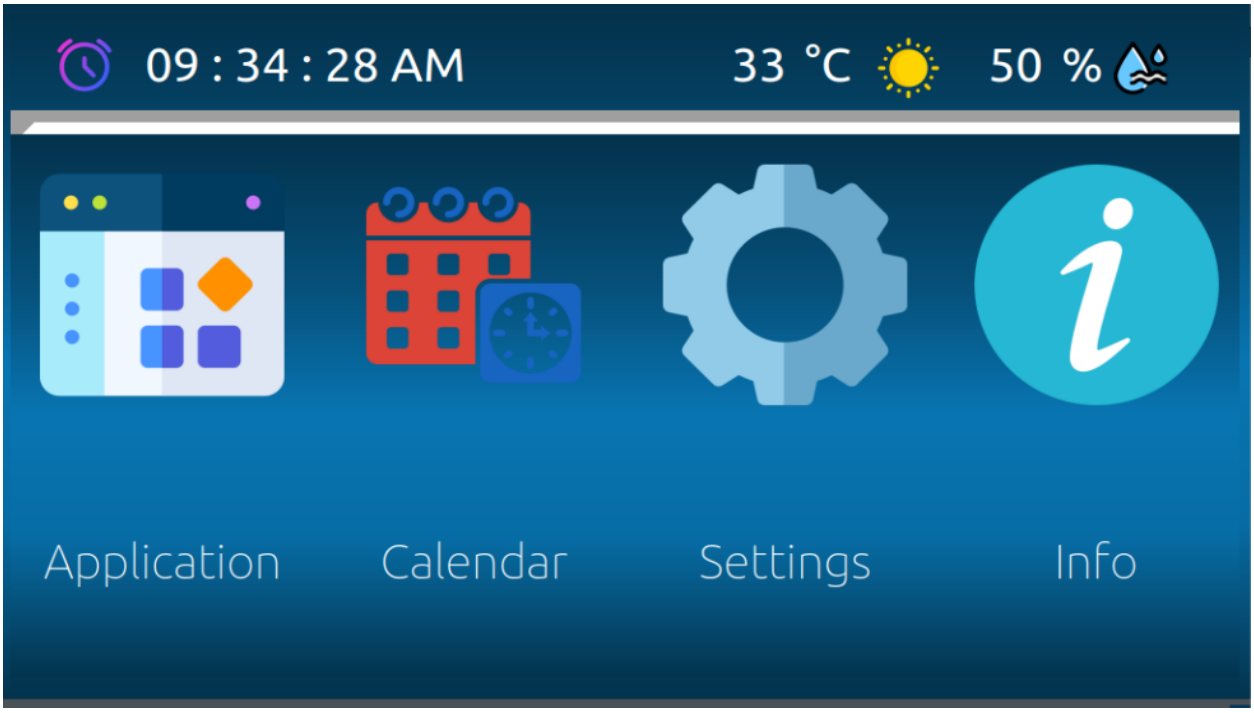

Figure 82 Qt application example

7.1.4 Why to choose Qt

Based on the points mentioned in the previous point, it seems like tkinter and Qt would be the better choices for our embedded device, but Qt would be the better choice regarding visualization. However, it's not the easy option. To conclude each point of interest:

Qt provides advanced visualization capabilities, which can be important for creating high-quality visualizations and user interfaces in embedded devices. Qt's support for touch screens like that we are using and other input devices, as well as support for hardware acceleration and graphics rendering, can also be important features for embedded devices.

Qt is an open-source framework that is distributed under the GNU Lesser General Public License (LGPL). This means that it can be freely used and modified by anyone, including for commercial or non-commercial projects. This can be an important factor for embedded device projects, which often have limited budgets and resources.

Qt is not the easiest to learn, but it'd be the better choice regarding what it offers in return, especially for developers who are already familiar with C++ programming language. Qt provides a range of features and tools for automating tasks and streamlining workflows, which can save time and effort for users. Additionally, Qt provides a comprehensive set of documentation and examples that can help developers get started quickly.

Overall, based on these factors, Qt appears to be a good choice for our project, in addition to that Qt is one of the best choices for small embedded systems devices, which's in our case the raspberrypi.

7.2 Why Qt is a great choice for small Embedded devices

7.2.1 Small footprint

Qt provides a small and efficient runtime environment that is well-suited for small embedded systems with limited resources. This helps to conserve resources and improve performance on devices like Raspberry Pi.

7.2.2 Cross-platform support

One of the main advantages of Qt is its support for cross-platform development, allowing you to write code once and run it on multiple operating systems and hardware platforms. This makes it an ideal choice for embedded systems, where hardware resources are often limited and need to be used efficiently. Qt provides a lightweight, modular runtime that can be customized to include only the features and libraries that are required for a particular application.

7.2.3 Powerful GUI tools

Qt provides a range of tools and libraries for creating high-quality, visually appealing graphical user interfaces (GUIs) on small embedded systems. This includes support for touch screens and other input devices, as well as support for hardware acceleration and graphics rendering.

7.2.4 Active community

Qt has a large and active community of developers who provide support and contribute to open-source projects that help to make Qt a great choice for small embedded systems on Raspberry Pi. This community provides access to a wide range of resources, including documentation, tutorials, and sample code.

7.2.5 Commercial support

Qt is developed and maintained by The Qt Company, which provides commercial support and services for developers who need additional assistance with their projects. This can be especially helpful for small teams or individual developers who may not have the resources to handle all aspects of development on their own.

7.2.6 Powerful set of user interface (Ui) tools

Qt includes a variety of UI components, including buttons, sliders, and text fields, which can be easily customized to fit the requirements of your application. Qt also provides a wide range of UI themes and styles, allowing you to create visually appealing interfaces that are consistent across multiple platforms.

7.2.7 Variety of libraries

Qt also provides a variety of libraries for networking (as openssl that we used to connect with our web server), database access, and multimedia. This makes it a versatile framework that can be used for a wide range of applications, from simple data collection devices to more complex multimedia systems.

7.3 Why did we go for Qt5 widgets C++ over other Qt options?

7.3.1 PyQt vs Qt C++

Qt is available in two main versions: PyQt, which is a Python binding for the Qt application framework, and Qt C++, which is the native C++ implementation of the Qt framework.

In terms of visualization, both PyQt and Qt C++ provide a range of tools and widgets for creating high-quality visualizations and user interfaces, such as charts, graphs, and tables so they’re the same for this point of view. PyQt provides a more Pythonic interface to the Qt API, which can make it easier to work with for developers who are more comfortable with Python than C++. Qt C++, on the other hand, provides more fine-grained control over the application and ensures optimal performance, but requires more expertise and experience in C++ programming.

In terms of ease of use, PyQt is generally regarded as easier to use than Qt C++. This is because PyQt allows developers to create powerful GUI applications using Python, which is known for its simplicity and ease of learning. PyQt also provides a comprehensive set of documentation and examples that can help developers get started quickly. In contrast, Qt C++ requires more expertise

and experience in C++ programming, and may require more development time and effort compared to using PyQt.

Both PyQt and Qt C++ are open-source frameworks that are distributed under the GNU Lesser General Public License (LGPL). This means that they can be freely used and modified by anyone, including for commercial or non-commercial projects.

In terms of heaviness and size, both PyQt and Qt C++ provide a lightweight runtime environment that is well-suited for small embedded systems with limited resources. However, writing an application directly in Qt C++ can offer better performance for certain tasks, as C++ is a lower-level language that provides more control over memory management and fine-grained details of the application.

In conclusion, the choice between PyQt and Qt C++ depends on the specific requirements of the project, the performance needs, the developer's experience and expertise, and other factors such as available resources and tools. While PyQt provides many benefits, such as ease of use and faster development times, Qt C++ can offer better performance and more fine-grained control.

7.3.2 Qt Quick vs Qt Widgets

Qt Quick is a high-level user interface technology that is built on top of the Qt application framework. It provides a set of components, animations, and visual effects that can be used to create modern and responsive user interfaces. Qt Quick is written in QML, a declarative language that allows developers to create user interfaces using a simple and intuitive syntax. QML provides a range of components and elements that can be used to create user interfaces, and allows developers to add logic and interactivity to the user interface using JavaScript code.

QT Quick is a declarative language designed for creating user interfaces using QML, a language similar to CSS and HTML. The declarative nature of QT Quick makes it easy to create complex interfaces quickly and efficiently, and a simple example is shown in **figure 83**.

```
import QtQuick 2.0

Rectangle {
    width: 200
    height: 200
    color: "white"

    Button {
        text: "Click me"
        anchors.centerIn: parent
    }
```

Figure 83 QML example code

In contrast, Qt widgets are the traditional way of creating GUIs in Qt applications and are based on the QWidget class. Qt widgets provide a range of pre-built controls and widgets, such as buttons, labels, and listboxes, that can be customized and arranged to create a user interface. Qt widgets are typically created and manipulated using C++ code and is generally considered to be more suitable for building complex, data-driven applications.

```
#include <QtWidgets>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton button("Click me!");
    button.show();

    return app.exec();
}
```

Figure 84 C++ example code

One of the main advantages of Qt Quick is its ease of use. QML allows developers to create user interfaces using a simple and intuitive syntax, which can be more accessible to developers who are not familiar with C++ programming. Qt widgets, on the other hand, can be more challenging to work with due to their use of C++ code. QML also provides a wide range of components, animations, and visual effects that can be used to create modern and responsive user interfaces. Qt widgets also provide a range of pre-built controls and widgets that can be customized and arranged to create a user interface. However, the range of visualization options may be more limited compared to Qt Quick.

Qt Quick provides a range of features and tools for creating responsive and interactive user interfaces, including animations, transitions, and visual effects. However, Qt widgets may offer better performance and more fine-grained control over the application due to their use of C++. C++ provides more direct control over the memory management and other low-level details, which can result in better performance.

In spite of the fact that Qt Quick is better than Qt widgets in visualization and easy of use, but the fact that QML is not even close to the C++ optimization makes Qt widgets widely adopted in the market and have a large community of developers and resources available for support and development.

In conclusion, the choice between Qt Quick and Qt widgets depends on the specific needs of the project and the preferences of the developer. Qt Quick may be a good choice for creating modern

and responsive user interfaces, while Qt widgets may be a better choice for applications that require more fine-grained control or high-performance. It is important to evaluate the trade-offs between ease of use, range of visualization, and performance when making a decision.

7.3.3 Qt6 vs Qt5

Qt6 is the latest major release of the Qt application framework, which was first released in December 2020. Qt6 includes many new features and improvements compared to Qt5. One of the most significant changes in Qt6 is the improved performance. Qt6 includes various performance improvements, such as faster startup times, reduced memory usage, and improved rendering

Another significant improvement in Qt6 is the enhanced support for C++17. Qt6 includes enhanced support for C++17, which provides developers with access to modern C++ features such as structured bindings, constexpr if, and nested namespaces. This can help developers write cleaner and more maintainable code, with improved performance and functionality.

Our team has already started implementing our project using Qt6, which we believed was the best choice given its many new features and improvements over Qt5. However, during the integration process on our image, we discovered that Qt6 is not yet available on our kirkstone poky branch on yocto, and as a result, we have had to switch to Qt5 to continue with our implementation.

While this has been a setback for our project, we understand that the choice between Qt6 and Qt5 ultimately depends on the specific needs of the project and the resources available. In our case, it was not feasible to continue with Qt6 given the limitations of our current distribution.

Despite this setback, we remain optimistic about the future of Qt6 as it’s announced that it’d be supported soon. In any future projects or updates to our current project where the distribution supports Qt6, we will definitely choose Qt6, given its many new features and improvements.

As a team, we understand the importance of staying up-to-date with the latest advancements in technology and we tried to do our best to evaluate the compatibility and availability of any technology before making a decision to use it. We tried to remain committed to deliver a unique and high quality project using the best technology available.

7.4 Starting with Qt

Installing Qt involves downloading the appropriate version of Qt for your operating system from the Qt website and installing it using the provided installer or package. The Qt website provides installers and other packages for various operating systems. Once you have downloaded the appropriate version of Qt, follow the installation instructions provided by Qt, and make sure to select any additional components or features that you need during the installation process.

After installing Qt, you may need to add additional features or modules to your installation. Qt provides a tool called the “Maintenance Tool” that can be used to manage your installation. You can use the Maintenance Tool to add or remove Qt modules, update your installation, or install additional components. If you need to add a specific module or feature to your installation, you can use the Maintenance Tool to download and install the necessary components.

If you need to compile Qt from source, you can download the source code from the Qt website and then follow the instructions provided in the documentation. Compiling Qt from source can be a complex process, and may require additional dependencies and tools depending on your operating system and the specific version of Qt being used. When compiling Qt, it is important to make sure that you have the necessary development tools installed on your system, such as a compiler and build tools. You may also need to set environment variables or configure your build options to ensure that Qt is built correctly.

The process of Installing, adding features, and compiling Qt can be complex and may vary depending on your specific needs and operating system. However, by following the instructions provided by Qt and carefully evaluating your specific requirements, you should be able to successfully install and use Qt for your development projects. It is important to note that Qt offers extensive documentation and resources to help developers get started and troubleshoot any issues that may arise during the installation or development process.

7.5 Overview about our GUI 2 modes of operation

The GUI application has been designed with two distinct modes: the customer mode and the agency mode. The customer mode is accessible to any car user without the need for any special permissions or access. This mode includes all the features and functions that the user can control, and this features are what the car owner bought the car with, and in our case is some models of our available models.

On the other hand, the agency mode is designed for authorized employees only and is accessible through a button found in the settings. When the user presses this button, a sequence of GUI widgets appears, prompting the user for confidential information to gain access to the car controlling widget. The sequence of widgets is designed to ensure that only authorized employees are granted access to the agency mode.

To ensure the security and confidentiality of the information required for accessing the agency mode, we have implemented strict security measures. All of these measures are via web server and contacting the email who wants to login or enter directly with the emails of the team who has the access to ensure the logins authorizations to ensure the security and integrity of our system from any attacks.

Our project also includes various models that simulate the agency mode, allowing users to purchase additional features and functions and add them to their accessible options found in the settings. This provides users with the flexibility to customize their car's features and functions according to their specific needs and preferences. The models are designed to be user-friendly and intuitive, allowing users to easily navigate through the various options available.

7.6 Customer GUI widgets

7.6.1 Dashboard

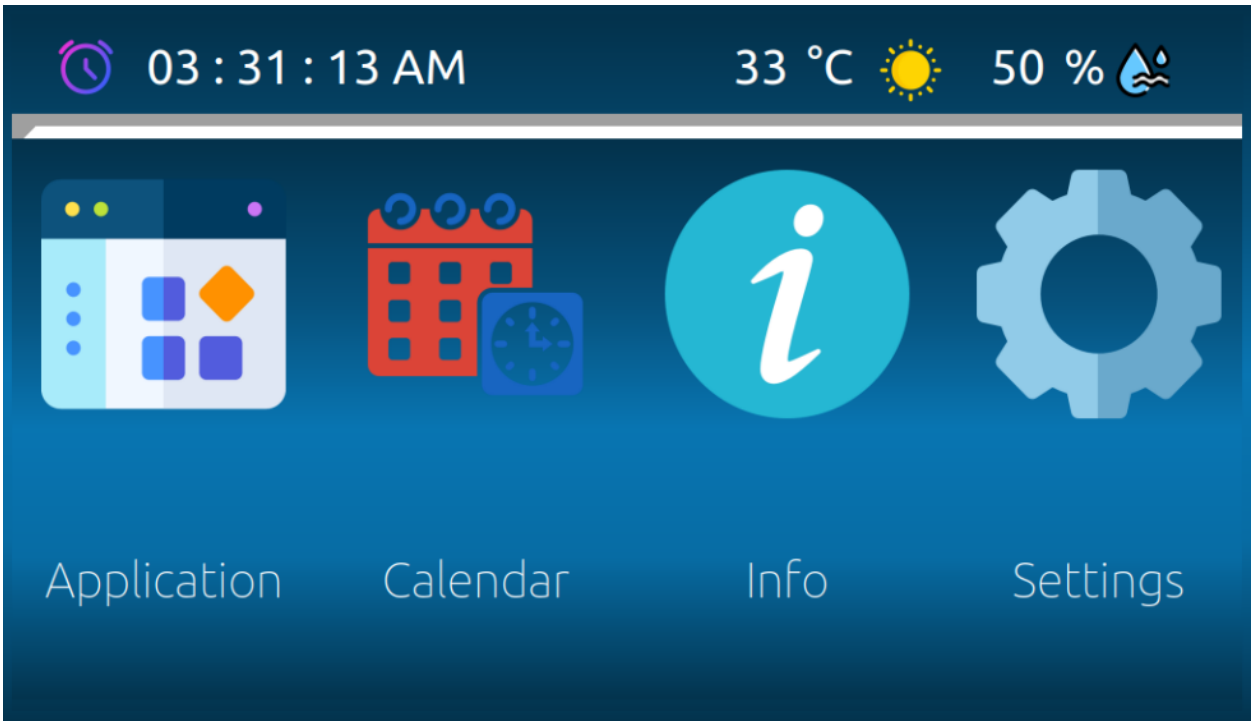

Figure 85 GUI main screen

7.6.2 Clock, temperature, and humidity

There's nothing special about clock or humidity, but temperature is visually different. Clock just returns the current time exactly, while humidity and temperature are measured using dht11 sensor.

Humidity icon is just constant like the clock, but temperature icon changes depending on the current measured temperature. The icon is sunny if the temperature is > 25, cold if < 15 and cloudy if in between, as shown below.

Figure 86 hot weather icon

Figure 87 notmal weather icon

Figure 88 cold weather icon

7.6.3 Application button

The application button serves as a simple and convenient way for users to start and stop the computer vision algorithm. Since our project uses multiple computer vision models, only one model can be available at a time as the project has only 1 camera. For users to select the model they need, they can choose from a list of available models displayed in the GUI when pressing on the settings button that'll be discussed in **8.6.4**. Once the user has selected the desired model, he can press the application button to start the algorithm.

The application button is designed as a toggling button, which means that pressing it once will start the algorithm, and pressing it again will stop the algorithm. This design allows users to easily start and stop the algorithm as needed, without having to navigate through multiple menus or windows.

In addition to its toggling functionality, the application button also provides visual feedback to the user. When the algorithm is running, the button is highlighted with a color to indicate that it is active. T stop the algorithm, they simply press the button again, and the color disappears.

Figure 89 application is one

Figure 90 application is off

7.6.4 Calendar button

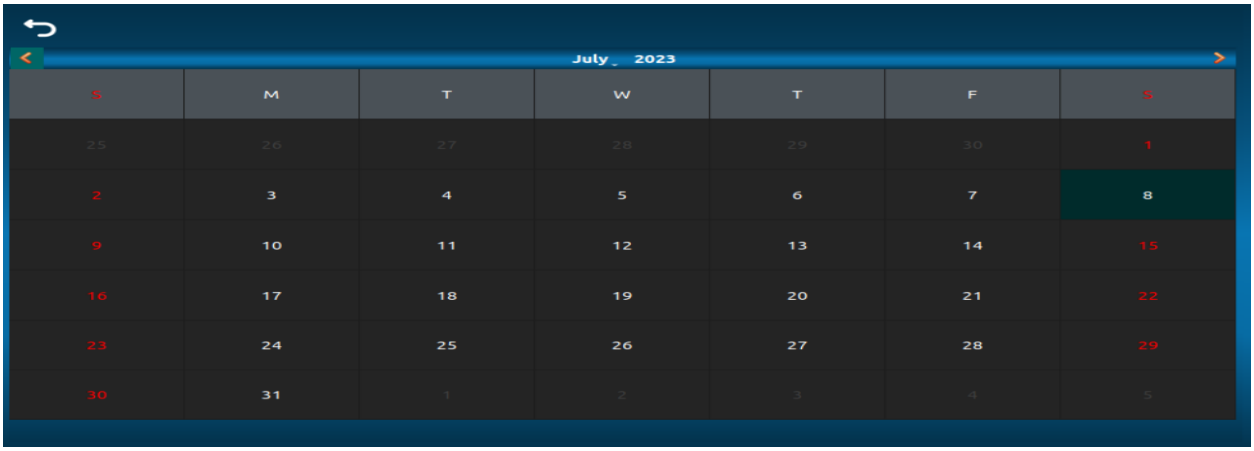

Figure 91 simple calendar

7.6.5 Info button

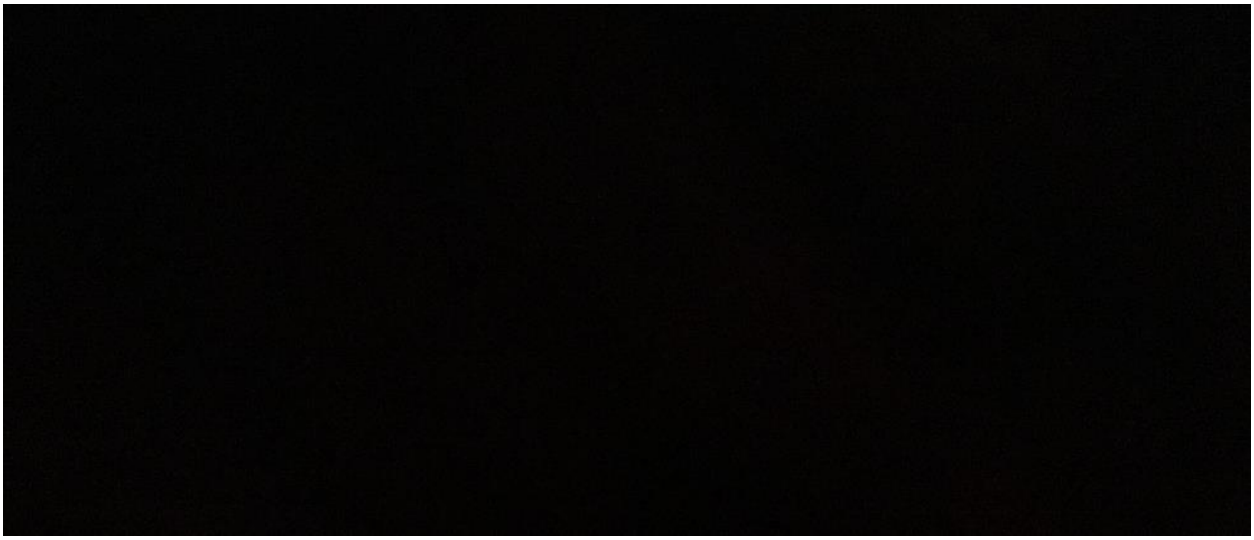

Figure 92 Some information to the user

7.6.6 Settings button

Clicking this button redirects you to the next widget, and you should know that the main window has no more available buttons that redirects you to other widgets except the settings button.

However, the settings widget has 2 buttons which are “BUG REPORT” and “AGENCY”.

Figure 93 widget to allow user control his models

7.6.7 Settings widget

As shown in **figure 93**, On the left-hand side of the GUI, users can find the list of available models that we mentioned earlier. These options were provided to the user by the agency and are displayed as a scrollable list. Users can select the model they need from this list, and once selected, the model is reflected to the application button.

Since the project contains only one camera, users can select only one option from the list at a time. To make the selected model the default application, users can simply press the "save" button located next to the list. This saves the selection and makes the selected model the default for future use.

The saved option is not discarded after closing the application, as users doesn't want the hustle to configure their settings each time they enter their car. It's just saved from the last time the screen got power.

In addition to the list of available models, there is also a "cameras" button located on the right-hand side of the GUI, which displays a pop-up window showing the number of available cameras in the simulated car. This information can be useful for users who need to adjust their computer vision models based on the number of cameras available, but in our case, we only have 1 camera so you can't adjust anything, you can just select 1 model.

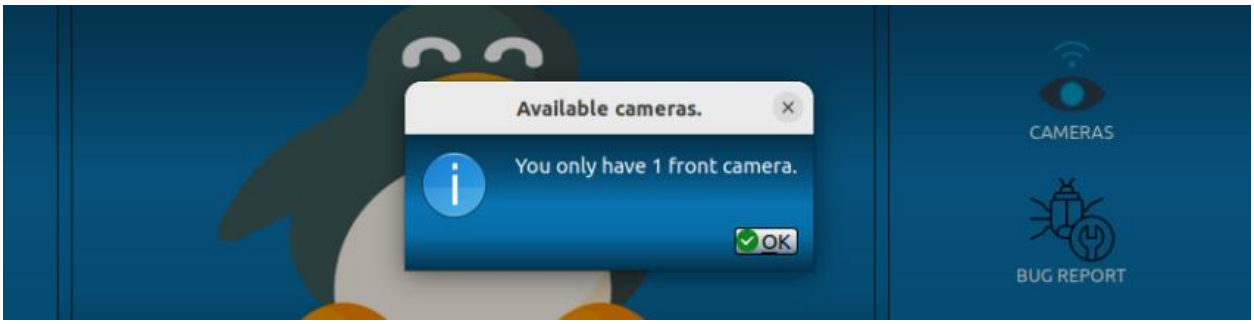

Figure 94 popup to display the number of cameras available

Furthermore, the GUI also features a "bug report" button located directly under the "CAMERAS" button in the window. This button allows users to report any bugs or issues they encounter while using the application. When clicked, the button directs the user to a dedicated page where they can provide detailed information about the issue and submit it for review, this'll be more reviewed in

Finally, the "AGENCY" button, discussed in **8.6.1.2**, is only accessible by the car agency. It provides access to all of the available computer vision models in the car, even those who were given to the customer, or other models that're not included in his car category. When clicked, the button directs the user to a login page that'll be discussed in **8.6.10**.

The GUI provides a user-friendly and intuitive way for users to select and use different computer vision models, with additional features like the "cameras" button, "bug report" button, and agency control to the car to provide any future modifications or subscriptions. The combination of these

features allows users to easily control the algorithms and report any issues encountered for optimal use of the application.

7.6.8 Reporting bugs widget

7.6.8.1 User point of view

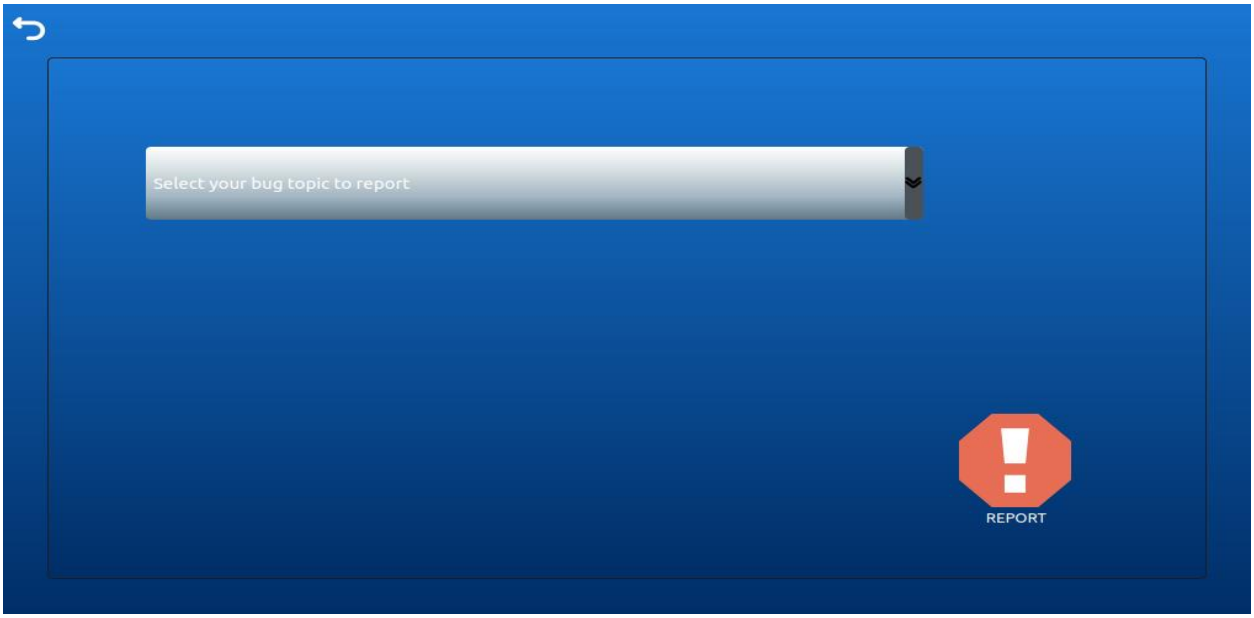

Figure 95 widget to allow user reporting bugs

The "Bug report" button is a critical feature in the **7.7.2** GUI design that allows users to report bugs and issues they encounter while using the software. In the GUI, a dropdown menu is provided for users to select their specific bug from a list of available options. This streamlined approach makes it easier and quicker for users to report their issues than writing their own problem on the touch screen of the car.

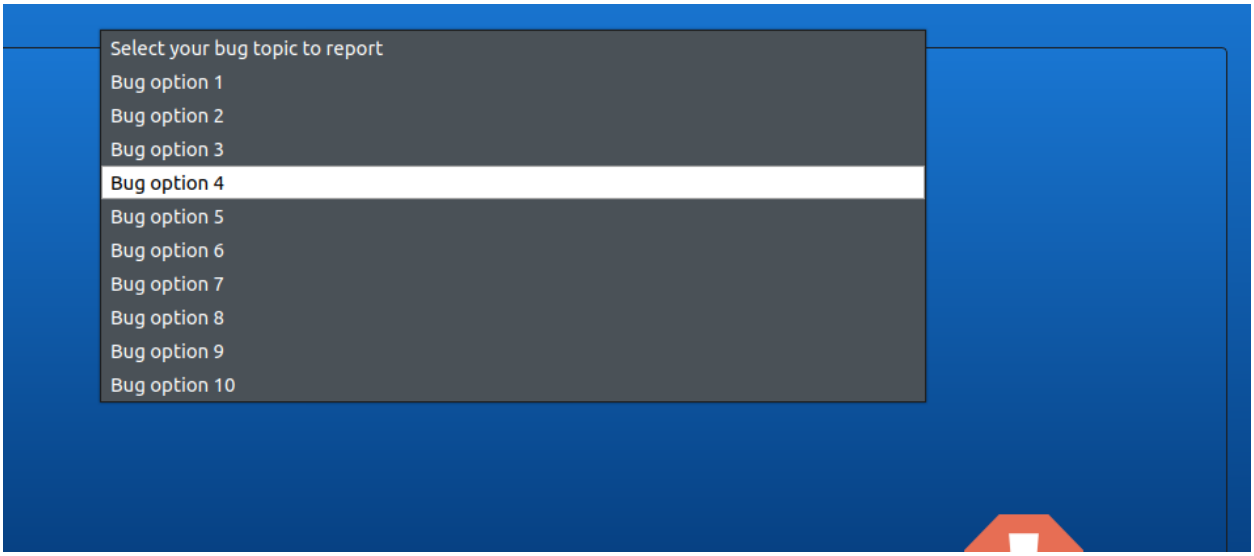

Figure 96 available options for bugs to report

If the user kept the dropdown menu on the first selection, it means that you have not made any choice. This is because the first option in a dropdown menu is usually the default option that appears when the menu is opened. Therefore, if you did not make any changes to the dropdown menu and kept the first selection, you essentially selected nothing.

If the user attempted to click on the report button without selecting any option from the dropdown menu, the process would not proceed. This is because selecting an option from the dropdown menu is a necessary step in the process of generating a report. Without selecting an option, the system does not have enough information to generate a report that meets your specific needs.

As a result, when you click on the report button without selecting an option from the dropdown menu, you will receive a popup message asking you to choose your option. This message is designed to prompt you to select an option from the dropdown menu so that the report generation process can proceed smoothly. Once you select an option, the process will continue, and you will send the report that you requested.

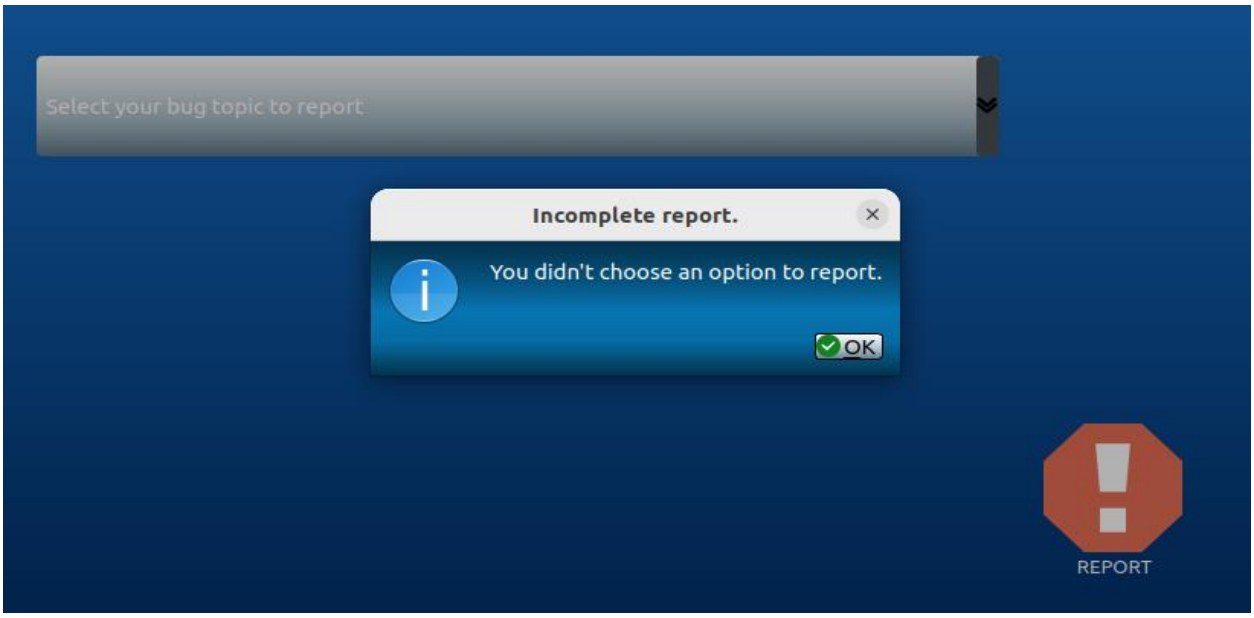

Figure 97 popup if you tried to report an invalid option

Once the user has selected his bug, he can submit his report by clicking the button located at the bottom right corner of the widget. This initiates the reporting process, and the report is sent via the web server to the development team, who can then work towards addressing the issue and they give each report an ID to address the parallel problems easier. This streamlined reporting process helps to ensure that bugs and issues are quickly identified and resolved, which ultimately improves the overall user experience and product quality.

7.6.8.2 Server point of view

Submitting a bug report through the "BUG REPORT" button's GUI is a crucial step in the software development process. However, due to the nature of internet connectivity, there may be a few seconds of delay before the report is successfully sent. During this time, the user may become impatient and try to submit the report multiple times, which can create unnecessary frustration and duplicate reports.

To avoid this issue, we have implemented a feature in the GUI design that links the web reporting submission process to the GUI. After the user presses the submit button, a popup loading menu appears, indicating that the report is being sent. This prevents the user from being able to interact with the GUI until the report has been successfully sent.

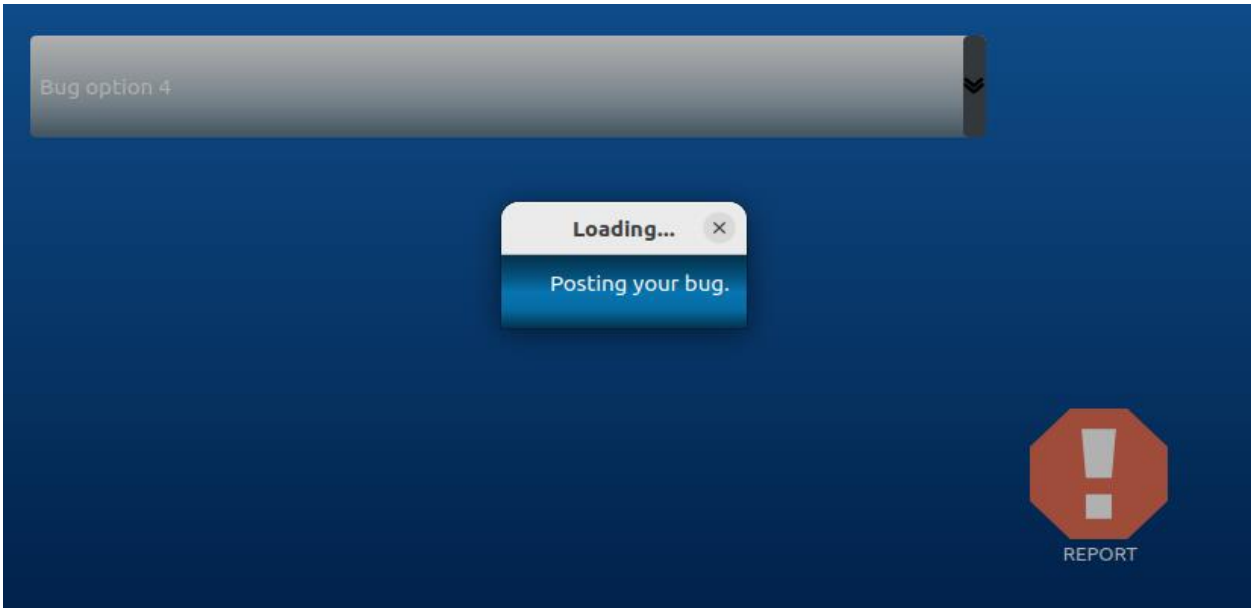

Figure 98 popup to remove frustration while reaching the server

As soon as the report is sent successfully via the server to the development team, a confirmation popup appears, guaranteeing the reception of the report.

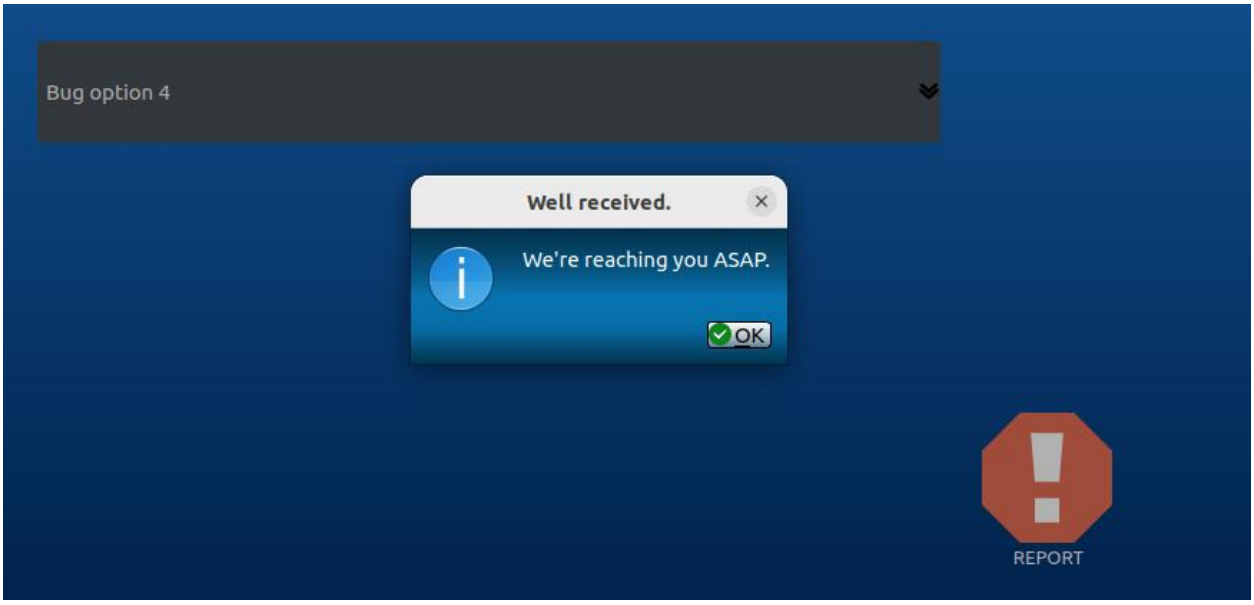

Figure 99 popup to verify that the report has reached out

Each report takes a different ID as shown in **figure 100**.

Figure 100 email received by the technical team

After the user submits his report, the system checks the current database of reported bugs via the web server. This check is done to ensure that the same inquiry is not submitted again while it is still being addressed by the development team. If the user tries to re-submit the same inquiry during this time, a popup message will appear, blocking them from reporting the issue again.

This feature is designed to prevent duplicate bug reports and inquiries, which can slow down the development team's progress in addressing and resolving issues. By blocking users from reporting the same issue again, the development team can focus on addressing each issue once and avoid wasting time on duplicate reports.

The popup message that appears is a helpful reminder that their issue is being addressed and that they need to wait for the development team. This message also helps to ensure that the reporting process remains efficient, which ultimately improves the overall user experience and quality.

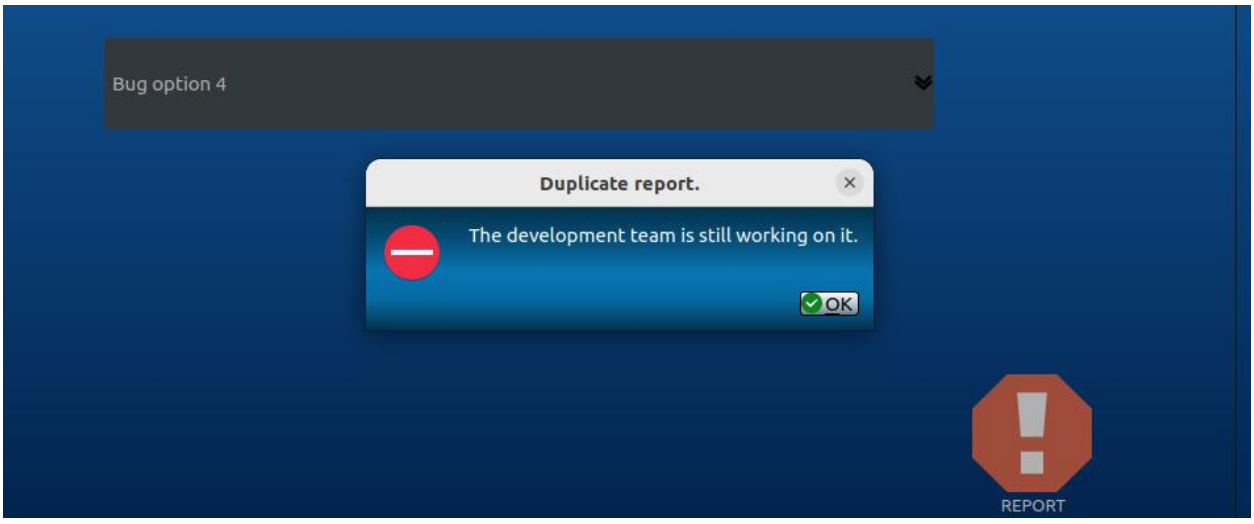

Figure 101 popup to block resending the same bug issue

In the event that a user tries to report a bug while they have no internet connection, the system will be unable to connect to the web server. As a result, the screen will not respond, which can be confusing for the user. To prevent this confusion, after around 10 seconds of failing to connect to the server because of a connection issue, not because of a bad request, a popup message will appear, informing the user of their network issue.

This popup message is designed to help the user understand that their report cannot be submitted due to a network issue, and not because of any problem with their report or the system. The system helps to prevent user frustration and confusion, which can lead to a negative user experience. It also helps to ensure that the reporting process remains efficient and streamlined, which ultimately improves the overall quality of the product.

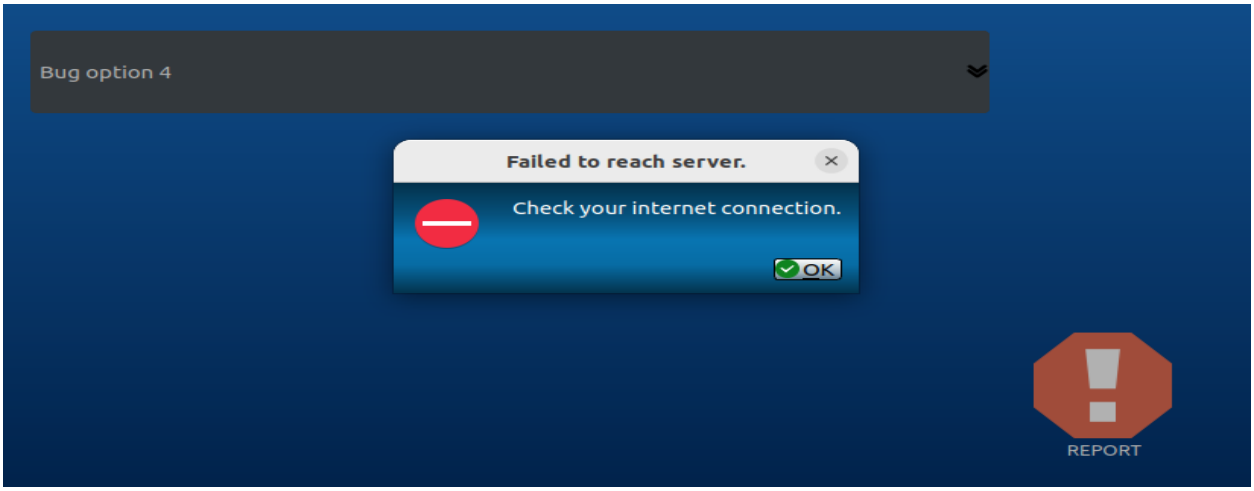

Figure 102 popup to identify weak internet connection

7.7 Sign-in to agency widgets

If a user wishes to change his car's category or subscribe to new features and functions, they must visit the agency in person. An authorized employee with a registered account on our web server will then take account of controlling the user's car's features through the agency button found in the settings. This ensures that only authorized employees have access to the user's confidential information and can make changes to their car's functionality. So, to navigate through this process, we've 2 widgets. One to verify the employee's email, discussed in **7.8.1**, while the other is to verify his password, discussed in **7.8.2**.

7.7.1 Employee's email

Old employees already have a registered account on our server, but for those who are new, who do not have registered accounts on our web server, the GUI offers a simple and user-friendly registration process. Even in case an authorized employee forgets their password, the GUI also offers a password reset button. These features ensure that the GUI application is user-friendly and accessible to all authorized employees.

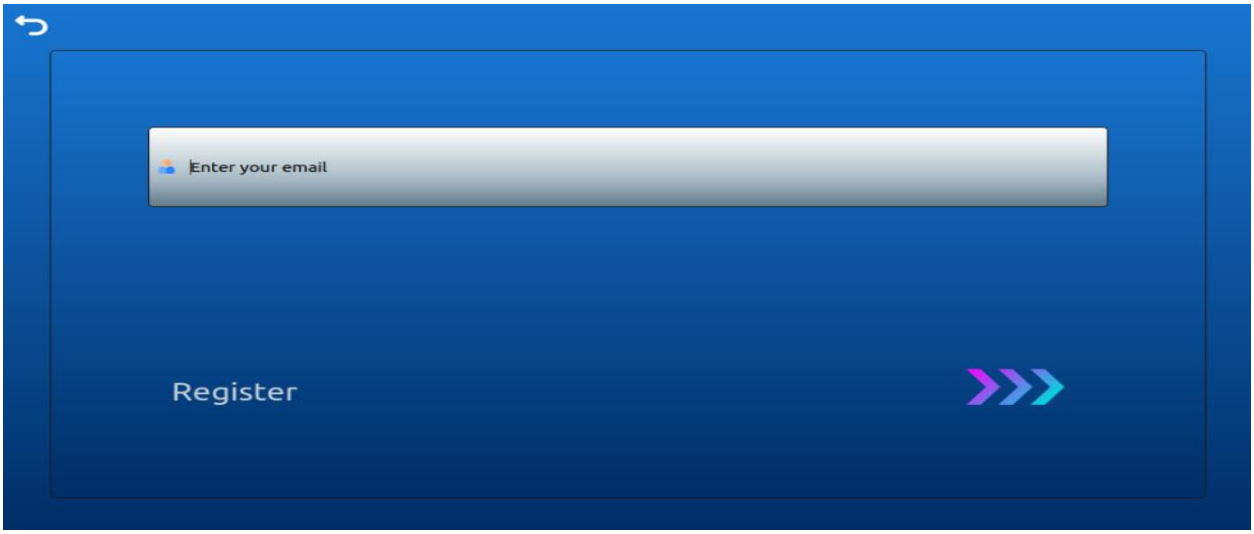

Figure 103 personnel email verification

7.7.1.1 Email submission

When an employee needs to make some configurations in the car's options, the process starts with clicking on the agency button. This action will redirect the employee to a Graphical User Interface (GUI) window that prompts them to enter their email address. The email address is used to ensure that only authorized employees can access and make changes to the car's options.

The GUI window is connected to the web server, which provides the employee with live feedback during the process. This feature ensures that the employee knows that their request is being processed and can help reduce frustration. The live feedback feature is why a loading popup appears as the server processes the employee's request.

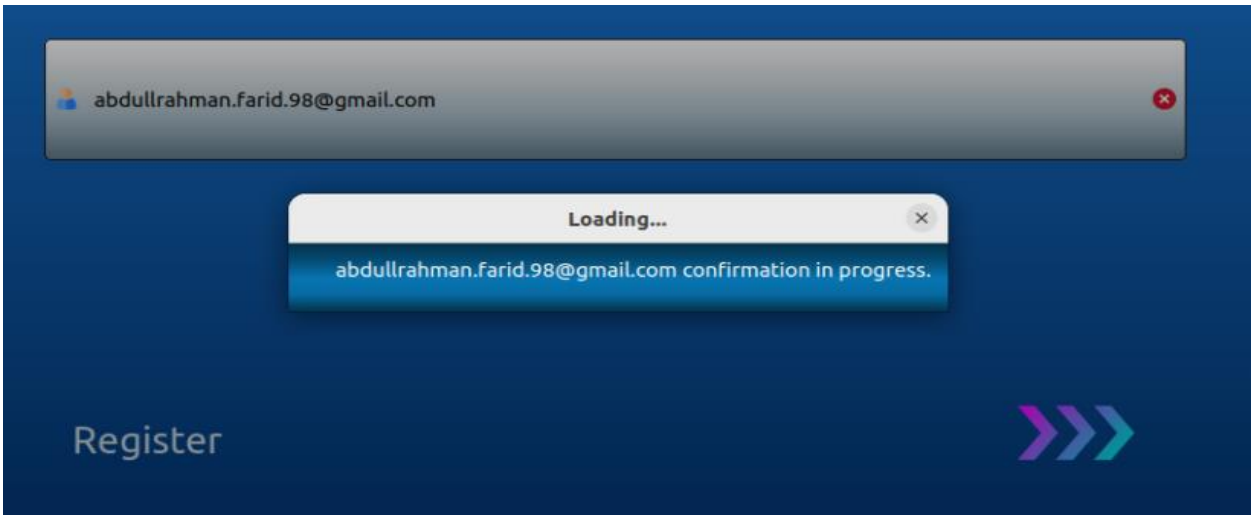

Figure 104 popup to make email verification process interactive to our server

If the employee's email address is found in the server's database, they are redirected to the next window where they confirm their password to start doing configurations in the car's options.

However, if the employee's email address is not registered in the server's database, a popup menu appears, prompting them to register first. This step is essential to ensure that only authorized personnel can access and make changes to the car's options. The registration process guarantees that employees have registration access, and if someone else tries to register, the registration team will deny their request.

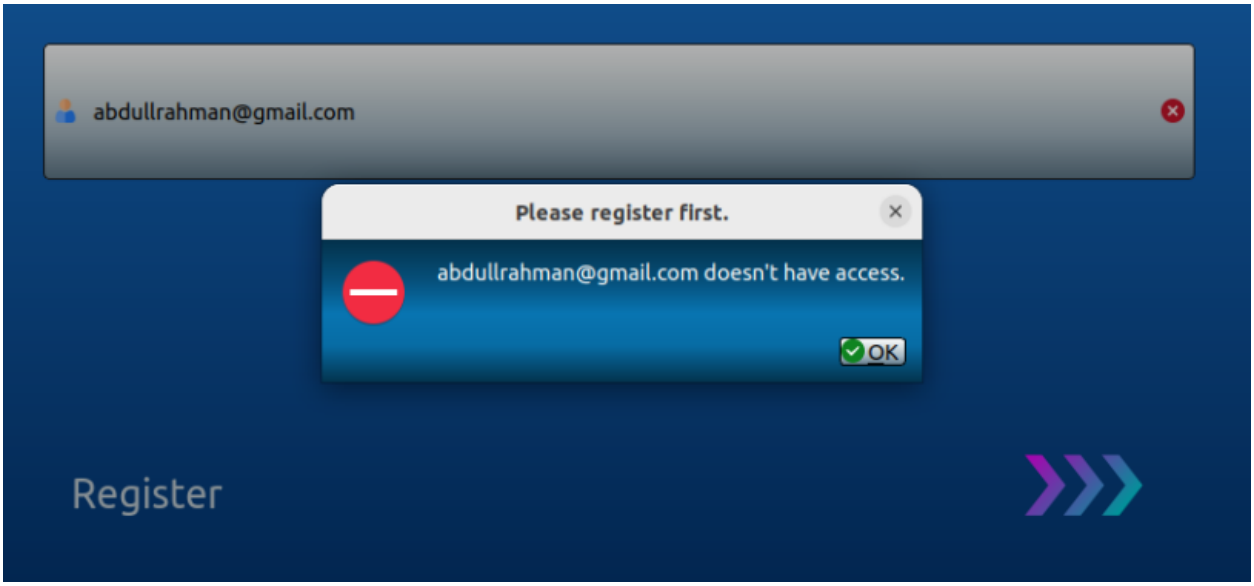

Figure 105 popup if a non-registered email tried to login

7.7.1.2 Email registration

The email registration process is a critical step in ensuring that only authorized employees can access and modify the car's options.

If an employee needs to register an account, they simply enter their email and click on the register button. The system then checks the registered accounts database on the server to see if the email address is registered. If it is not registered, the email is sent to the registration team for verification.

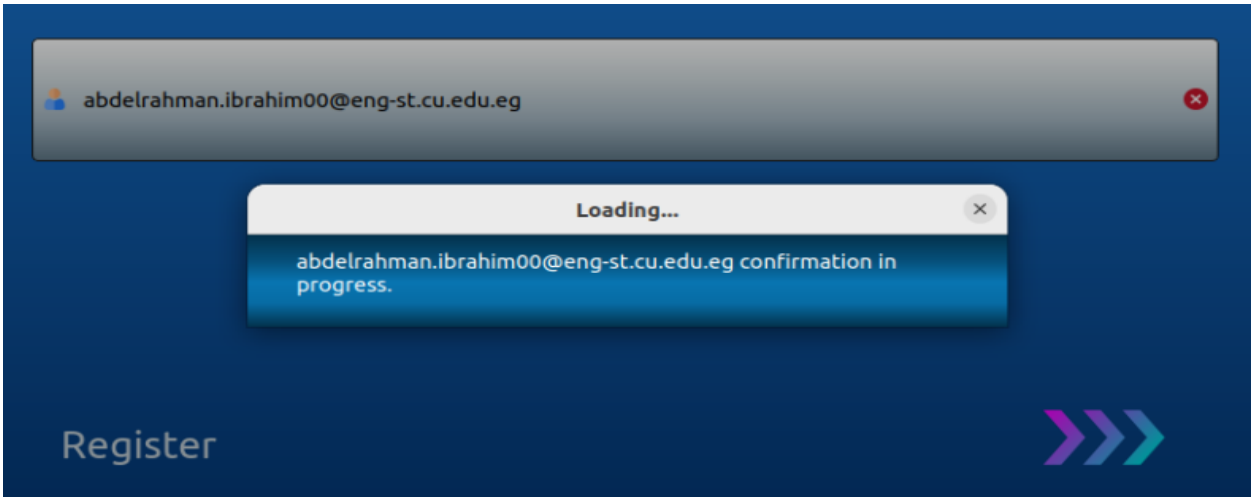

Figure 106 popup to make email verification process interactive to our server as usual

In reality, the registration team would carefully review each email address to ensure that only authorized personnel have access to the car's options. The registration team can verify the identity of the employee and confirm that they have the necessary authorization to access the car's options. This step is critical for maintaining the integrity of the company's data and preventing unauthorized access.

It is important to note that while our demo simplified the registration process so that any valid email is registered, in practice, the registration process is a crucial security measure.

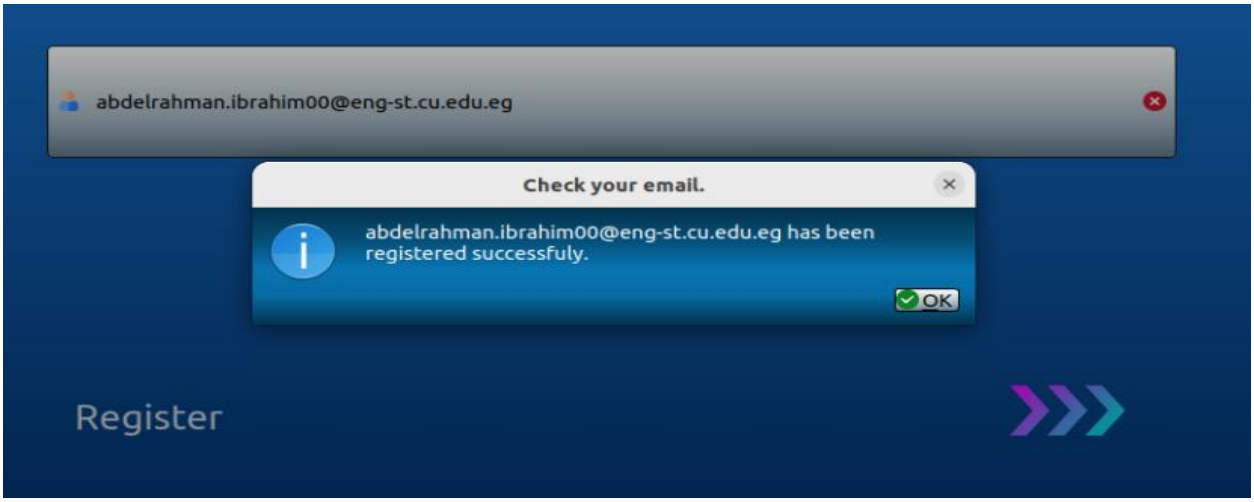

Figure 107 popup to confirm registration and ensure that logging info is sent to this email

If the employee's email address is already registered in the server's database, a popup menu will appear to inform the employee that their email address is already registered.

This step is an important security measure that ensures that employees do not accidentally register multiple accounts. The popup menu helps to streamline the registration process and minimize the risk of errors.

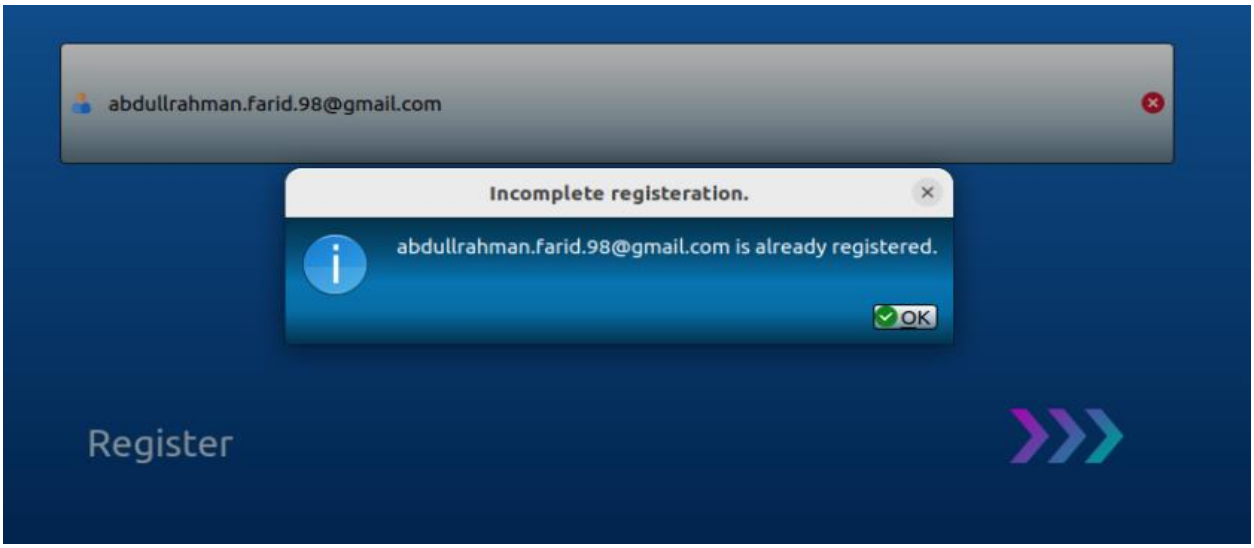

Figure 108 popup to identify the employee that he is already registered on the system

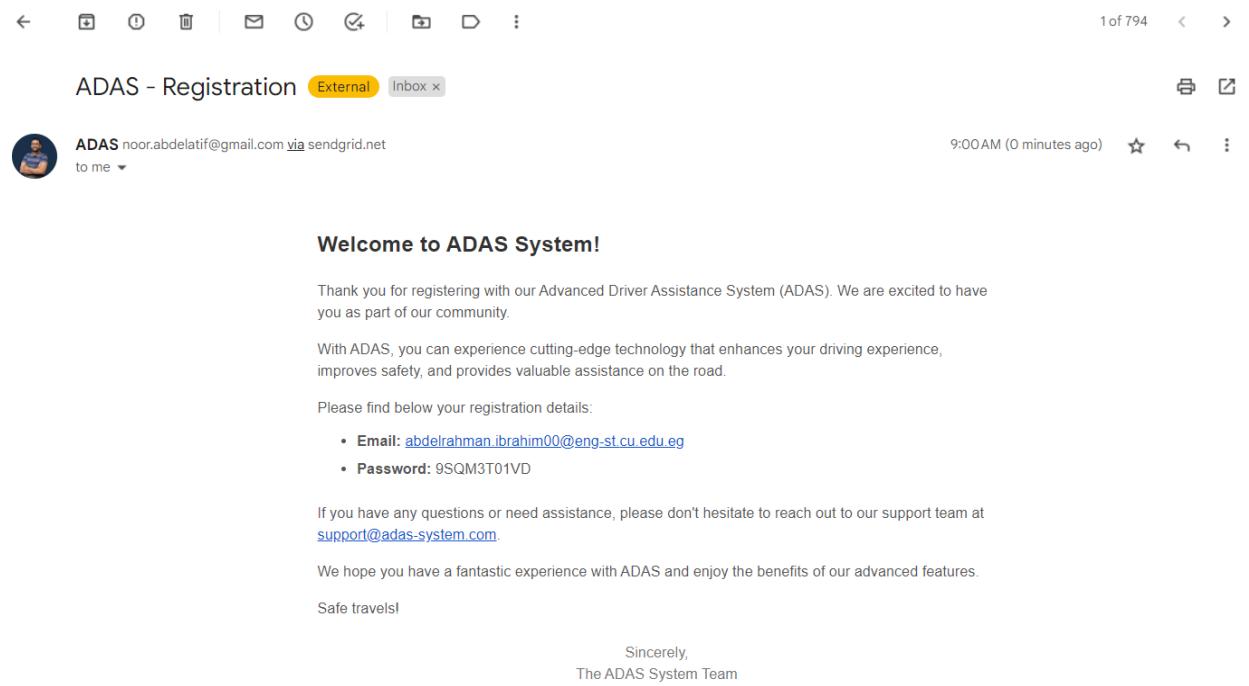

Figure 109 the received email when registration is complete

7.7.1.3 Invalid email format

If an employee attempts to register an account with an invalid email format, the system will check the server to confirm the email's validity. If the email format is invalid, the system will display a popup message to notify the user of the error.

This step is essential for maintaining the integrity of the company's data by ensuring that email addresses are in the correct format, the system can minimize the risk of errors and ensure that only authorized personnel have access to the car's options.

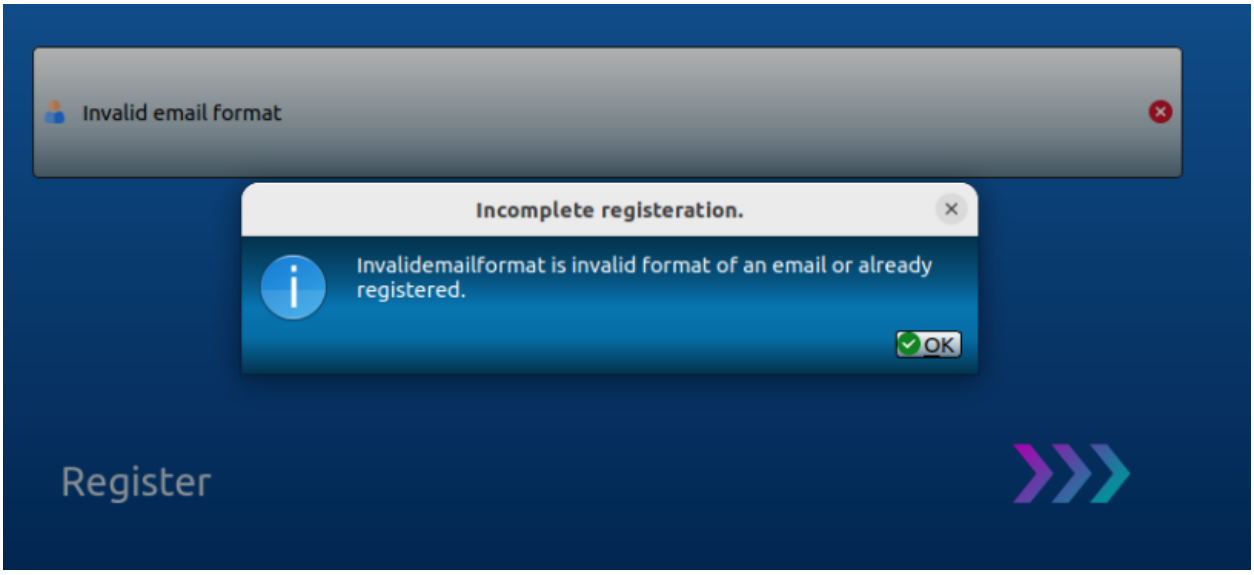

Figure 110 popup to indicate that this is not an email

7.7.2 Employee’s password

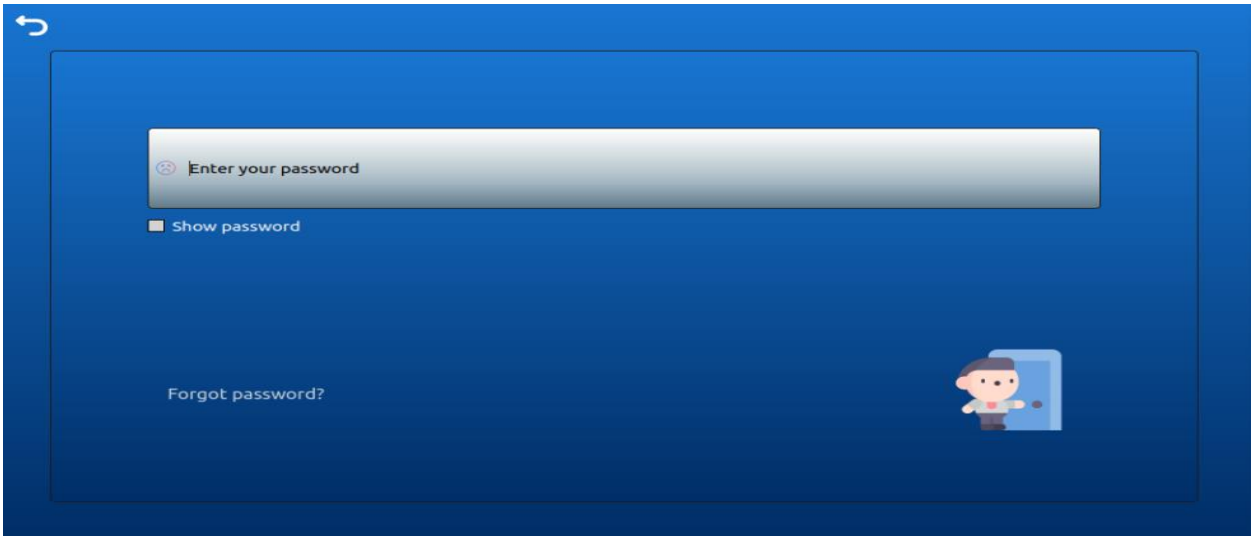

Figure 111 personnel password verification

7.7.2.1 Password confirmation

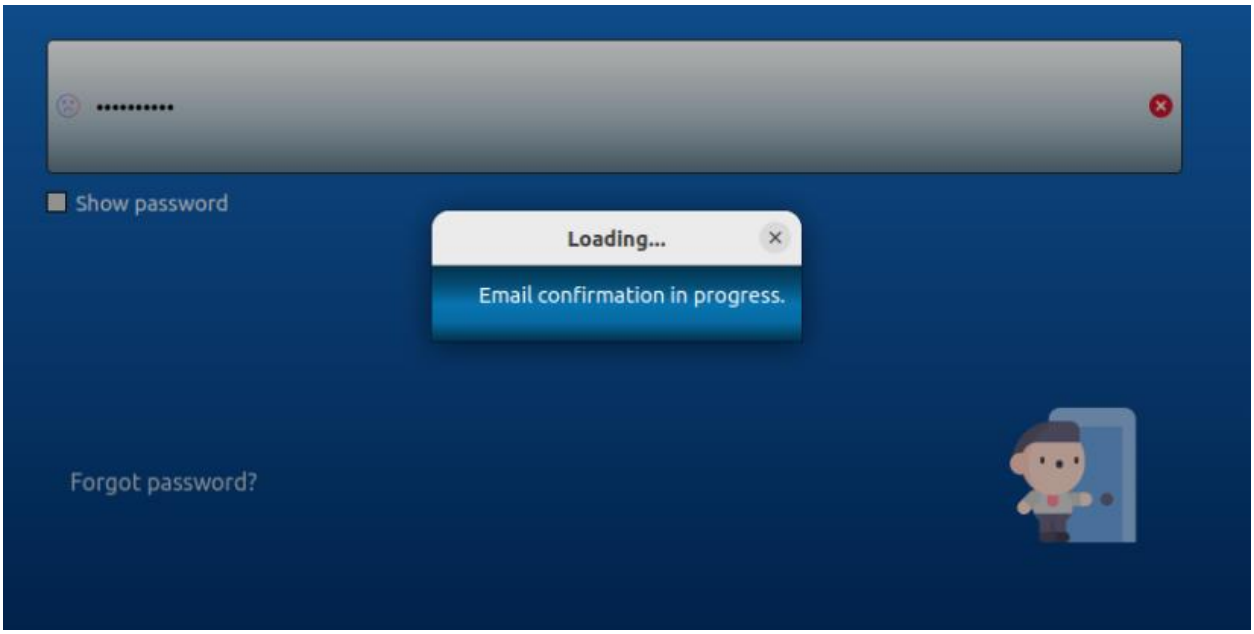

Figure 112 popup to make email confirmation interactive with our server

When the employee types his password in the widget to confirm his registration, the password appears as stars for security reasons. This ensures that the password remains confidential and cannot be viewed by anyone else while it is being entered. Once the employee submits his password, the system initiates a check of the server’s database to ensure that the password matches the email provided.

During the database check process, a loading popup message appears to inform the user that their request is being processed. This message is designed to keep the user informed of the progress of their request and helps to prevent confusion or frustration.

Once the database check is complete and the server's reply is received, the system verifies whether the password matches the email provided or not. If the password does match, the system will confirm the user's access and redirect them to the agency widget.

If the password does not match, a popup message will appear, informing the user that the password is incorrect as shown below. The message will also provide information on the remaining number of attempts they have to enter the correct password before the account is being blocked. This helps to prevent unauthorized access by someone who does not have permission to access the confidential car features.

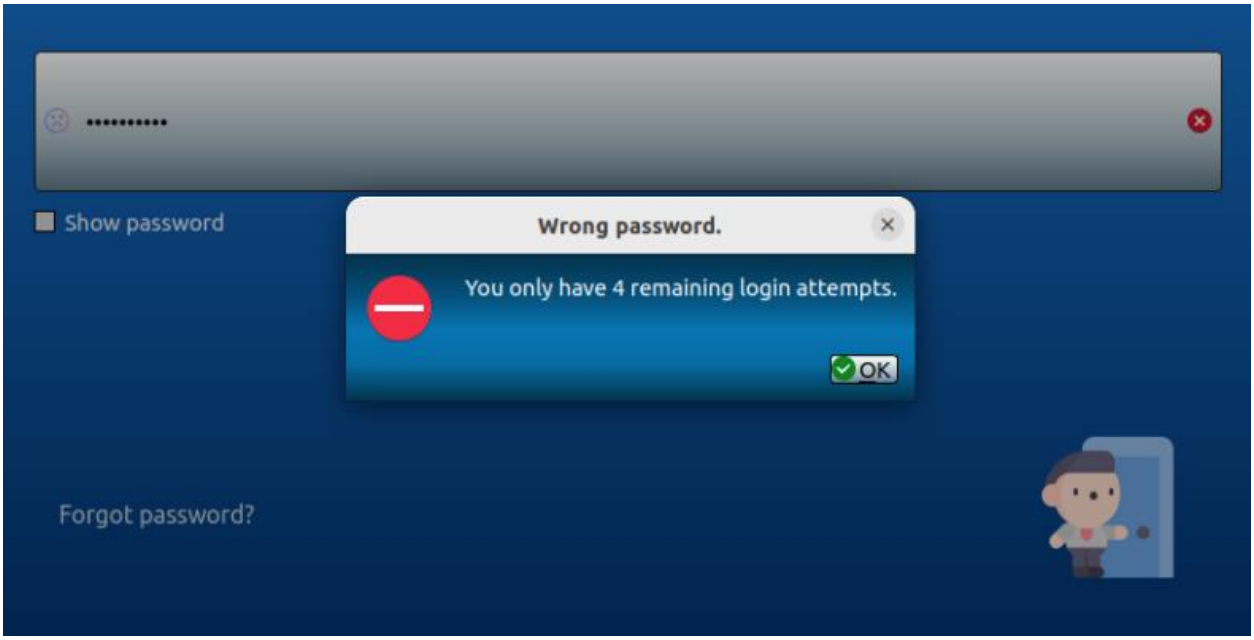

Figure 113 popup to display password remaining trials

There are three scenarios that could arise when an employee enters an incorrect password. The first scenario is that the employee has made a typo while typing his password. The second scenario is that he has forgotten his password and needs to reset it. The final scenario is that someone who does not have permission to access the confidential car features is attempting to log in. By providing information about the remaining number of attempts, the system helps to prevent unauthorized access and ensures that only authorized personnel can access the confidential car features.

7.7.2.2 Wrong password procedures

Under the first scenario, if the employee is not sure about his password typing, there is a checkbox available that allows them to view their password in plain text, instead of the private star state. Additionally, the line edit for the password field includes an emoji on the left side which is a one who cannot see while the password is in private mode. However, when the user checks the "show

password" checkbox, the emoji changes to a happy face emoji that can see. To address this scenario, the system will provide the employee with the remaining number of attempts for each trial to enter the correct password.

Figure 114 show password checkbox

If the password entered is still being incorrect after the employee views it in plain text, he should go to the second scenario. This scenario involves the possibility that the employee has forgotten his current password. In such a case, the system includes a "forgot password" button in the GUI that the employee can click to reset his password.

When the employee clicks the "forgot password" button, the system automatically changes his account password and sends an email to the employee with the new password. This email provides the employee with instructions to change their password when they guarantee access to the widget.

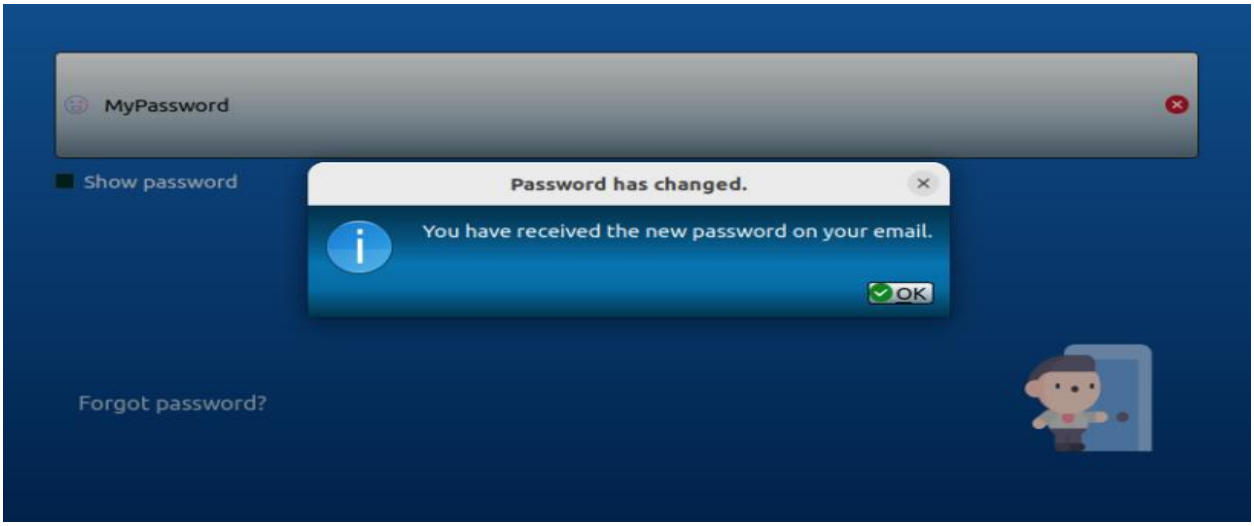

Figure 115 popup to notify the employee to check his email

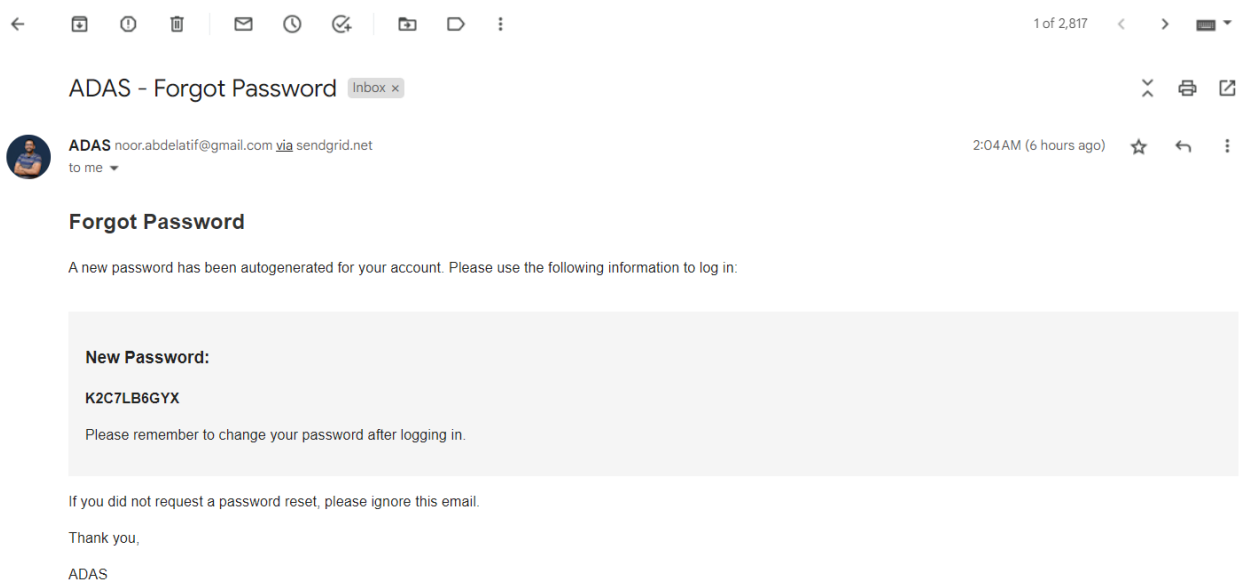

Figure 116 received email by the personnel

7.7.2.3 Banned account

If the employee exhausts all their attempts without entering the correct password, the system will assume the third scenario which's someone who is not authorized is attempting to access the confidential car features. In such a case, the system will block further attempts to log in and will inform the relevant authorities to investigate the incident. This ensures that only authorized personnel are able to access the confidential car features and helps to maintain the security and integrity of the system.

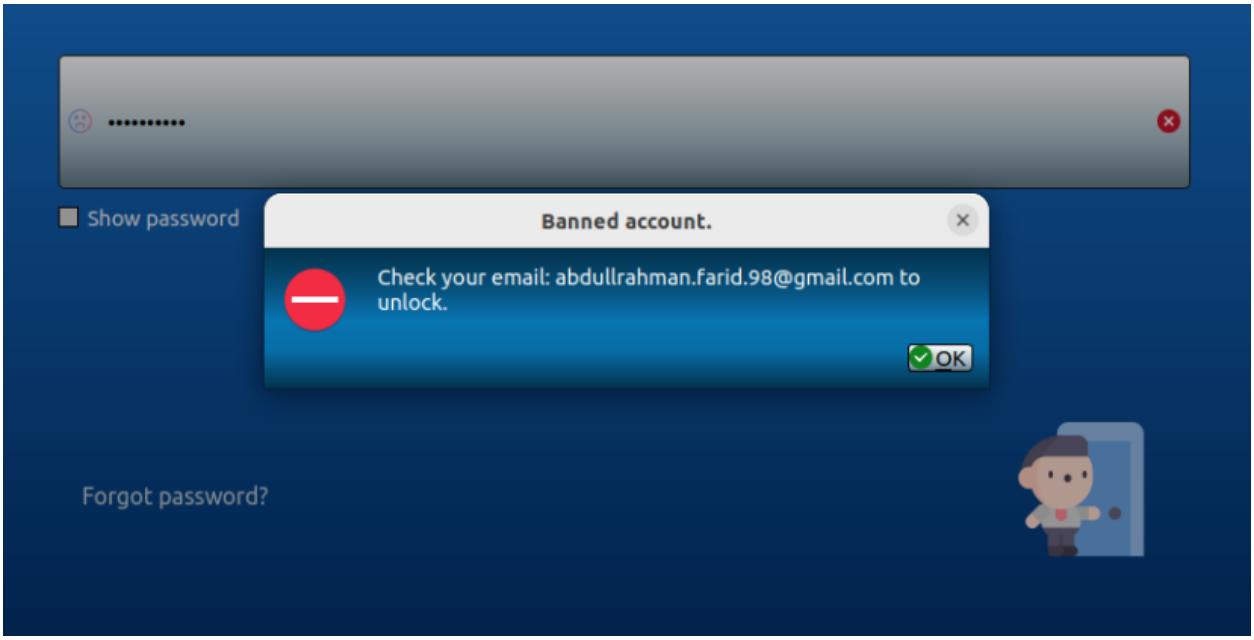

Figure 117 popup that blocks the account due to multiple wrong passwords

If the employee attempts to re-login using the email that was banned due to multiple login attempts with incorrect passwords from the initial point of the login as shown in **figure 118**, which's clicking the agency button, they will find that the email is still banned. This ensures that the system remains secure and that unauthorized persons cannot access the confidential car features.

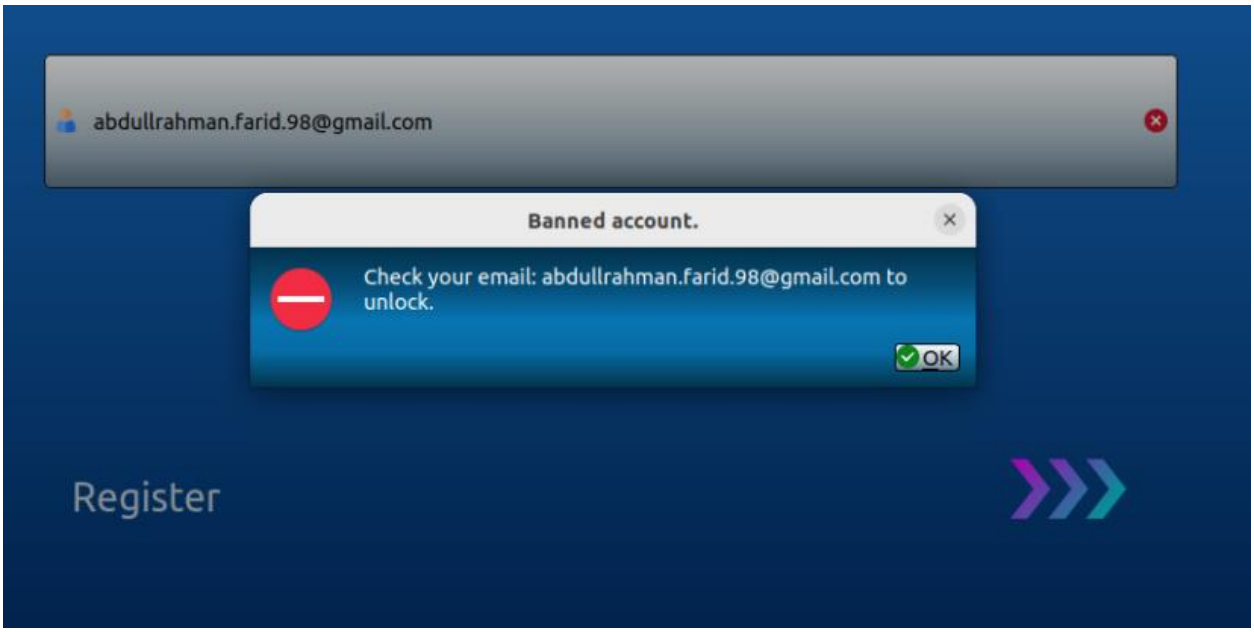

Figure 118 popup that appears if the banned account tried to re-login

When an email address is banned, the account will be locked and the password will be changed automatically for security reasons. This ensures that the system remains secure and that unauthorized personnel cannot access the confidential car features. To re-activate the account, the system will send an email to the employee with an activation link.

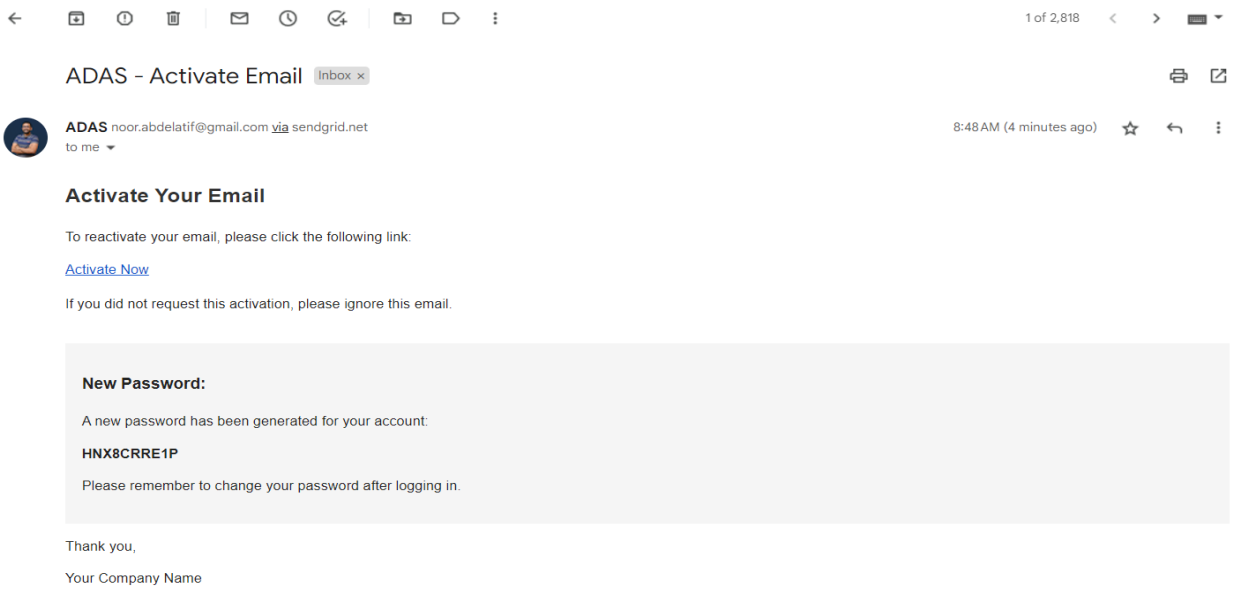

Figure 119 re-activate account email

The activation link is designed to be secure and efficient, ensuring that only the employee with the banned email address can re-activate their account. By providing clear instructions and a secure activation link, the system helps to ensure that any issues with accessing the confidential car features are addressed promptly and efficiently, while also maintaining the security and integrity of the system.

In conclusion, the system's account lockout and activation features are essential components of its security infrastructure, providing a helpful solution to employees who have been banned due to multiple login attempts with incorrect passwords while also maintaining the security and integrity of the confidential car features.

7.8 Agency widget

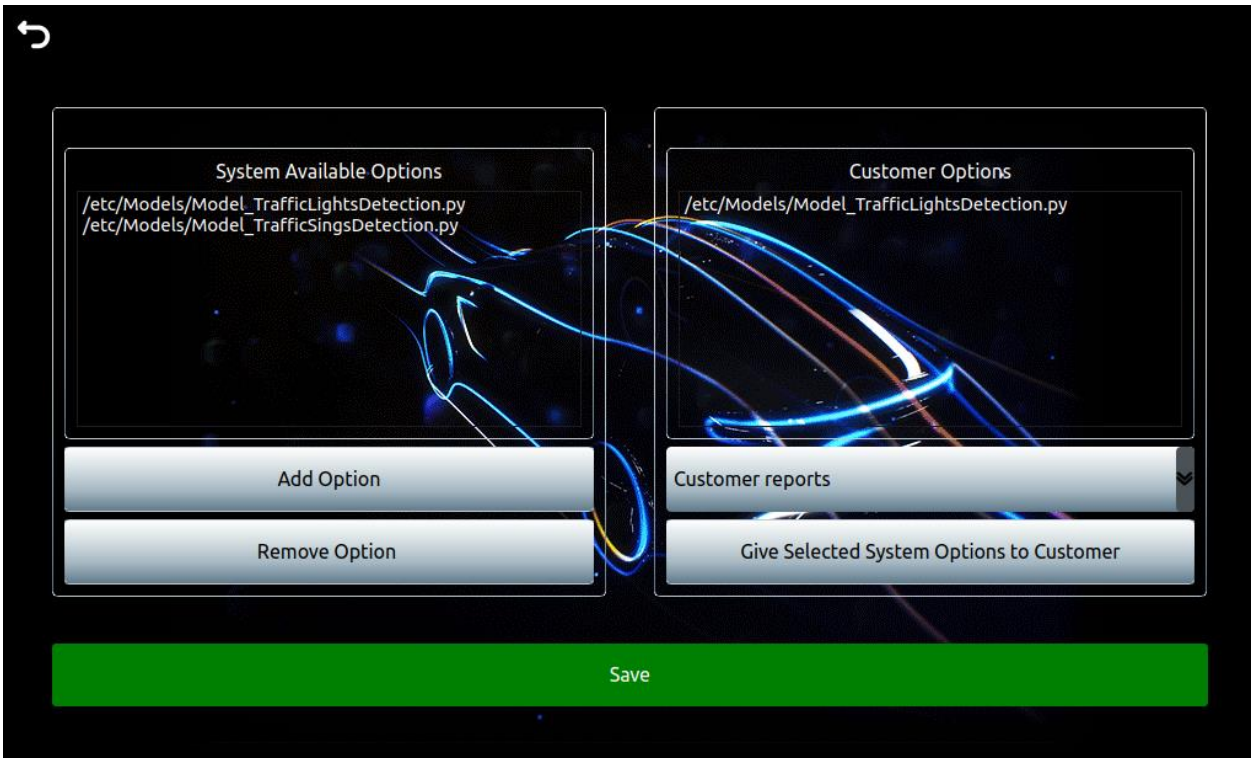

Figure 120 widget of the programmer mode

Once the system has verified that the person who is trying to log in has guaranteed access, the agency widget will be displayed. This widget is not designed to be visually satisfying, as it is intended to be used primarily by programmers who need access to confidential car features.

Instead, the agency widget is designed to be professional and efficient, providing programmers with the tools and information they need to perform their jobs effectively. As such, it may be less visually appealing than other widgets in the system.

Despite its professional appearance, the agency widget does include one promising and new feature: a dynamic background image of a car. This dynamic behavior adds a touch of visual interest to the widget without compromising its professional appearance. By incorporating

dynamic elements into the design of the agency widget, the system helps to maintain the interest and engagement of programmers who use it regularly.

When accessing the agency widget, programmers will see two list widgets displayed on the screen as shown in **figure 120**. The list widget on the left-hand side displays the features that the agency is adding to the car. However, access to these features is restricted by the programmer, and the car owner has the right to use only the features in the right hand side list widget. On the right-hand side, programmers will see a list widget that displays the features that the car owner has access to. These features have been approved by the agency and have been deemed safe and appropriate for use by the car owner, in addition to they are provided to his car category, or he has purchased them.

7.8.1 Add/Remove Option button

To add features to the left-hand side list widget on the agency widget, programmers can simply click on the "Add" option button. This will allow them to browse through a pre-configured location on their host that contains all the features available through the agency **as shown below**.

Once the programmer has located the desired feature, they can add it to the left-hand side list widget. However, it is important to note that access to these features will not be granted until the car owner has guaranteed access to them, as mentioned previously.

By providing a pre-configured location for available features, the system ensures that programmers have easy access to the features they need to add to the car. This helps to streamline the process of adding new features while also maintaining the security and integrity of the system.

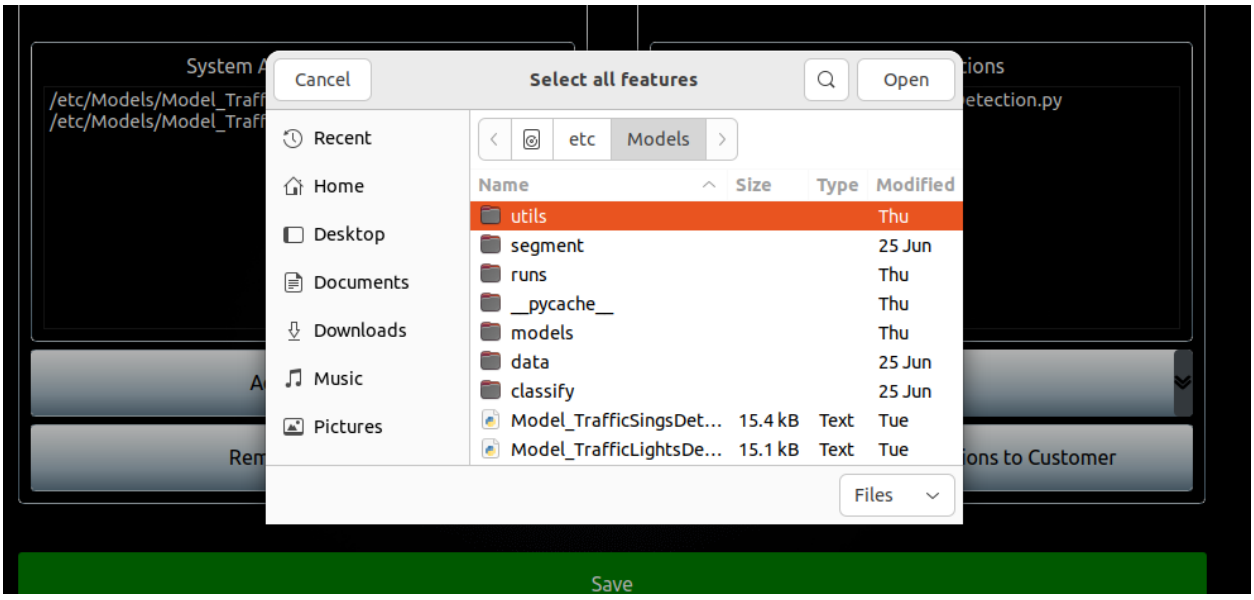

Figure 121 browse location of the "Add Option" button

If he wants to remove any options from the left hand side widget, he just needs to select all of them and press on the "Remove Option" button.

7.8.2 Give Selected System Options to Customer button

To guarantee access to features for the car owner, programmers can select all the desired features to be in the right-hand side list widget. Once the desired features have been selected, he can click on the "Give Selected System Options to Customer" button as shown in **figure 122**.

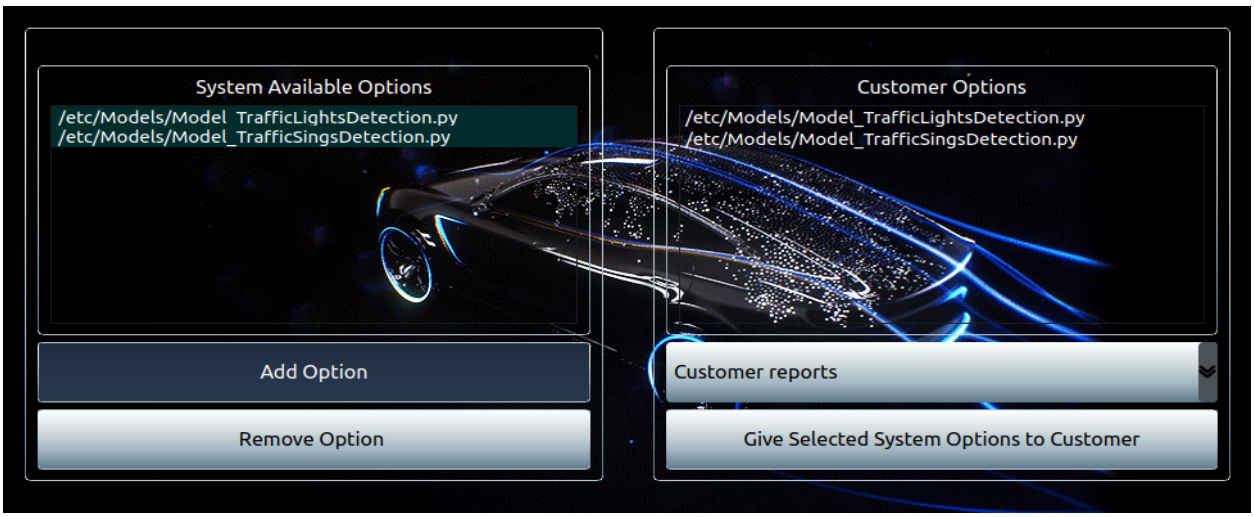

Figure 122 guarantee access to customer to the highlighted options from the left list

This action will grant the car owner access to the selected features, allowing them to use them safely and effectively. By providing a clear and efficient method for granting access to features, the system helps to ensure that the car owner has access to the tools they need to operate the car safely and efficiently.

If a programmer needs to remove access to features for the car owner, they can follow a similar process to granting access. By selecting the features, they want to remove access to in the right-hand side list widget on the agency widget, programmers can then click on the "Give Selected System Options to Customer" button to remove access to those features as shown below.

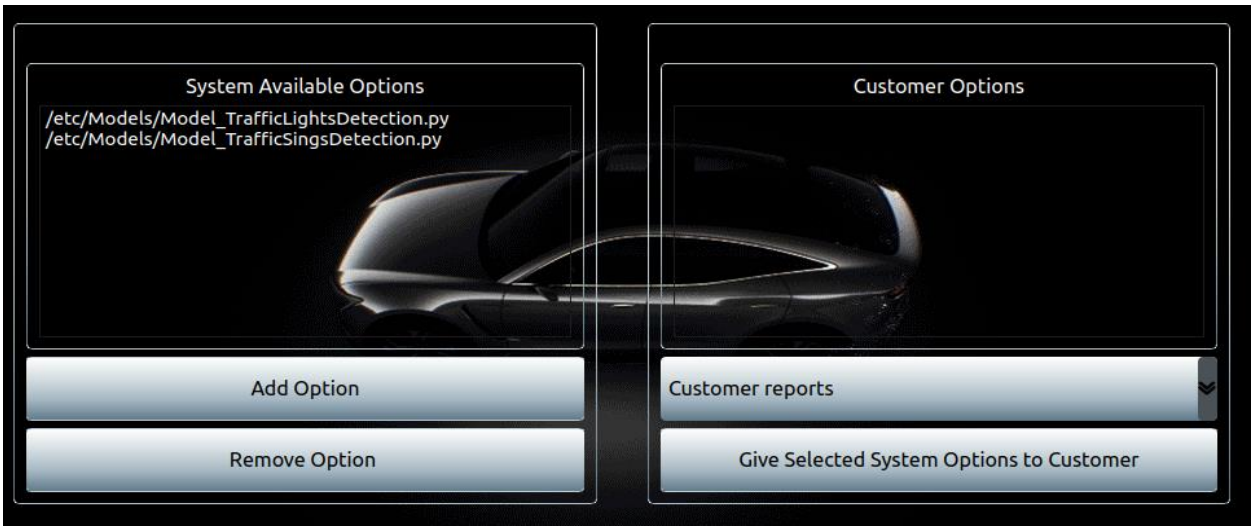

Figure 123 guarantee access to customer to the highlighted options from the left list which’re none

It is important to note that if a programmer accidentally removes access to all features available to the car owner, a popup message will appear when they click on the "Save" button. This message will indicate what has happened as shown in **figure 124**, ensuring that the programmer is aware of the changes he made.

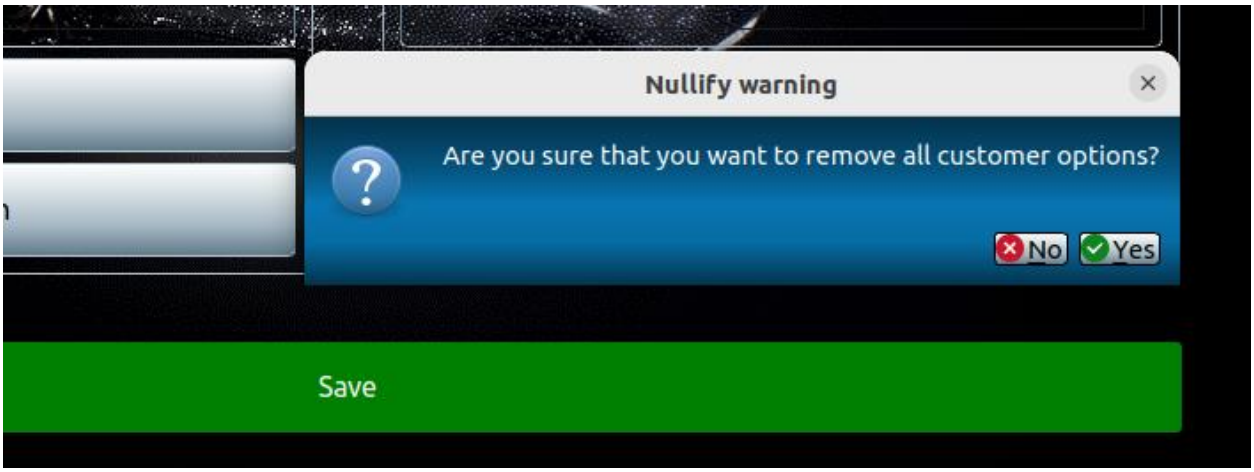

Figure 124 popup that warns the programmer from removing all the features available to the customer

By providing a popup message to alert programmers to potential mistakes, the system helps to prevent accidental removal of access to all features for the car owner. This helps to maintain the security and integrity of the system while also ensuring that the car owner has access to the features they need to operate the car safely and efficiently.

7.8.3 Customer reports dropdown menu

As mentioned in section 7.7.3, the agency widget includes a "Bug Report" button that programmers can use to report any issues or bugs they encounter while accessing confidential car features. These reports are stored in the "Customer reports" drop-down menu, allowing employees to view all unresolved bugs and track their progress as shown in **figure 125**.

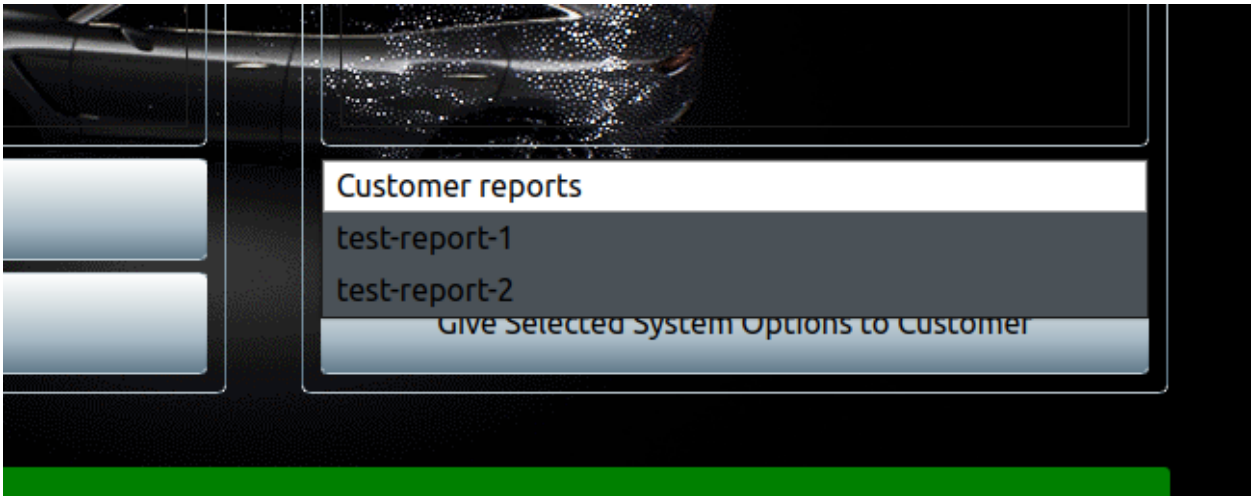

Figure 125 all the un-resolved bugs reported by the customer

By providing a centralized location for bug reports, the system ensures that programmers can quickly and efficiently report any issues they encounter while accessing confidential car features. This helps to ensure that any bugs are addressed promptly, minimizing any potential impact on the system's security and integrity.

7.9 General details

7.9.1 Interactive popups between the GUI and our server

This feature offers several benefits. Firstly, it ensures that the user is aware that the request is being sent and that there may be a brief delay before the report is successfully transmitted. This helps to prevent any confusion or frustration that the user may experience when clicking on any button related to our server.

7.9.2 Discard the line edit spaces and end lines

```
/* Read the email in the life edit and remove any spaces and new lines */
auto email = ui->idLineEdit->text().remove(" ").remove("\n");
```

Figure 126 C++ code that removes spaces and end lines

It is not uncommon for users to accidentally add a space at the beginning or end of their email or password when logging in. This can result in a submission problem, as the system may not recognize the email or password with the added space.

This issue is often caused by copying and pasting the email or password, which can sometimes bring a new line with it, represented by "\n". To avoid this issue, users should ensure that they remove any spaces or new lines from their email or password before submitting their login information.

By removing any spaces or new lines, users can ensure that they have a better login experience, with fewer submission problems and faster access to confidential car features. Overall, paying attention to these small details can help to streamline the login process and improve the overall security and integrity of the system.

7.9.3 Internet connection popups

As illustrated in most of the previous sections, our system features an interactive GUI with our web server. However, one potential issue that can arise is what happens if the internet connection is lost while using the GUI.

To prevent frustration for users, the GUI has been designed to display a loading prompt while attempting to reach the web server. However, if the connection is lost for an extended period of time, the GUI may become stuck at the loading prompt. This can be frustrating for users, as clicking buttons or links will have no effect.

To avoid this problem, the system has been designed to display a popup message to the user if it is unable to reach the web server due to internet connection issue, not just because of another issue

facing it to hit the end point. This message will indicate that there is a problem with the internet connection, allowing the user to take appropriate action, such as checking their internet connection or trying again later.

7.9.4 Qmake

Qmake is a build system tool that is used to generate Makefiles for building applications. It is part of the Qt toolkit, which is a popular framework for developing cross-platform applications. One of the key advantages of qmake is that it is highly configurable, allowing developers to customize the build process to suit their needs.

In a qmake file, developers can specify a wide range of options to control how their application is built. For example, they can specify the location of header files, source files, and resources, as well as any external libraries that the application depends on. They can also set various compiler and linker options, such as optimization flags, debug symbols, and library search paths.

Qmake uses a set of predefined variables and functions to help developers manage their project's dependencies. For example, the "SOURCES" variable can be used to specify a list of source files, while the "HEADERS" variable can be used to specify a list of header files. Qmake also provides a number of built-in functions for handling more complex tasks, such as generating moc files for Qt's meta-object system or processing resource files.

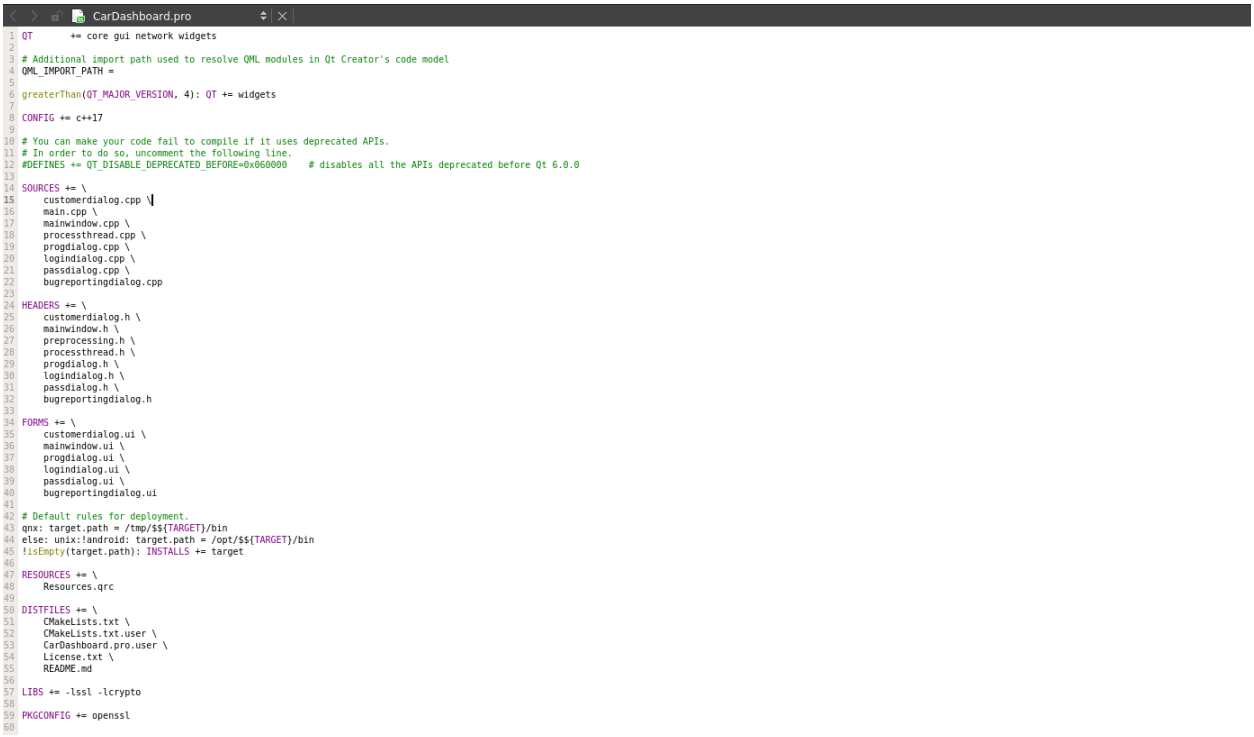

Figure 127 our qmake

One of the key benefits of using qmake is that it automates many of the common tasks involved in building a GUI application. For example, it can automatically generate the necessary Makefiles

and build scripts based on the dependencies specified in the qmake file. This can save developers a significant amount of time and effort compared to manually managing the build process.

To build our application using qmake, we first need to ensure that we have a .pro file in our project directory. This file contains the necessary information about the project's dependencies and build process, and is used by qmake to generate the Makefiles that are needed to build the application.

Once we have a .pro file in our project directory, we can build the application using the following steps. First, we need to navigate to the directory that contains the .pro file using the terminal. We can do this by using the "cd" command to change to the appropriate directory.

Next, we need to create a build directory within the project directory. This directory will contain the generated Makefiles and object files that are needed to build the application. We can create the build directory using the "mkdir" command.

After creating the build directory, we need to navigate to it using the terminal. We can do this by using the "cd" command again, this time to change to the build directory that we just created.

Once we are in the build directory, we can use the "qmake" command to generate the Makefiles for the project. To do this, we need to specify the location of the .pro file in the previous directory by typing "qmake ../" in the terminal. This tells qmake to look for the .pro file in the parent directory and generate the necessary Makefiles in the current build directory.

With the Makefiles generated, we can now use the "make" command to build the application. This will compile the source code and generate the executable file that we can run on our system.

```
hussam@hussam-VirtualBox:~/ADAS/Integrated-GUI/build$ qmake ../ && ls
Makefile
```

Figure 128 building our application using our qmake

After we have generated the Makefile for our application using qmake, we can use the "make" command to build the application. The Makefile is a file that automates the build process based on the dependencies and settings specified in our qmake file. It contains a set of rules and commands that tell the compiler how to build the application.

To build the application using the Makefile, we simply need to navigate to the build directory and type "make" in the terminal. This tells the Makefile to execute the necessary commands to build the application. We can see the commands being executed in the terminal as the build process progresses.

The Makefile typically contains a set of rules that define how to build the different components of the application. For example, there may be rules for compiling the source code, linking the object files, and generating the executable file. The Makefile also includes information about the dependencies between different components of the application, which allows it to automatically rebuild the necessary components if any of the dependencies change.

As the build process progresses, we can see the commands being executed in the terminal. These commands typically include compiler and linker commands, as well as other tools that may be

required to build the application. The output of these commands is displayed in the terminal, allowing us to monitor the build process and identify any errors or warnings that may occur.

Once the build process is complete, we should have a fully functional executable file that we can run on our system. If any errors or warnings were encountered during the build process, we can use the information displayed in the terminal to troubleshoot and resolve the issue.

```
hussam@hussam-VirtualBox:~/ADAS/Integrated-GUI/build$ make
/usr/lib/qt5/bin/uic ../customerdialog.ui -o ui_customerdialog.h
/usr/lib/qt5/bin/uic ../mainwindow.ui -o ui_mainwindow.h
/usr/lib/qt5/bin/uic ../progdialog.ui -o ui_progdialog.h
/usr/lib/qt5/bin/uic ../logindialog.ui -o ui_logindialog.h
/usr/lib/qt5/bin/uic ../passdialog.ui -o ui_passdialog.h
/usr/lib/qt5/bin/uic ../bugreportingdialog.ui -o ui_bugreportingdialog.h
g++ -c -pipe -O2 -std=gnu++1z -Wall -Wextra -D_REENTRANT -fPIC -DQT_NO_DEBUG -DQ
T_WIDGETS_LIB -DQT_GUI_LIB -DQT_NETWORK_LIB -DQT_CORE_LIB -I../../Integrated-GUI
 -I. -I/usr/include/x86_64-linux-gnu/qt5 -I/usr/include/x86_64-linux-gnu/qt5/QtW
idgets -I/usr/include/x86_64-linux-gnu/qt5/QtGui -I/usr/include/x86_64-linux-gnu
/qt5/QtNetwork -I/usr/include/x86_64-linux-gnu/qt5/QtCore -I. -I. -I/usr/lib/x86
_64-linux-gnu/qt5/mkspecs/linux-g++ -o customerdialog.o ../customerdialog.cpp
g++ -c -pipe -O2 -std=gnu++1z -Wall -Wextra -D_REENTRANT -fPIC -DQT_NO_DEBUG -DQ
T_WIDGETS_LIB -DQT_GUI_LIB -DQT_NETWORK_LIB -DQT_CORE_LIB -I../../Integrated-GUI
 -I. -I/usr/include/x86_64-linux-gnu/qt5 -I/usr/include/x86_64-linux-gnu/qt5/QtW
idgets -I/usr/include/x86_64-linux-gnu/qt5/QtGui -I/usr/include/x86_64-linux-gnu
/qt5/QtNetwork -I/usr/include/x86_64-linux-gnu/qt5/QtCore -I. -I. -I/usr/lib/x86
_64-linux-gnu/qt5/mkspecs/linux-g++ -o main.o ../main.cpp
```

Figure 129 make command example

As shown in **figure.129**, the first line of the Makefile that is generated by qmake typically contains a set of commands that are used to generate the necessary object files and header files from the UI files in our project. This is necessary because the UI files are created using Qt Designer and need to be converted into a format that can be used by our application code.

The first command in this set typically specifies the path to the uic runner, which is a tool that is provided by the Qt framework for converting UI files into C++ code. The path to the uic runner is typically specified as "/usr/lib/qt5/bin/uic", although this may vary depending on the installation of the Qt framework on your system.

The next part of the command specifies the location of the UI file that we want to convert. In this case, we are using the "../customerdialog.ui" file, which is located in the parent directory of the current directory. This specifies the source file that we want to convert into C++ code.

The "-o" option is used to specify the name of the output file that we want to generate. In this case, we are specifying "ui_customerdialog.h" as the name of the output file. This file will contain the generated C++ code that is based on the contents of the customerdialog.ui file.

Once we have generated the necessary object files and header files using the Makefile and uic runner, we can use the "make" command to build the executable for our application. This process will compile all of the source code files, link them together, and generate an executable that we can run on our system.

As we can see in **figure 130**, the Makefile has successfully generated the necessary object files and binaries for all of the .cpp files in our project. Additionally, the binaries for the .ui files have also been generated, allowing us to use the user interface components in our application.

With all of the necessary binaries generated, we can use the "make" command to build the final executable for our application. This executable will be named according to the settings specified in our qmake file, and will be located in the build directory of our project.

```
hussam@hussam-VirtualBox:~/ADAS/Integrated-GUI/build$ ls
bugreportingdialog.o          moc_logindialog.cpp     passdialog.o
CarDashboard                  moc_logindialog.o       processthread.o
customerdialog.o              moc_mainwindow.cpp      progdialog.o
logindialog.o                 moc_mainwindow.o        qrc_Resources.cpp
main.o                        moc_passdialog.cpp      qrc_Resources.o
mainwindow.o                  moc_passdialog.o        ui_bugreportingdialog.h
Makefile                      moc_predefs.h           ui_customerdialog.h
moc_bugreportingdialog.cpp    moc_processthread.cpp   ui_logindialog.h
moc_bugreportingdialog.o      moc_processthread.o     ui_mainwindow.h
moc_customerdialog.cpp        moc_progdialog.cpp      ui_passdialog.h
moc_customerdialog.o          moc_progdialog.o        ui_progdialog.h
```

Figure 130 all of the qmake generated binaries in addition to the C++ executable

Once the executable has been built, we can run it using the command shown in **figure 131**. This will launch the application and allow us to interact with the user interface and perform any necessary tasks.

```
hussam@hussam-VirtualBox:~/ADAS/Integrated-GUI/build$ ./CarDashboard
```

Figure 131 C++ executable running command

7.9.5 Resources to binaries

As our application grows in complexity, it is common to add images and other resources such as CSS files to our project. However, when we add these resources using absolute paths, this can create problems when the application is moved to a different host or directory. In this case, the paths to the resources may no longer be valid, and we would need to modify the code to reflect the new paths.

To avoid this problem, we can use a technique called resource embedding. This involves adding all of the necessary resources to the binaries of the Qt framework, so that they are included as part of the application itself. This ensures that the resources are always available, regardless of the location of the application on the file system.

By embedding our resources in this way, we can ensure that our application is fully configurable and portable. We no longer need to worry about the paths to our resources, as they are included as

part of the application itself. This makes it easy to move the application to a different host or directory without needing to modify the code.

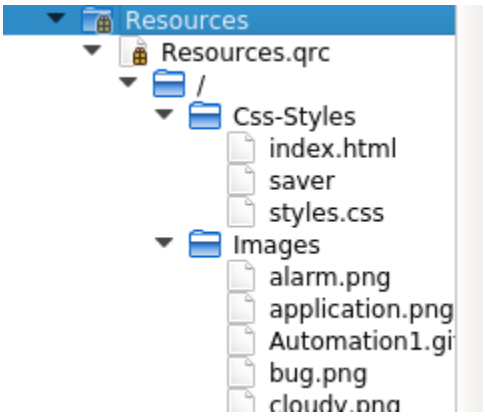

Figure 132 resources of the application

To include our resources in the application binaries using resource embedding, we first need to specify them in our qmake file. This is typically done using the Resource Collection File (.qrc) format, which allows us to specify a list of resources that should be included in the application.

As shown in figure 127, we can add our resource file to the "resources" variable in our qmake file. This specifies the name of the .qrc file that contains our resources.

The .qrc file itself contains a list of resources, each of which is specified using an absolute path that starts with ":/" as shown in **figure 133**. This path specifies the default location of the resources on the host of the GUI application, and allows us to navigate to the resources from this point using a relative path.

For example, if we have an image file located in a subdirectory of our project called "images", we can include it in our .qrc file using the following syntax: “:/images/myimage.png</file>”

This specifies the absolute path to the image file, starting from the default location of the resources on the host. We can then use this path to reference the image in our application code, without needing to worry about the location of the image file on the file system.

```
60 void MainWindow::SetTempAndHumidityIcons()
61 {
62     int tempReading = ui->temperature_label->text().toInt();
63     QPixmap pixmap;
64     if(tempReading < 15)
65     {
66         pixmap.load(":/Images/snow.png");
67     }
68     else if(tempReading < 25)
69     {
70         pixmap.load(":/Images/cloudy.png");
71
72     }
73     else
74     {
75         pixmap.load(":/Images/sun.png");
76     }
77     QIcon icon(pixmap);
78     ui->tempImg->setIcon(icon);
79 }
```

Figure 133 example function to show how the absolute path is specified

7.9.6 Memory management

One of the main reasons for choosing C++ over Python for GUI application development is the ability to have full access to memory. This is particularly important when dealing with dynamically created widgets, which are typically created and destroyed during the lifetime of the application.

In our C++, we created each widget dynamically before it is displayed, meaning that its destructor is called after it is closed or destroyed. This ensures that memory is properly freed up and prevents memory leaks from occurring.

Furthermore, in order to avoid wasting memory, many of the resources and objects in the application are created dynamically, using the "new" keyword. This allows us to allocate and deallocate memory as needed, rather than pre-allocating a fixed amount of memory that may not be fully utilized.

In addition to dynamic memory management, C++ also provides the ability to declare variables using the "auto" keyword. This allows us to specify the type of a variable implicitly, based on its initialization value, while still ensuring that the correct amount of memory is allocated.

7.10 Hosting .NET7 Web APIs on Azure

The purpose of this part is to provide a comprehensive guide on hosting a .NET 7 Web API server on Azure while implementing Clean Architecture and CQRS (Command Query Responsibility Segregation) principles. This part aims to help developers and software engineers understand the necessary steps, best practices, and considerations involved in designing, developing, and deploying a robust and scalable Web API server using these architectural and design patterns.

By following this part, you will gain insights into the principles and benefits of Clean Architecture and CQRS, and how you can be effectively applied to a .NET 7 Web API server. You will learn how to set up their development environment, design the architecture following Clean Architecture principles, implement CQRS using the MediatR library, develop the Web API server, test the solution, prepare the Azure environment, and deploy the application to Azure App Service. Additionally, this part will cover topics such as monitoring, scaling, and managing the deployed application on Azure.

Ultimately, this part aims to equip developers with the knowledge and practical guidance needed to build highly maintainable, scalable, and loosely coupled Web API servers on Azure, leveraging the power of Clean Architecture and CQRS. By adopting these architectural and design patterns, developers can improve the modularity, testability, and maintainability of their codebase, enabling them to build robust and scalable applications that can easily evolve over time.

7.10.1 .NET7 Web APIs

7.10.1.1 Introduction

.NET 7 is the latest release of the popular and powerful open-source framework developed by Microsoft. Building upon the success of its predecessors, .NET 7 introduces several new features and improvements that enhance the development experience for building robust and scalable applications. Whether you are a seasoned developer or just starting your programming journey, .NET 7 provides a comprehensive platform for creating a wide range of applications, including web APIs, desktop applications, mobile apps, and more.

One of the key highlights of .NET 7 is its focus on performance and productivity. Microsoft has invested significant efforts in optimizing the framework, resulting in faster execution times and reduced memory usage. These improvements enable developers to build high-performance applications that can handle demanding workloads with ease.

In addition to performance enhancements, .NET 7 brings new language features and APIs that simplify development tasks and make code more concise and expressive. It introduces support for the latest C# language features, allowing developers to leverage the full power of modern programming techniques. Furthermore, .NET 7 incorporates various improvements to existing libraries, frameworks, and tools, enhancing the overall development experience.

Another significant aspect of .NET 7 is its cross-platform capabilities. With .NET 7, you can build applications that run on multiple operating systems, including Windows, macOS, and Linux. This cross-platform support opens up new opportunities for developers to reach a wider audience and deploy applications on diverse environments.

Furthermore, .NET 7 seamlessly integrates with various development tools and services, making it easy to adopt existing workflows and leverage the extensive ecosystem of libraries and frameworks. Whether you prefer using popular integrated development environments (IDEs) like Visual Studio or prefer a command-line interface (CLI) approach with tools like .NET CLI or Visual Studio Code, .NET 7 offers flexibility and compatibility with your preferred development environment.

In this documentation, we will explore the process of hosting a .NET 7 Web API server on Azure while implementing clean architecture and CQRS principles. We will delve into the intricacies of clean architecture, which promotes separation of concerns and maintainability, and CQRS principles, which provide a powerful pattern for handling commands and queries in your application.

By the end of this documentation, you will have a comprehensive understanding of how to leverage the capabilities of .NET 7 to build a robust, scalable, and high-performing Web API server on Azure while adhering to clean architecture and CQRS principles. So let's embark on this journey together and unlock the potential of .NET 7 for your application development needs.

7.10.1.2 Web API

A Web API, short for Application Programming Interface, is a set of rules and protocols that allows different software applications to communicate and interact with each other over the internet. It serves as a bridge between the server-side and client-side components of an application, enabling them to exchange data and perform various operations.

In the context of web development, a Web API is typically a part of a web server that exposes a set of endpoints or URLs that client applications can access. These endpoints represent specific actions or resources that can be requested or manipulated through standard HTTP protocols. The Web API processes incoming requests, performs the necessary actions, and returns the appropriate responses, usually in a structured data format such as JSON or XML.

Web APIs are commonly used to enable interaction between different systems, services, or platforms. They provide a standardized way for applications to access and utilize functionality or data from external sources. For example, a social media platform may expose a Web API that allows developers to integrate features like posting updates, retrieving user information, or interacting with social connections.

One of the fundamental characteristics of Web APIs is their platform independence. They can be consumed by various types of client applications, including web applications, mobile apps, desktop software, or even other APIs. This flexibility enables developers to build scalable and distributed systems that can leverage the capabilities of different technologies and platforms.

Web APIs are designed to follow the principles of Representational State Transfer (REST), which is an architectural style for designing networked applications. RESTful APIs, as they are commonly referred to, use a set of well-defined HTTP methods (such as GET, POST, PUT, DELETE) to perform operations on resources identified by unique URLs. These APIs adhere to a stateless client-server communication model, where each request from the client contains all the necessary information for the server to process it.

7.10.1.3 Benefits of .NET7 Web API

With .NET 7, you can develop Web APIs that can run on different operating systems, including Windows, macOS, and Linux. This cross-platform support allows you to reach a wider audience and deploy your APIs in diverse environments without significant modifications or additional development efforts.

Performance and scalability are key advantages of .NET 7 Web API. The framework introduces performance enhancements and optimizations that make Web APIs built on it highly efficient. The improvements in execution times and reduced memory usage enable your APIs to handle large workloads and scale effectively. Whether you have a small-scale API or a high-demand enterprise-level solution, .NET 7 provides the performance capabilities necessary to meet your scalability requirements.

.NET 7 Web API benefits from a rich development ecosystem as part of the larger .NET ecosystem. This ecosystem provides numerous pre-built components and modules that can accelerate your development process. Additionally, you can leverage existing skills and knowledge

in the .NET community to access support, resources, and community-driven contributions, making development faster and more efficient.

Extensibility and customizability are important aspects of .NET 7 Web API. The framework allows you to tailor your APIs to meet specific requirements. You can easily integrate additional functionalities, third-party libraries, or custom modules into your API, enhancing its capabilities and flexibility. This extensibility enables you to adapt your API to changing business needs and integrate with other systems seamlessly.

.NET 7 provides robust security features and authentication mechanisms for your Web API. You can leverage industry-standard protocols, such as OAuth and JWT, to implement secure authentication and authorization mechanisms. Additionally, .NET 7 integrates well with Azure Active Directory and other identity providers, simplifying the implementation of secure authentication in your Web API.

.NET 7 aims to enhance developer productivity with improved tooling and development experience. It offers support for the latest C# language features and provides a wide range of development tools, including Visual Studio, Visual Studio Code, and .NET CLI. These tools enable developers to write clean, maintainable code and streamline the development process, ultimately boosting productivity.

Testing and debugging capabilities are well-supported in .NET 7 Web API. The framework offers built-in unit testing frameworks, such as NUnit and xUnit, which facilitate the creation and execution of tests to verify the functionality of your APIs. Additionally, .NET 7 integrates with various debugging tools, allowing developers to diagnose and resolve issues efficiently.

.NET 7 Web API seamlessly integrates with Azure cloud services, providing a robust platform for hosting, managing, and scaling your Web API. You can leverage Azure services like Azure App Service, Azure Functions, Azure API Management, and Azure DevOps for deployment, monitoring, scaling, and continuous integration/continuous deployment (CI/CD) processes. This integration streamlines the deployment and management of your Web API, leveraging the power of Azure services.

7.10.1.4 Clean Architecture

Clean Architecture is a software design approach that emphasizes separation of concerns, maintainability, and testability. It provides a set of principles and guidelines for organizing the structure of software systems in a way that promotes flexibility, extensibility, and ease of maintenance. Here are the key principles of Clean Architecture

7.10.1.5 Separation of Concerns

Clean Architecture advocates for the separation of different concerns or responsibilities within a software system. It emphasizes dividing the system into distinct layers, where each layer has a specific purpose and encapsulates a particular set of functionalities. This separation helps to isolate changes and allows for independent development, testing, and modification of each layer without affecting the others.

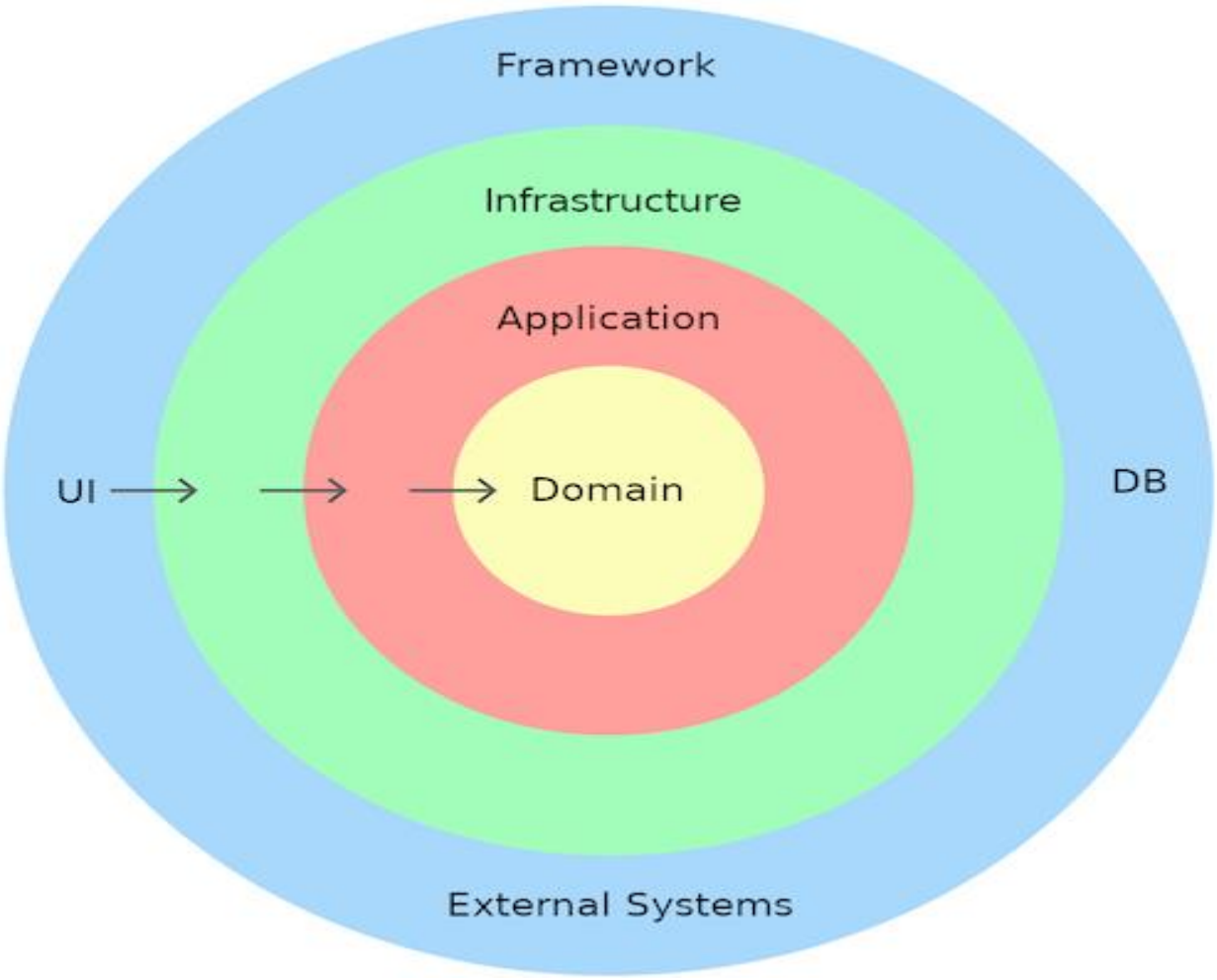

Figure 134 Clean Architecture Example

7.10.1.6 Dependency Rule

The Dependency Rule is a fundamental principle of Clean Architecture. It states that dependencies should always point inward, with higher-level modules depending on lower-level modules. This rule ensures that the innermost layers, containing business logic and core functionality, are independent of the outer layers, such as frameworks, libraries, or external dependencies. This decoupling facilitates easier maintenance, as changes in external dependencies do not ripple through the entire system.

7.10.1.7 Independence of Frameworks

Clean Architecture promotes the idea of keeping the business logic and core functionality of the system independent of any specific frameworks, libraries, or external technologies. By isolating the core logic from the infrastructure or presentation layers, the system becomes more adaptable

to changes in these external factors. This principle enables the application to evolve independently of the chosen frameworks or technologies.

7.10.1.8 Testability

Clean Architecture places a strong emphasis on testability. By separating concerns and isolating dependencies, it becomes easier to write unit tests for individual components and layers. Tests can focus on specific functionalities without the need for complex setups or external dependencies, leading to faster and more reliable testing. This testability aspect helps ensure the stability and quality of the software system over time.

7.10.1.9 Domain-centric Design

Clean Architecture encourages a domain-centric design approach. The core business logic and rules should be at the center of the architecture, with other layers and components built around it. This approach enables a clear understanding and modeling of the domain, leading to a more robust and maintainable system. The domain model should be independent and agnostic of any technical concerns or implementation details.

7.10.1.10 Use Cases or Interactors

Clean Architecture promotes the concept of use cases or interactors. Use cases encapsulate specific actions or operations that the system can perform. They represent the core functionality and business rules of the application. By explicitly defining and separating these use cases, the architecture becomes more modular, readable, and maintainable. Use cases should communicate with the external world through interfaces or ports, allowing for flexibility and potential integration with different user interfaces or external systems.

7.10.1.11 Dependency Injection

Clean Architecture advocates for the use of dependency injection as a means to manage dependencies and decouple components. Dependency injection helps ensure that dependencies are provided externally, allowing for easier testing, modularity, and flexibility. It enables the system to be easily extended or replaced with alternative implementations of components without modifying the core logic.

7.10.1.12 Implementing Clean Architecture

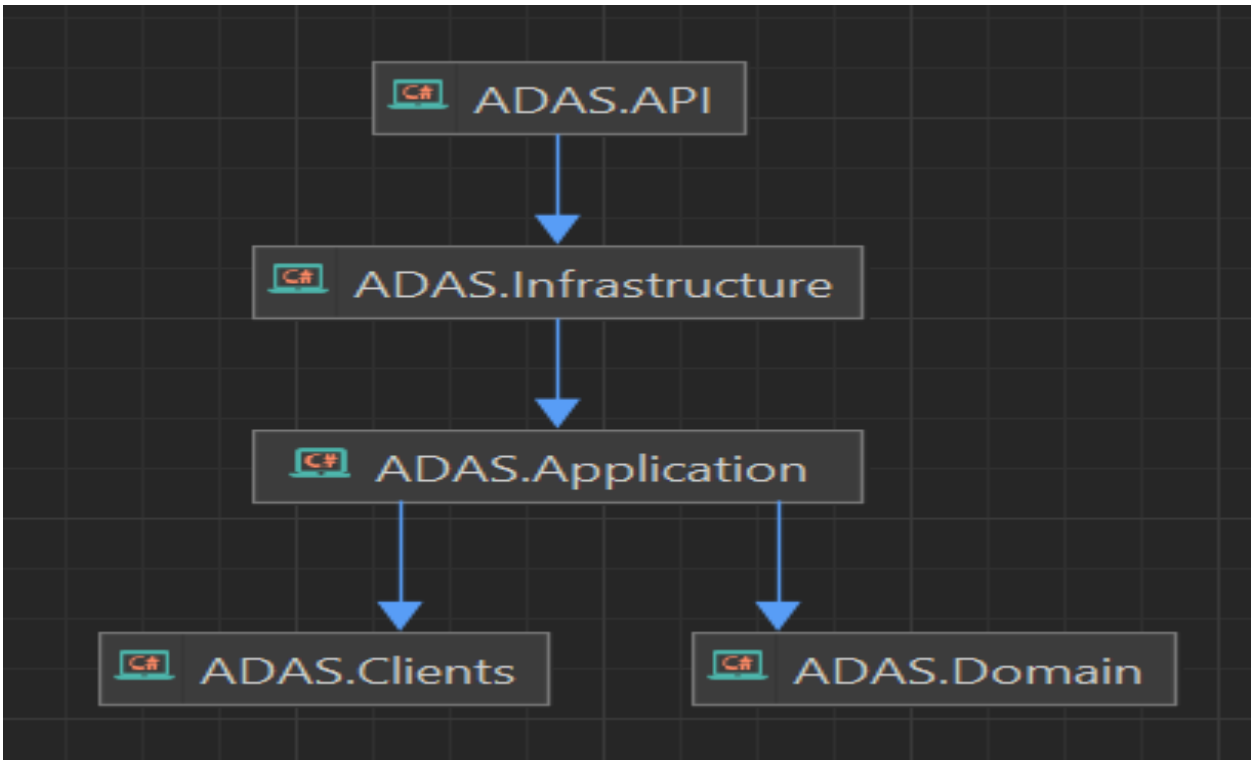

Figure 135 Project Implements Clean architecture principles

In the shown figure, the ADAS.API project serves as the entry point or the presentation layer of the application. It represents the Web API endpoints and handles the incoming HTTP requests. The ADAS.API project references the ADAS.Infrastructure project, which is responsible for handling data access, external services, and other infrastructure-related concerns. This separation ensures that the presentation layer remains decoupled from the specific implementation details of data access and infrastructure.

The ADAS.Infrastructure project, being the lower-level layer, depends on the ADAS.Application project. The ADAS.Application project contains the application layer, which orchestrates the use cases and business logic of the system. It acts as an intermediary between the presentation layer and the domain layer, encapsulating the application-specific behavior and interacting with the domain entities.

Moreover, the ADAS.Application project also references the ADAS.Clients and ADAS.Domain layers. The ADAS.Clients layer represents the client-side components or modules of the application, which can include various client types such as web or mobile. The ADAS.Domain layer holds the core business logic, entities, and rules of the application, which are independent of any specific technology or infrastructure.

This project structure, with the ADAS.API referencing the ADAS.Infrastructure, which in turn references the ADAS.Application, and the ADAS.Application referencing the ADAS.Clients and ADAS.Domain layers, adheres to the principles of Clean Architecture. It enforces separation of

concerns, with each layer having distinct responsibilities and dependencies flowing inward. The application layer acts as an intermediary, ensuring that the business logic remains isolated from the infrastructure and presentation layers.

By following this project structure, you can achieve a modular and maintainable architecture where changes in one layer have minimal impact on other layers. It also allows for easy testing, as dependencies can be mocked or stubbed to focus on testing specific components. This project structure promotes scalability, extensibility, and flexibility in the development and evolution of the ADAS system.

7.10.2 CQRS (Command Query Responsibility Segregation)

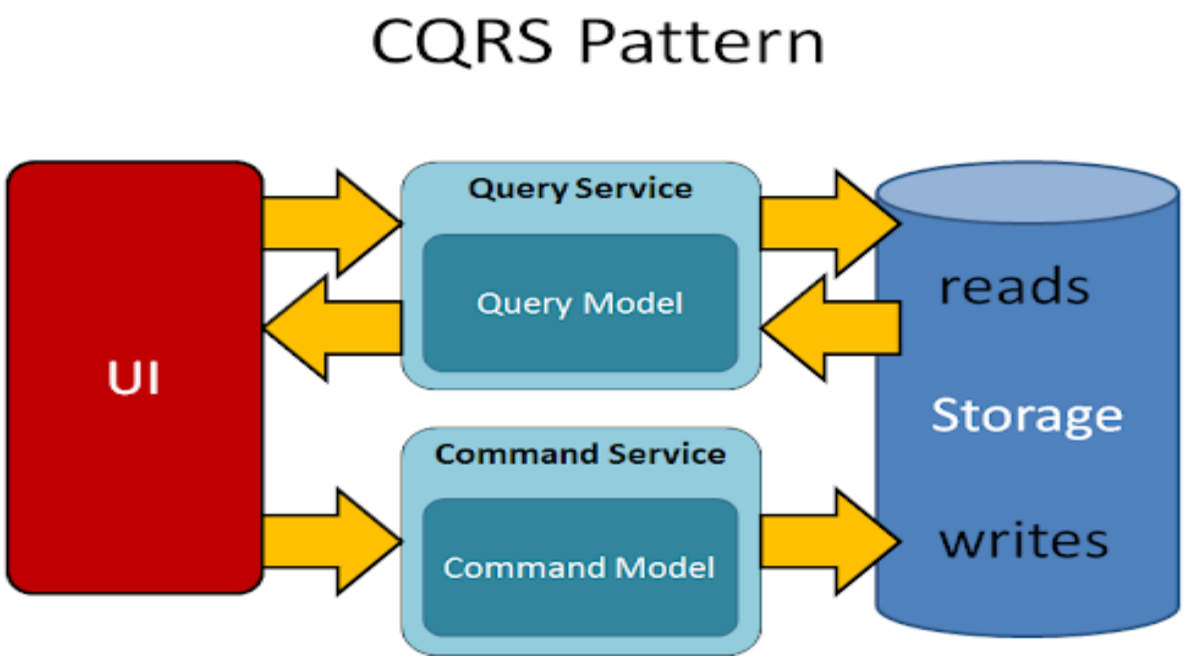

Figure 136 CQRS Pattern

7.10.2.1 Introduction

CQRS, an acronym for Command Query Responsibility Segregation, is a design pattern that has gained popularity in recent years for developing complex applications. It proposes the separation of responsibilities between commands, which are responsible for modifying the application's state, and queries, which retrieve data from the application. Unlike traditional architectures where a single model is used for both reading and writing, CQRS advocates for the use of separate models optimized for their specific tasks.

The central idea behind CQRS is to recognize that reading and writing operations have different requirements and characteristics. Queries are typically more frequent than commands, and they often involve complex data retrieval or aggregation. On the other hand, commands are responsible for modifying the state of the application and enforcing business rules.

By segregating the models for commands and queries, CQRS enables developers to optimize each model independently. The read model can be designed to be highly optimized for querying, employing denormalization, caching, or other techniques to enhance performance. The write model, on the other hand, can focus on enforcing complex business rules, maintaining data consistency, and handling transactions.

7.10.2.2 Benefits of CQRS

Improved Performance: With CQRS, you can optimize the read model to deliver fast and efficient querying. By denormalizing data or using specialized data storage technologies, you can significantly improve the performance of read-heavy operations. Separating the read and write models also allows you to scale each model independently based on its specific requirements, ensuring optimal performance for both reads and writes.

CQRS provides scalability benefits by enabling horizontal scaling of the read and write models independently. As the application load increases, you can scale the read side independently to handle high read loads. This scalability flexibility ensures that your application remains responsive and can handle increased traffic. Additionally, CQRS can facilitate the use of event sourcing, enabling efficient handling of high-frequency events and ensuring consistency across the system.

CQRS allows for flexibility in structuring and representing data. The read model can be optimized for querying purposes, utilizing techniques like pre-aggregation or data denormalization. This flexibility in data representation enables efficient querying, reporting, and analytics scenarios. On the write side, CQRS supports the use of event sourcing, where changes to the application state are captured as a sequence of events. This approach enables complex auditing, temporal querying, or rebuilding of state from historical events.

CQRS simplifies the maintenance and evolution of the application over time. With the separation of models, changes to one side of the application do not impact the other side. This decoupling allows for easier modifications and enhancements without affecting the overall system behavior. For example, adding new read models or introducing new business rules to the write side can be done independently, reducing the risk of unintended side effects.

7.10.2.3 Key Concepts of CQRS

To fully understand CQRS, it is important to grasp some key concepts associated with this design pattern:

Commands: Commands represent operations that modify the state of the application. They encapsulate the intention to perform an action and typically contain the necessary data and parameters to carry out that action. Commands can be initiated by user interactions or triggered by internal events within the system.

Queries: Queries are operations that retrieve data from the application. They are responsible for fetching information from the read model without modifying the state of the system. Queries can be simple or complex, involving filtering, sorting, and aggregation operations to retrieve specific subsets of data.

Command Handlers: Command handlers are responsible for processing and executing commands. They receive commands, validate them, enforce business rules, and update the write model accordingly. Command handlers are typically implemented as part of the application layer and interact with the domain layer to enforce business logic and maintain data consistency.

Query Handlers: Query handlers are responsible for processing queries and returning the requested data. They receive queries, execute the necessary operations on the read model, and return the results. Query handlers are often implemented as part of the application layer and interact with the read model to retrieve the required data.

Event Sourcing: Event sourcing is a technique commonly used in CQRS, where changes to the application state are captured as a sequence of events. Instead of directly modifying the state, commands generate events that represent the changes made. These events are then stored and can be replayed to rebuild the state of the application at any given point in time.

Domain Model: The domain model represents the core business logic of the application. It defines the entities, aggregates, and business rules that govern the behavior of the system. The domain model is responsible for enforcing invariants, performing validations, and maintaining the integrity of the data.

7.10.2.4 Implement CQRS in project

Our application adopts the CQRS (Command Query Responsibility Segregation) pattern, ensuring that all endpoints adhere to the principles and benefits of this architectural approach. CQRS is a pattern that emphasizes the segregation of responsibilities between read and write operations within a system. By decoupling these operations, we can achieve greater scalability, performance, and maintainability.

In our implementation, each endpoint is designed with a clear distinction between commands and queries. Commands represent operations that modify data or trigger actions, such as creating, updating, or deleting resources. On the other hand, queries are responsible for retrieving data without modifying it, supporting operations like fetching records or generating reports.

By segregating these responsibilities, we can optimize the system for its respective tasks. Write operations are typically associated with high concurrency and contention, as multiple users may attempt to modify the same resources simultaneously. This in our case is applied to many endpoints that will be discussed in the next part in detail, but for now they for the case of the User they are:
-Login

-Register

-Forgot password

-Validate Email

-Activate Email

To handle this efficiently, we employ dedicated write-side components that ensure data integrity, enforce business rules, and handle complex transactional operations. This separation allows us to

scale and tune these components specifically for write-intensive workloads, enhancing performance and reducing contention.

Similarly, read operations, which often involve retrieving large amounts of data or performing complex aggregations, in our case getting the list of Bug Tickets for each car are handled by dedicated read-side components. These components are optimized for fast data retrieval and can be scaled independently, enabling horizontal scalability to support high-volume read operations efficiently.

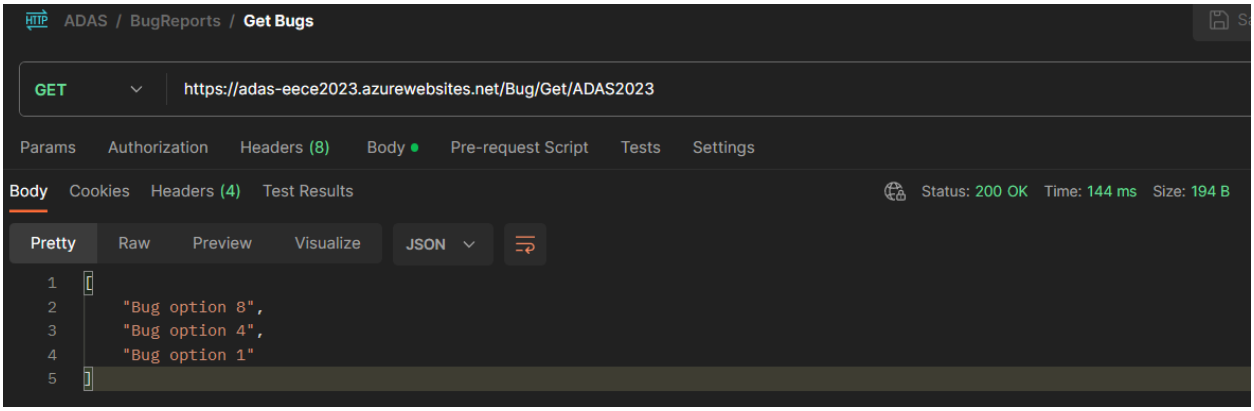

Figure 137 Get Bug Ticket titles request

Furthermore, the segregation of commands and queries simplifies system maintenance and extensibility. As the read and write responsibilities are clearly defined, modifications and enhancements can be made to each side independently, minimizing the risk of unintended side effects.

Overall, by implementing CQRS in all endpoints, our application achieves a more scalable, performant, and maintainable architecture. It optimizes resource allocation, improves concurrency handling, and provides flexibility for future enhancements. Embracing the principles of CQRS empowers us to build a robust and efficient system that meets the evolving needs of our users and business requirement.

7.10.3 APIs Implemented

- **User APIs**

7.10.3.1 Register User

Request Body

```
{
  "email": "test@gmail.com"
}
```

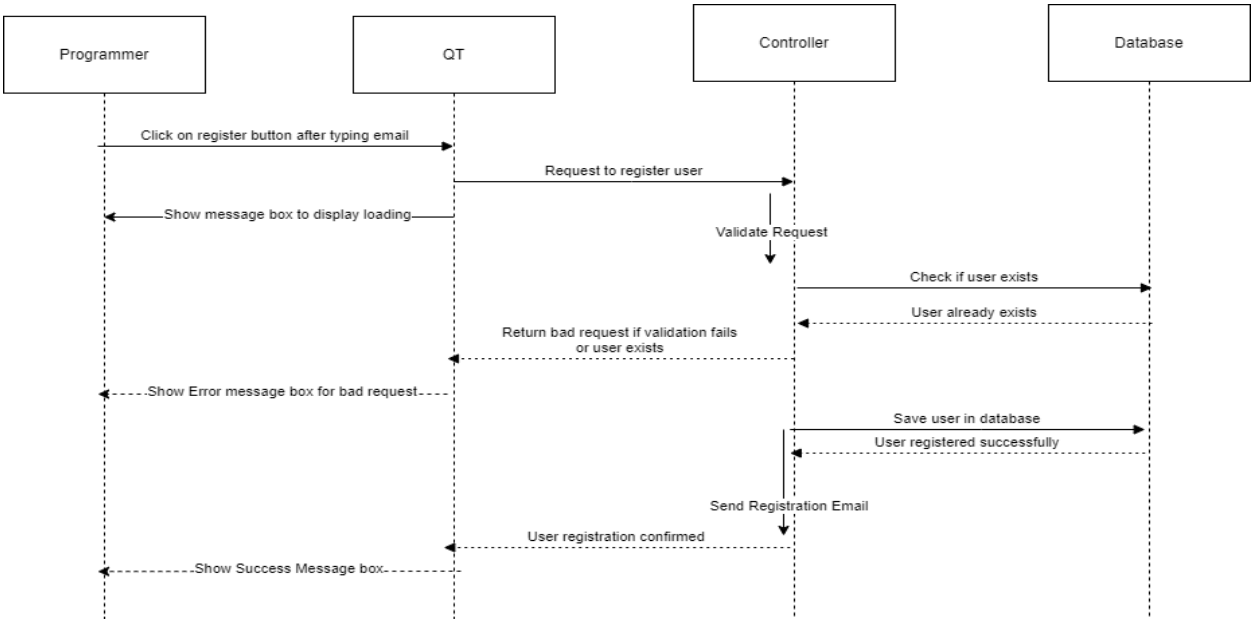

Figure 138 Register User Sequence Diagram

7.10.3.2 Login User

Request Body

```
{
  "email": "test@gmail.com",
  "password": "123"
}
```

Response

```
{
    "remainingLoginAttempts": 4,
    "id": null
}
```

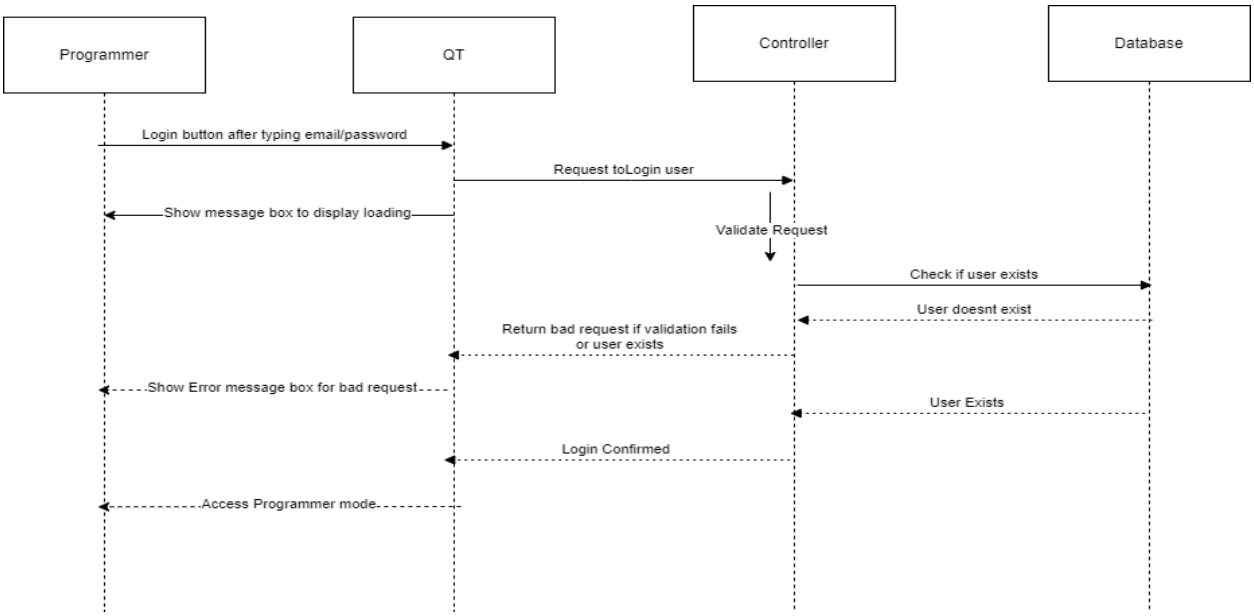

Figure 139 Login User Sequence Diagram

7.10.3.3 Forgot Password

```
{
  "email": "test@gmail.com"
}
```

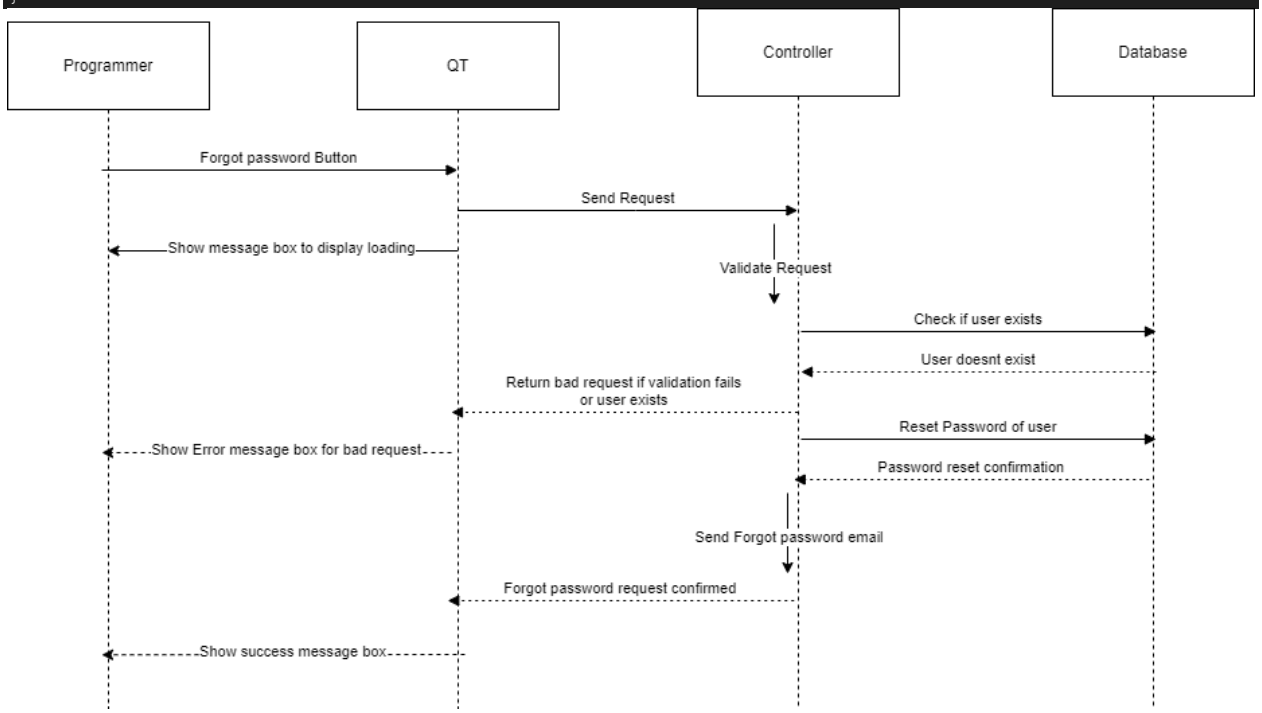

Figure 140 Forgot password Sequence Diagram

7.10.3.4 Validate Email

```
{
  "email": "test@gmail.com"
}
```

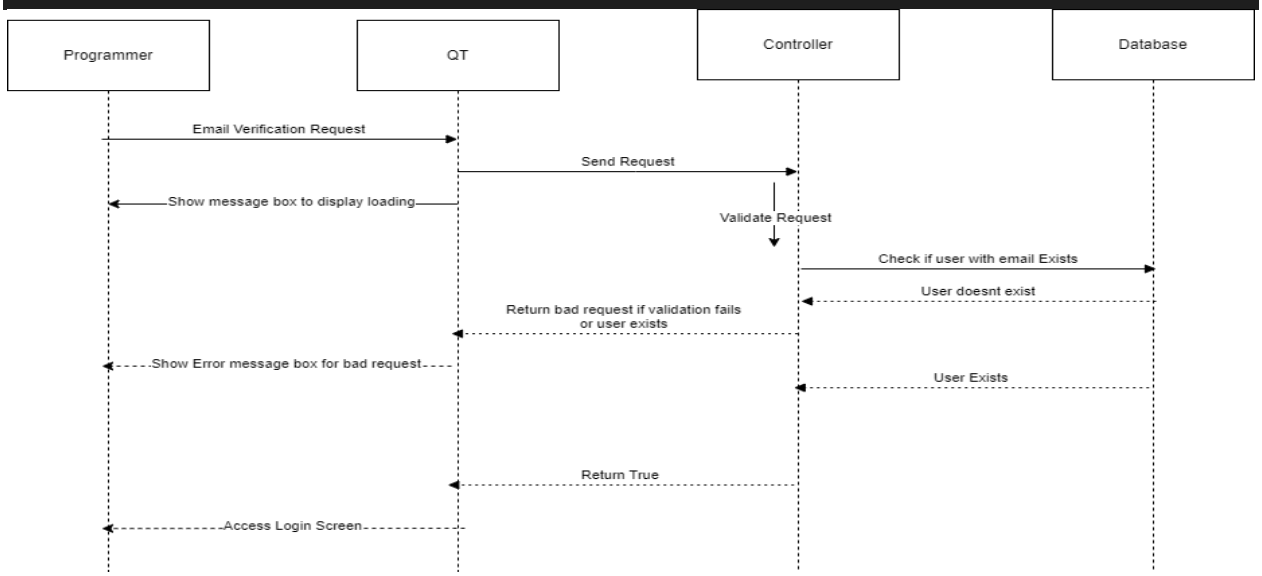

Figure 141 Validate Email Sequence Diagram

7.10.3.5 Subscribe

If you click on this link, I have integrated with stripe (payment gateway) to be able to subscribe to our services with special features. Of course this is just a demo and no payments can be made at the time being.

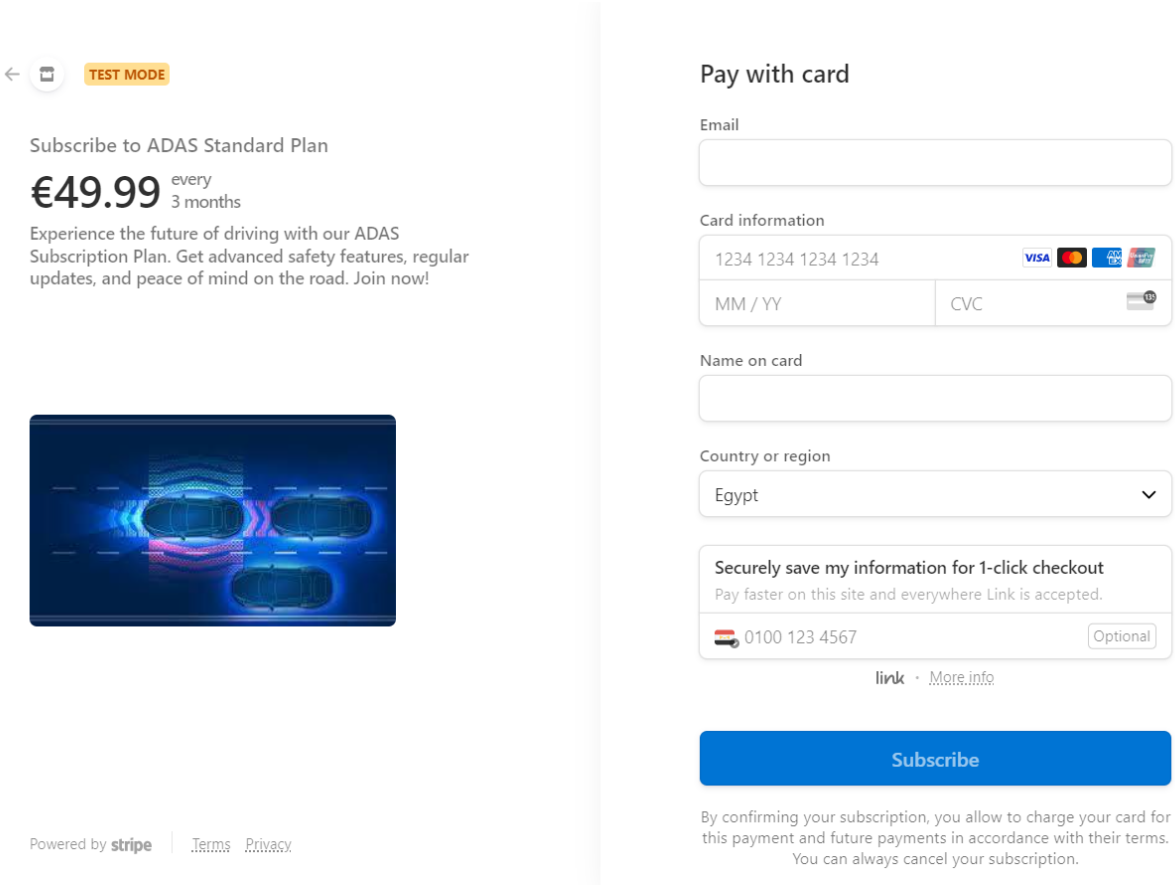

Figure 142 Subscription page

- **Bugs APIs**

7.10.3.6 Create Bug

Request body

```
{
  "title": "Issue with user settings",
  "carId": "ADAS2023"
}
```

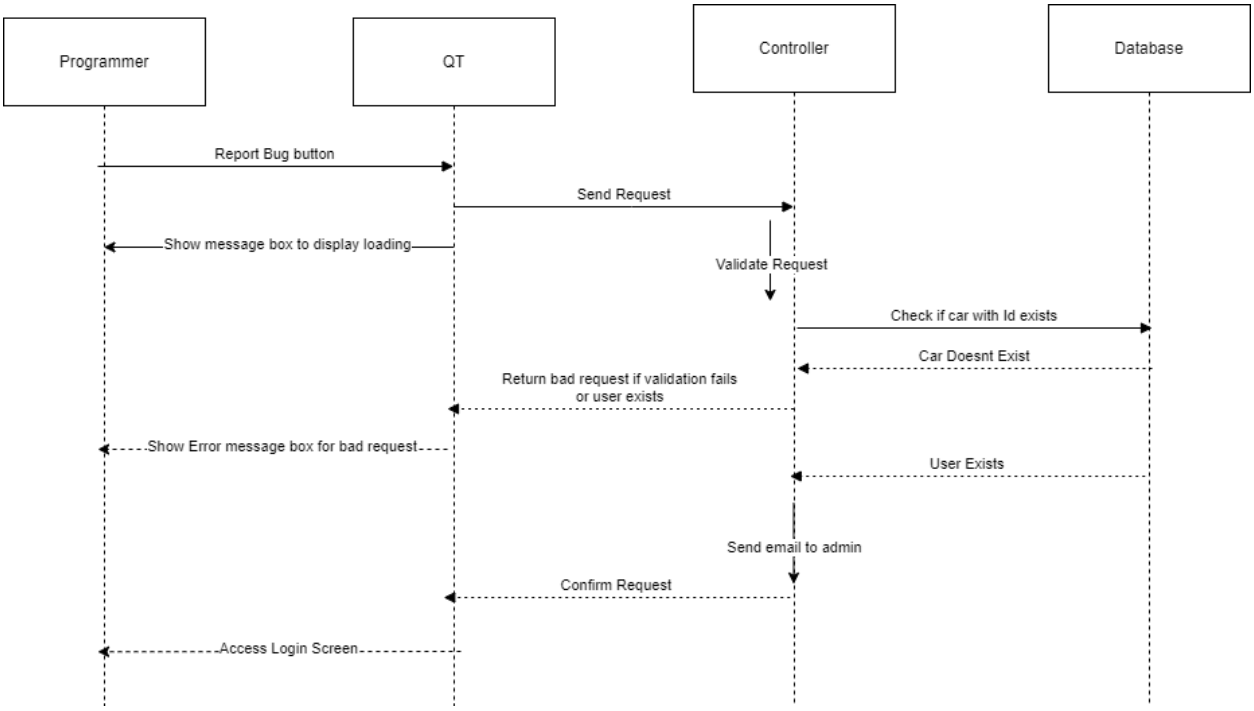

Figure 143 Create Bug Ticket Sequence Diagram

7.10.3.7 Get Bug Tickets

Request
Route param (https://adas-eece2023.azurewebsites.net/Bug/Get/{id})

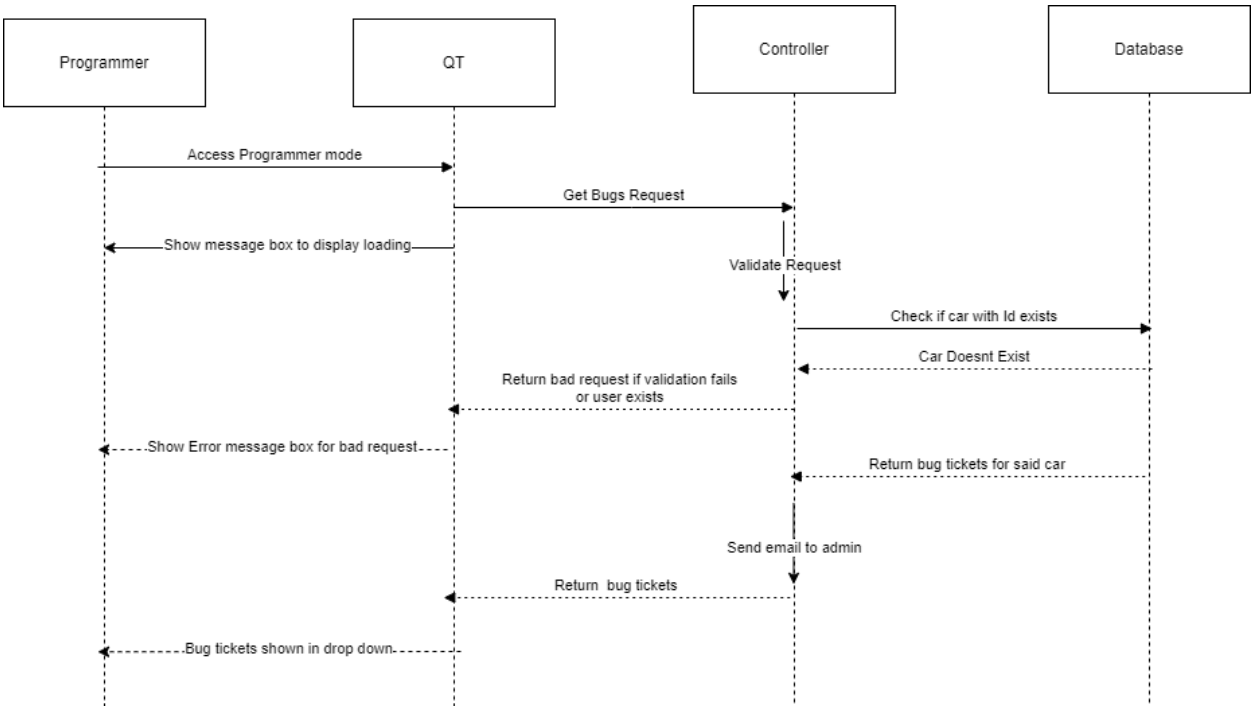

Figure 144 Get Bug Tickets Sequence Diagram

7.10.4 Azure Hosting

7.10.4.1 Database

Created azure database for PostgresSQL on the azure portal. The database is also configured to hold geo-spatial data for future work so that the position of every car would be tracked in the database and any calculations done between cars would be database side optimizing the server.

The following image contains the schema.

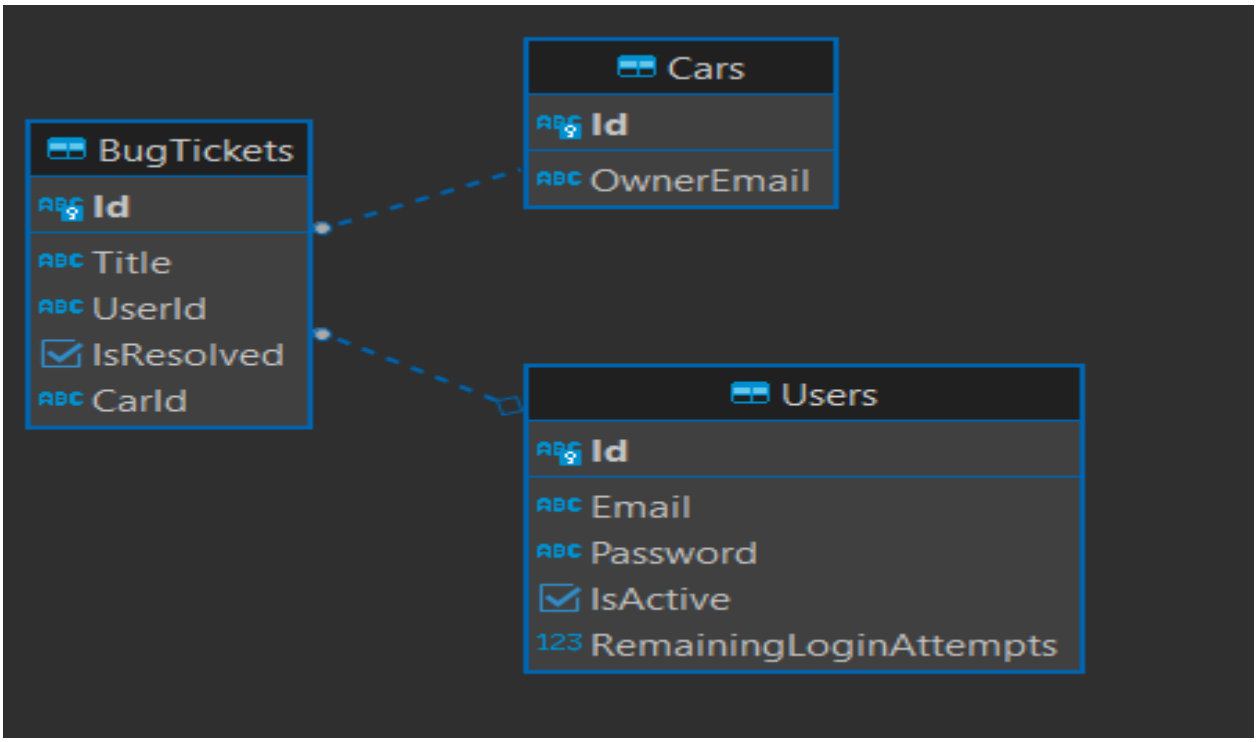

Figure 145 Database Schema

After creating the database, from the configuration tab I get the connection string required to connect to the database and in the code I automatically run migrations on startup that configure the database with the corresponding tables, relationships, foreign keys and constraints.

```
var scope = app.Services.CreateAsyncScope();
var db = scope.ServiceProvider.GetService<AdasDBContext>();
await db.Database.MigrateAsync();
```

Figure 146

7.10.4.2 App service

The app service plays a vital role in managing and maintaining a .NET Web API server. As a critical component of the overall architecture, the app service is responsible for ensuring the smooth operation and availability of the server while handling various aspects such as deployment,

scalability, monitoring, and security. In this article, we will explore the importance of the app service in supporting a .NET Web API server and its key responsibilities in detail.

The app service is configured using the configuration tab where all the launchsettings would be added. These include api keys, connection strings and any sort of sensitive data like follows.

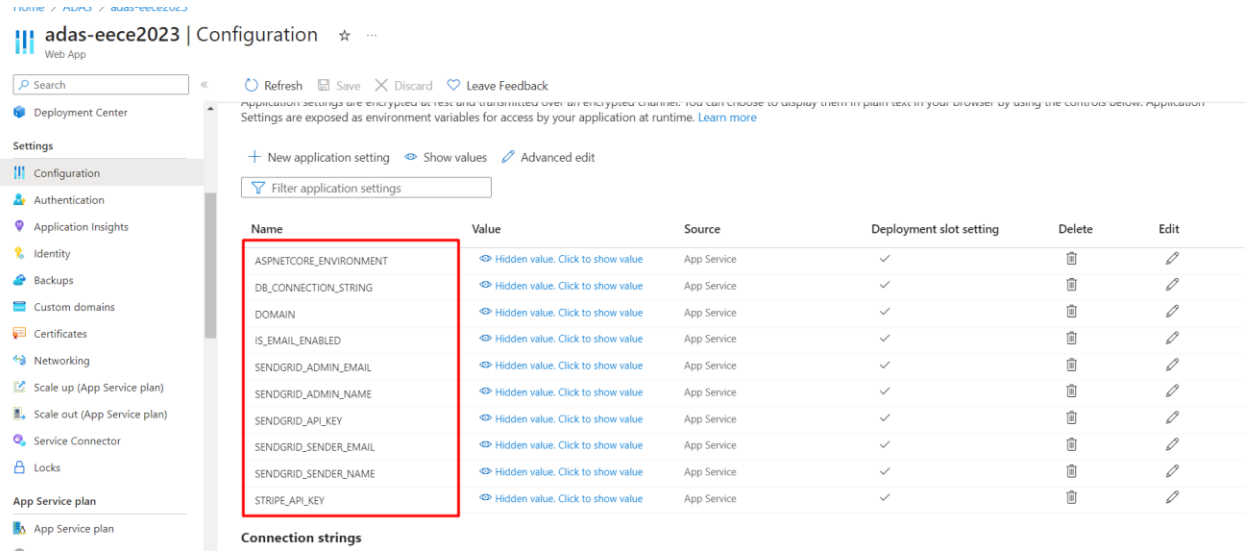

Figure 147 App service configuration

First and foremost, the app service provides a robust deployment platform for hosting the .NET Web API server. It simplifies the process of deploying the server by offering a range of deployment options, including continuous integration and deployment (CI/CD) pipelines. With the app service, developers can easily publish their applications, manage deployment slots, and automate the release process, ensuring efficient and streamlined deployment of the Web API server.

Scalability is another crucial aspect handled by the app service. It enables seamless scaling of the server to meet varying levels of demand. Through features like automatic scaling, developers can define scaling rules based on metrics such as CPU usage, memory consumption, or request count. The app service automatically adjusts the server's capacity, ensuring optimal performance during peak times and cost-effective resource utilization during low-demand periods.

Monitoring and diagnostics are essential for maintaining the health and performance of the Web API server. The app service provides built-in monitoring capabilities, allowing developers to collect and analyze metrics, logs, and traces. This data helps identify issues, troubleshoot problems, and optimize performance. With integrated logging and monitoring tools, the app service enables proactive monitoring and ensures the server's reliability and availability.

Security is a critical concern for any web application, and the app service addresses this aspect comprehensively. It offers various security features, including authentication and authorization mechanisms, SSL/TLS certificates for secure communication, and integration with Azure Active

Directory for identity management. The app service also supports advanced security features such as IP whitelisting, firewall rules, and application gateway integration, ensuring robust protection for the Web API server and its resources.

High availability is another key responsibility of the app service. It provides redundancy and fault tolerance by distributing the server across multiple availability zones or regions. With automatic load balancing and failover capabilities, the app service ensures that the Web API server remains accessible and responsive, even in the event of infrastructure or application failures. This enhances the server's reliability and minimizes downtime, delivering a seamless experience to users.

Furthermore, the app service integrates with various Azure services, offering a wide range of additional capabilities to enhance the Web API server's functionality. It seamlessly integrates with Azure SQL Database for data storage, Azure Key Vault for secure key management, and Azure Application Insights for advanced application monitoring and analytics. These integrations enable developers to leverage the power of Azure's ecosystem to enhance their Web API server and build robust, scalable solutions.

In conclusion, the app service plays a critical role in managing and maintaining a .NET Web API server. With its deployment, scalability, monitoring, security, and integration capabilities, the app service provides a reliable and efficient platform for hosting and managing the server. By leveraging the app service's features and functionalities, developers can focus on building their Web API server's core functionality while relying on a robust and scalable infrastructure.

Chapter 8: Device Drivers

8.1 Device Drivers Definition

Device drivers are essential software components that enable communication between the operating system and hardware devices in a computer system. In embedded systems, device drivers are particularly important as they allow the system to interact with the various sensors, actuators, and other peripherals that make up the system. The device driver acts as a translator, providing a standardized interface between the hardware and the operating system, allowing applications to interact with the hardware without needing to know the specific details of how it works. In this chapter, we will explore the role of device drivers in embedded Linux systems, the different types of device drivers, and the process of developing and integrating device drivers into an embedded Linux system.

8.2 Linux Kernel Architecture

Device drivers are part of the kernel, the core of the operating system that manages system resources and provides a bridge between the hardware and the software. Understanding the Linux kernel architecture is essential for developing device drivers in Embedded Linux. The Linux kernel is modular, with different subsystems responsible for different functions, such as memory management, process scheduling, and device drivers. Device drivers are implemented as kernel modules, which can be dynamically loaded and unloaded as needed. The Linux kernel provides a set of APIs (Application Programming Interfaces) for device drivers to interact with the hardware,

including functions for accessing memory-mapped I/O, interrupt handling, and DMA (Direct Memory Access). In this chapter, we will provide an overview of the Linux kernel architecture and how device drivers fit into it, including the kernel's role in managing device resources and the different subsystems involved in device drivers.

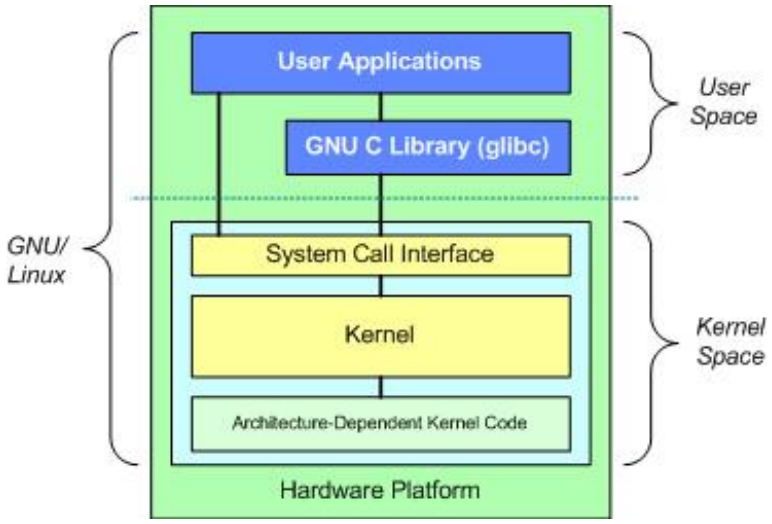

Figure 148 Fundamental Architecture of Linux

As seen in the above figure, the architecture of Linux is divided into two main parts which are User Space and Kernel Space.

1. User Space

User programs and applications are executed in user space, which is a protected area of memory that cannot directly access system hardware. Instead, user space applications must interact with the kernel, which operates in kernel space and provides an interface for accessing hardware resources. User space programs can access limited parts of memory through system calls. Because of this protection, crashes in user mode are recoverable. The GNU C library provides a mechanism for switching between user space applications and kernel space, allowing programs to access the full range of hardware resources.

2. Kernel Space

Kernel programs are executed in kernel space, which has direct access to all system hardware resources, including RAM and hard disk. The kernel is divided into different blocks and modules that manage various operations, such as file management, memory management, and process management, both in kernel space and in user space applications. Kernel space also includes the system call interface, which serves as the intermediary layer between user space and kernel space. Through system calls such as open(), write(), and read(), user space applications can interface with the kernel.

The kernel is hardware-independent and can run on any processor supported by Linux, such as Intel, ARM, and Atmel. It acts as a resource manager in kernel space and performs a wide range of operations, including process management, file management, memory management, interrupt

handling, and scheduling. With its powerful structure and capabilities, the kernel is able to handle all kinds of operations in an efficient and effective manner.

8.3 Linux Kernel Module

Kernel modules are code sections that can be uploaded or removed from the kernel as required, allowing the kernel to expand its capabilities without requiring a system reboot. There are two methods for adding custom code to Linux kernels.

- The first is to include the code in the kernel source tree and compile the kernel again.
- The second, more effective method is to load the code into the kernel while it is running. This process is known as module loading, and the code being added to the kernel is referred to as a loadable kernel module (LKM) because it is not part of the official Linux kernel and is loaded at runtime.

However, it is still a critical component of the kernel and communicates with the base kernel to perform its functions. The base kernel is always loaded during machine boot-up from the /boot directory, while LKMs are loaded afterward.

Loadable kernel modules (LKMs) can accomplish a range of tasks, but they can be broadly classified into three main categories:

- Device drivers

A device driver is intended for a particular hardware component, allowing the kernel to communicate with it without needing to understand the hardware's inner workings.

- Filesystem drivers

A filesystem driver is responsible for interpreting the contents of a filesystem, which usually includes files and directories stored on disk drives or network servers. Since there are various methods of organizing and storing files and directories, each requires a specific filesystem driver. For instance, Linux disk drives commonly use the ext2 filesystem type, which has its own filesystem driver, as does MS-DOS and NFS.

- System calls

System calls are utilized by userspace programs to obtain services from the kernel. The kernel includes system calls to read files, create new processes, and shut down the system, among other functions. As most system calls are essential and standardized, they are typically included in the base kernel and are not available as LKMs.

However, it is possible to create and install a custom system call as an LKM. Alternatively, if you are dissatisfied with how a particular aspect of Linux works, you can use an LKM to override an existing system call.

8.3.1 Advantages of LKMs

Loadable kernel modules (LKMs) offer several advantages. One significant advantage is that they eliminate the need to rebuild the kernel each time a new device is added or an existing device is upgraded. This saves time and helps maintain the integrity of the base kernel by reducing the chance of errors.

Additionally, LKMs are highly flexible and can be loaded or unloaded using a single command line. This helps conserve memory since the LKM is only loaded when it is required.

8.3.2 Differences between Kernel Modules and User Programs

Kernel modules operate within a separate memory address space and run in kernel space, while applications run in userspace. This separation protects system software from user programs. Kernel space and userspace have their own memory address spaces.

Kernel modules possess higher execution privileges, granting them greater privilege than code that runs in userspace.

Unlike a user program, which typically executes sequentially and completes a single task from start to finish, a kernel module does not operate sequentially. Instead, it registers itself in order to serve future requests.

Kernel modules necessitate a distinct set of header files compared to user programs. As such, they require different header files to function properly.

8.3.3 Difference Between Kernel Drivers and Kernel Modules

A kernel module is a pre-compiled code segment that can be dynamically loaded into the kernel during runtime using tools like insmod or modprobe.

On the other hand, a driver is a code segment that operates within the kernel and facilitates communication with specific hardware devices. The driver "drives" the hardware by enabling the kernel to interact with it. Nearly every hardware device in a computer requires an associated driver.

8.4 Device Driver

A device driver is a specialized type of software application that facilitates communication between software and hardware devices. The absence of a device driver results in the hardware device being non-functional.

Typically, a device driver communicates with the hardware device through the communications subsystem or computer bus to which the device is connected. Since device drivers are specific to particular hardware and operating systems, they are both hardware-dependent and operating system-specific.

Furthermore, device drivers act as translators between hardware devices and the programs or operating systems that use them.

8.4.1 Types of Devices

- Character Device

A char file is a type of hardware file that reads and writes data character by character. Examples of char files include keyboards, mice, and serial printers. If a user employs a char file to write data, other users cannot use the same char file simultaneously to write data, as this would impede access to the file.

Character files employ synchronization techniques to write data, and they are commonly used for communication purposes. Unlike other file types, char files cannot be mounted.

Character Devices can be sensors, motors, LCD, etc.

- Block Device

A block file is a type of hardware file that reads and writes data in blocks rather than character by character. Block files are particularly useful when dealing with large amounts of data. All disk drives, such as HDDs, USBs, and CDROMs, are examples of block devices. When formatting block devices, the block size is a crucial consideration.

Writing data to block files is an asynchronous process and is computationally intensive. Block devices are used for storing data on physical hardware and can be mounted so that the data can be accessed.

- Network Device

In Linux's network subsystem, a network device is defined as an entity that transmits and receives data packets. Typically, a network device refers to a physical device such as an Ethernet card. However, some network devices are entirely software-based, such as the loopback device, which is employed for sending data to oneself.

8.4.2 Types of Drivers

- Static in Tree

Runs in boot time.

- Module Dynamic

Runs in user space like when you plug in a mouse of keyboard.

These two types were discussed above.

8.5 Pseudo Device Driver Example

In this section, we will be showcasing an example of a Pseudo Device Driver which displays some hello messages to the kernel.

The following is an example kernel module of pseudo device driver which displays Hello kernel and initializes the device when the module is loaded and displays “Good bye kernel” and de initializes the device when the module is unloaded.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/moduleparam.h>
#include <linux/fs.h>
#include <linux/cdev.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Hussam ");
MODULE_DESCRIPTION("A hello world Pseudo device driver");


dev_t device_number;
struct cdev st_characterDevice;
struct class *myClass;
struct device *myDevice;
static int driver_open(struct inode *device_file, struct file *instance)
{
    printk("%s mytest_driver - open was called", __FUNCTION__);
    return 0;
}
static int driver_close(struct inode *device_file, struct file *instance)
{
    printk("%s mytest_driver - close was called", __FUNCTION__);
    return 0;
}

struct file_operations fops =
{
    .owner=THIS_MODULE,
    .open=driver_open,
    .release=driver_close
};


static int __init hellodriver_init(void)
{
    int retval;
```

```
    printk("Hello Kernel\n");
    // 1-allocate device number
    retval      =      alloc_chrdev_region(&device_number,      0,      1,
"test_devicenumber");
    if(retval == 0)
    {
        printk("%s retval=0 - registered device Major number: %d, Minor
Number: %d\n", __FUNCTION__,MAJOR(device_number), MINOR(device_number));

    }
    else
    {
        printk("Could not register device number");
        return -1;
    }
    // 2- define driver character or block or network
    cdev_init(&st_characterDevice, &fops);
    retval = cdev_add(&st_characterDevice, device_number, 1);
    if(retval != 0)
    {
        /* In case of failure, you have to delete everything made before */
        unregister_chrdev_region(device_number, 1);
        printk("Registering the device to the kernel failed!\n");
        return -1;
    }
    // 3- generate file (class - device)
    if( (myClass = class_create(THIS_MODULE, "test_class")) == NULL )
    {
        /* In case of failure, you have to delete everything made before */
        printk("Device class can not be created!\n");
        cdev_del(&st_characterDevice);
        unregister_chrdev_region(device_number, 1);
        return -1;
    }
    myDevice    =    device_create(myClass,    NULL,    device_number,    NULL,
"test_file");
    if(myDevice == NULL)
    {
        /* In case of failure, you have to delete everything made before */
        printk("Device can not be created!\n");
        class_destroy(myClass);
        cdev_del(&st_characterDevice);
        unregister_chrdev_region(device_number, 1);
        return -1;
    }
```

```
    printk("Device Driver is created successfully !\n");
    return 0;
}

static void __exit hellodriver_exit(void)
{
    cdev_del(&st_characterDevice);
    device_destroy(myClass, device_number);
    class_destroy(myClass);
    unregister_chrdev_region(device_number, 1);
    printk("Good bye Kernel\n");
}

module_init(hellodriver_init);
module_exit(hellodriver_exit);
```

To compile this kernel module first a Makefile is created with the following lines:

```
obj-m += trial4.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

This Makefile runs through the following command “make” which goes to the kernel source and use its makefile to build the kernel module and generate the kernel object file which will be loaded by using this command “sudo insmod trial4” and a file named trial4.ko will be added to current working directory. Also, when “make clean” command is used it deletes the kernel object file which is recently created.

This command “uname -r “ specifies which kernel version is used.

The kernel module object file will be loaded into the kernel by using this command “sudo insmod trial4.ko”.

To see the kernel messages “sudo dmesg -c” will be used.

```
hussam@hussam-VirtualBox:~/Desktop/Device-Drivers/Device-Drivers$ sudo dmesg -c
[ 1789.319579] Hello Kernel
[ 1789.319582] hellodriver_init retval=0 - registered device Major number: 234, Minor Number: 0
[ 1789.319880] Device Driver is created successfully !
```

Figure 149 Showing the output of sudo dmesg -c after inserting the kernel module

As we can see in the above figure, the function “hellodriver_init” is called making the following:

- Printed “Hello Kernel”
- Registered a device driver with a major number of 234 and minor number of 0.

- Created a device file called “test_file” which can be found under “/dev” directory.
- Created a class file called “test_class” which can be found under “/sys/class”.

The device file called “test_file” is the file which is used by the user space to interact with the kernel module. For example, when you read this file with the command “cat test_file” the function called “driver_open” will be called and when file is closed a function called “driver_close” will be called.

These functions is specified for the kernel module in the struct called “file_operations” which has three members:

- .owner : takes the owner ‘s name of the module’s developer.
- .open : takes the name of the function which is to be called whenever an attempt to read the device file is made
- .release : takes the name of the function which is to be called whenever an attempt to close the device file is made.

When the module is unloaded using this command “sudo rmmod trial4” the function “hellodriver_exit” is called which de-initializes everything made in the init function including the deletion of the device file and the class file created.

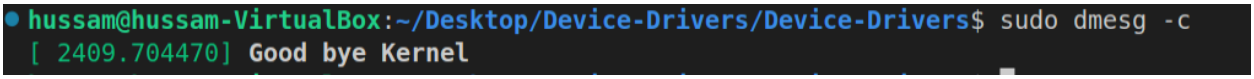

Figure 150 Showing the output of sudo dmesg -c after removing the kernel module

8.6 Kernel Module for DHT-11 Sensor

In this section we will be discussing the kernel module for DHT-11 to the Yocto Image to be able to get readings of temperature and humidity.

The code for this kernel module is based on the DHT-11 kernel module provided by Linux as it supports the DHT-11 sensor.

We will not be discussing the code of this module; it will be found in the project files.

First, we will create a recipe for the sensor called “dht11km” in the “recipes_kernel” folder in our custom layer “meta-mylayer”

Figure 151 Listing the files inside the kernel module folder

The folder called “files” contains the kernel module source file and the Makefile used to compile and build the kernel module.

```
hussam@hussam-VirtualBox:~/poky/meta-mylayer/recipes-kernel/dht11km/files$ ls
dht11km.c  Makefile
```

Figure 152 Listing the files inside files directory

The Makefile is used by Yocto to build and compile the kernel module and produce the kernel module object file which will be loaded into the kernel automatically.

```
obj-m := dht11km.o
SRC := $(shell pwd)
all:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC)
modules_install:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install
```

The variables “KERNEL_SRC” and “MAKE” are defined in Yocto which tells Yocto where is the kernel source which will be used to compile and build the module.

As for the recipe, it inherits the module class to tell Yocto that this will be a kernel module and the source file and the Makefile are provided.

```
SUMMARY = "This is a hello world kernel module recipe"
DESCRIPTION = "${SUMMARY}"
LICENSE = "CLOSED"
inherit module
SRC_URI = "file://Makefile \
           file://dht11km.c \
          "
S = "${WORKDIR}"
KERNEL_MODULE_AUTOLOAD += "dht11km"
# The inherit of module.bbclass will automatically name module packages with
# "kernel-module-" prefix as required by the oe-core build environment.
RPROVIDES:${PN} += "kernel-module-dht11km"
```

The kernel module is automatically loaded into the kernel without the need for manually using the command “insmod dht11km.ko” to load the module and this kernel object file will be found under “/lib/modules/extra”.

To check if the module is installed correctly or not, the command “lsmod” will be used to list all the loaded modules in the kernel.

Now, the DHT-11 sensor kernel module will be loaded automatically into the kernel and the temperature and humidity will be read through the device file “/dev/dht11km” and both the temperature and humidity will be displayed in the kernel and to see them, the command “dmesg -c” will be used.

Also, the device driver code prints the temperature and humidity to the user space in the currently opened terminal by using some specific function called “copy_to_user” which copies the data in the buffer in form of string to the user which will be displayed in the terminal.

These readings can be used later to be displayed in the GUI to give information about the current temperature and humidity to the user through the GUI.

Chapter 9: Extra Tools

In this chapter, we will be discussing some of the tools that are used during the project which has helped us a lot to complete the project much faster and efficiently.

9.1 Bash Script

Bash (Bourne-Again SHell) is a Unix shell and command language that is commonly used in Linux and macOS systems. Bash scripts are used to automate tasks and perform various operations using command-line tools. Bash scripts are written in plain text files and can be executed on the command line or as part of a larger script.

Bash scripts are useful for automating repetitive tasks, such as file management, system configuration, and data processing. They can be used to combine different command-line tools and perform complex operations that would be difficult or impossible to do manually.

In addition to the standard Unix tools, Bash scripts can also use various programming constructs, such as variables, loops, and conditional statements, to make the scripts more flexible and powerful. By using Bash scripts, developers can save time and reduce errors by automating repetitive tasks and ensuring that all components of a project are properly configured.

Bash scripts are written in plain text files with the .sh extension. To make a Bash script executable, you need to set the execute permission using the chmod command, like this:

```
chmod +x myscript.sh
```

The basic syntax of a Bash script is as follows:

```
#!/bin/bash
# This is a comment

# Declare variables
VARNAME=value

# Execute commands
command1
command2
...
```

The “#!/bin/bash” line at the beginning of the script specifies the shell that should be used to execute the script. Comments in Bash scripts start with the # character and are used to provide documentation and explanations for the code.

Variables in Bash scripts are declared using the syntax VARNAME=value, where VARNAME is the name of the variable and value is its initial value. To reference the value of a variable, use the syntax $VARNAME.

Bash scripts execute commands just like you would on the command line. Commands can include system commands, shell built-in commands, or other scripts. To execute a command, simply type its name followed by any arguments.

Once you have written a Bash script, you can execute it by typing its name on the command line, like this:

```
./myscript.sh
```

This will run the Bash script and execute all the commands in the script in order.

Command-line arguments in Bash scripts are used to pass values to the script when it is executed. Bash scripts can handle command-line arguments using the special variables $1, $2, $3, etc., which correspond to the first, second, third, etc. argument passed to the script.

For example, if you run the following command:

```
./myscript.sh arg1 arg2 arg3
```

The value of $1 will be arg1, the value of $2 will be arg2, and the value of $3 will be arg3.

Bash scripts can also use the shift command to shift the values of the command-line arguments to the left. For example, if you run the following command:

```
./myscript.sh arg1 arg2 arg3
```

And your script contains the following code:

```
#!/bin/bash

echo "The first argument is: $1"
shift
echo "The second argument is: $1"
shift
echo "The third argument is: $1"
```

The output will be:

The first argument is: arg1

The second argument is: arg2

The third argument is: arg3

9.2 CMake

Figure 153 CMake Logo

CMake is a tool that generates build systems for various platforms from a single, platform-independent configuration file. This means that you can write a single CMakeLists.txt file that defines the project's build process, and then use CMake to generate the appropriate build system for the platform you're working on. CMake is designed to be highly portable and can generate build systems for a wide variety of platforms and compilers.

The build process for a CMake project is defined in a CMakeLists.txt file, which lists the project's source files, libraries, and dependencies. These files are written in a simple scripting language that allows you to define the build process in a way that is both easy to read and highly customizable. CMake provides a large number of built-in commands that you can use in your CMakeLists.txt files to define the build process.

CMake is designed to be highly flexible and supports a wide variety of build systems, including Makefiles, Ninja, and Visual Studio solutions. This means that you can use CMake to generate build systems that are compatible with the tools that you're familiar with. CMake's support for multiple build systems also makes it easier to port projects between different platforms and compilers.

CMake has a modular design that allows you to customize the build process in a wide variety of ways. CMake provides a large number of built-in commands that you can use to define the build process in your CMakeLists.txt files, and you can also write your own custom commands to extend CMake's functionality. This modular design makes it easy to create complex build processes that are tailored to your specific needs.

Some CMake commands will be shown:

1. cmake_minimum_required() sets the CMake version to be used.
2. project() sets the project name.

3. add_subdirectory() is used to add a subdirectory to the build process. This command tells CMake to look for a CMakeLists.txt file in the specified directory and include it in the build process.
4. add_library() is used to create a new library target in the build process. This command tells CMake to create a new library with the specified name and source files.
5. target_include_directories() sets the directory where our include files are located.
6. target_link_libraries() is used to specify the libraries that a target (e.g., an executable or a shared library) depends on. It links the specified libraries to the target during the build process.
7. add_executable() is used to create a new executable target in the build process. This command tells CMake to create a new executable with the specified name and source files.

CMake is designed to be language-agnostic, which means that it can be used to build projects written in a wide variety of programming languages. CMake provides built-in support for many popular programming languages, including C, C++, Fortran, and Python. This means that you can use CMake to build projects that use multiple languages, and you can also easily add support for new languages by writing custom CMake modules.

CMake can generate project files for many different integrated development environments (IDEs), including Visual Studio, Xcode, and Eclipse. This means that you can use CMake to generate project files that are compatible with the IDE that you're using, which can make it easier to work with the project in your chosen development environment. CMake's support for multiple IDEs also makes it easier to collaborate with other developers who may be using different tools.

CMake is widely used in industry, and it is a standard tool for many large-scale software projects. Many popular open-source projects, such as KDE, LLVM, and OpenCV, useCMake as their build system. This means that there is a large body of knowledge and experience around using CMake, and it is a well-established tool that you can rely on for your own projects.

The following example illustrates an example of CMakeLists.txt file:

There are three directories involved. The top level directory has two subdirectories called ./Demo and ./Hello. In the directory ./Hello, a library is built. In the directory ./Demo, an executable is built by linking to the library. A total of three CMakeLists.txt files are created: one for each directory.

The first, top-level directory contains the following CMakeLists.txt file.

```
# CMakeLists files in this project can
# refer to the root source directory of the project as ${HELLO_SOURCE_DIR} and
# to the root binary directory of the project as ${HELLO_BINARY_DIR}.
cmake_minimum_required (VERSION 2.8.11)
project (HELLO)

# Recurse into the "Hello" and "Demo" subdirectories. This does not actually
# cause another cmake executable to run. The same process will walk through
# the project's entire directory structure.
add_subdirectory (Hello)
```

```
add_subdirectory (Demo)
```

Then for each subdirectory specified, CMakeLists.txt files are created. In the ./Hello directory, the following CMakeLists.txt file is created:

```
# Create a library called "Hello" which includes the source file "hello.cxx".
# The extension is already found. Any number of sources could be listed here.
add_library (Hello hello.cxx)

# Make sure the compiler can find include files for our Hello library
# when other libraries or executables link to Hello
target_include_directories (Hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

Finally, in the ./Demo directory, the third and final CMakeLists.txt file is created:

```
# Add executable called "helloDemo" that is built from the source files
# "demo.cxx" and "demo_b.cxx". The extensions are automatically found.
add_executable (helloDemo demo.cxx demo_b.cxx)

# Link the executable to the Hello library. Since the Hello library has
# public include directories we will use those link directories when building
# helloDemo
target_link_libraries (helloDemo LINK_PUBLIC Hello)
```

9.3 Makefile

Makefile is a build automation tool that is widely used in software development. It provides a way to automate the build process and manage dependencies between different parts of a project. Makefile uses a set of rules to build target files from source files, and it automatically determines which files need to be rebuilt based on their dependencies. By using Makefile, developers can save time and reduce errors by automating repetitive build tasks and ensuring that all dependencies are properly managed.

Makefile consists of a set of rules that define how to build target files from source files. Each rule consists of a target, a list of dependencies, and a set of commands to execute if any of the dependencies have changed. The syntax of a Makefile rule is as follows:

```
target: dependency1 dependency2 ...
    command1
    command2
    ...
```

The target is the file that needs to be built, and the dependencies are the files that the target depends on. The commands are the shell commands that are executed to build the target file.

Makefile supports various types of targets, including simple targets, pattern targets, and phony targets. Simple targets are files that need to be built, pattern targets are used to build files based on

a pattern, and phony targets are used to specify targets that don't correspond to actual files, but instead trigger a particular action.

Makefile also supports variables, which are used to store values that can be used throughout the Makefile. Variables are defined using the syntax VARNAME = value and can be referenced using $(VARNAME).

In addition to basic syntax, Makefile also supports more advanced features such as conditionals, loops, and functions, which are used to make the build process more flexible and powerful.

To run Makefile, simply navigate to the directory containing the Makefile and run the make command. Makefile will automatically read the Makefile and determine which files need to be built or rebuilt based on their dependencies.

By default, Makefile will build the first target listed in the Makefile, but you can specify a different target by using the syntax make target. You can also build multiple targets by listing them separated by spaces, like make target1 target2.

Makefile also supports various command-line options, such as -n to print the commands that would be executed without actually executing them, -C dir to change the current directory to dir, and -j to specify the number of jobs to run in parallel.

Once Makefile has finished building the targets, it will print a message indicating whether the build was successful or if there were any errors. If there were errors, Makefile will list the files that failed to build and the error messages.

Variables in Makefile are used to store values that can be referenced throughout the Makefile. They are defined using the syntax VARNAME = value or VARNAME := value, where the = operator performs a simple expansion and the := operator performs immediate expansion.

Variables can be used to store file names, compiler flags, or any other value that needs to be used repeatedly in the Makefile. They are referenced using the syntax $(VARNAME). For example, if you define a variable CC to store the name of the C compiler, you can use it like this:

```
CC = gcc
CFLAGS = -Wall -Werror
myprogram: main.c util.c
    $(CC) $(CFLAGS) main.c util.c -o myprogram
```

In this example, the CC variable is defined to store the name of the C compiler (gcc), and the CFLAGS variable is defined to store the compiler flags (-Wall and -Werror). These variables are used in the myprogram target to compile the source files and generate the executable.

Variables in Makefile can also be used in more advanced ways, such as to generate file names dynamically or to create conditional rules. Makefile supports various built-in functions that can be used with variables, such as $(wildcard), $(patsubst), and $(shell).

9.4 SystemD and Service Files

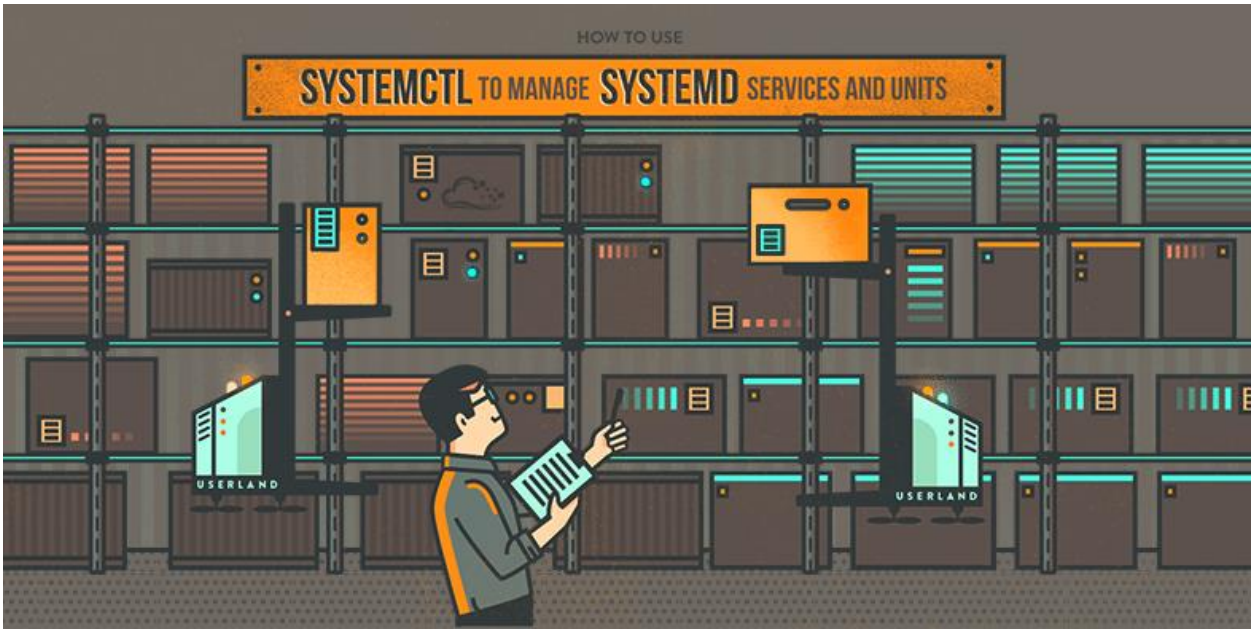

Figure 154 SystemD

Systemd is a system and service manager for Linux operating systems. It is responsible for managing the system's startup and shutdown processes, as well as the services that run on the system. A service file is a configuration file used by systemd to define a service. The service file specifies how the service should be started, stopped, and restarted, as well as other settings related to the service.

Systemd is a replacement for the traditional System V init system used in Linux. It is designed to be more efficient and flexible than the old init system. Systemd uses a dependency-based approach to managing services, which means that services are started and stopped in a specific order based on their dependencies.

Systemd also provides a few other features, such as logging, process management, and resource management. These features make it easier to manage and monitor the system, and to ensure that services are running efficiently.

A service file is a configuration file used by systemd to define a service. The service file specifies how the service should be started, stopped, and restarted, as well as other settings related to the service.

The service file is written in the systemd unit file format, which is a simple and easy-to-understand format based on key-value pairs. The service file is organized into sections, each of which contains a set of key-value pairs.

The most important sections of the service file are the [Unit], [Service], and [Install] sections. The [Unit] section specifies metadata about the service, such as its name and description. The [Service] section specifies how the service should be started, stopped, and restarted, as well as other settings related to the service. The [Install] section specifies how the service should be installed on the system.

The [Unit] section of the service file contains metadata about the service, such as its name and description. The most important settings in the [Unit] section are:

- Description: Specifies a description of the service.
- After: Specifies the services that must be started before this service.
- Requires: Specifies the services that this service requires to run.

The [Service] section of the service file specifies how the service should be started, stopped, and restarted, as well as other settings related to the service. The most important settings in the [Service] section are:

- ExecStart: Specifies the command to start the service.
- ExecStop: Specifies the command to stop the service.
- Restart: Specifies when the service should be restarted, if at all.

The [Install] section of the service file specifies how the service should be installed on the system. The most important setting in the [Install] section is:

- WantedBy: Specifies the target that the service should be installed into.

Here is an example service file that starts a simple web server:

```
[Unit]
Description=Simple Web Server

[Service]
ExecStart=/usr/bin/python /path/to/server.py
ExecStop=/usr/bin/pkill -f /path/to/server.py
Restart=always

[Install]
WantedBy=multi-user.target
```

This service file specifies that the service is a simple web server. It also specifies that the service should be started using the command "/usr/bin/python /path/to/server.py", and that it should be stopped using the command "/usr/bin/pkill -f /path/to/server.py". Finally, it specifies that the service should be installed into the "multi-user.target" target.

9.5 QMake

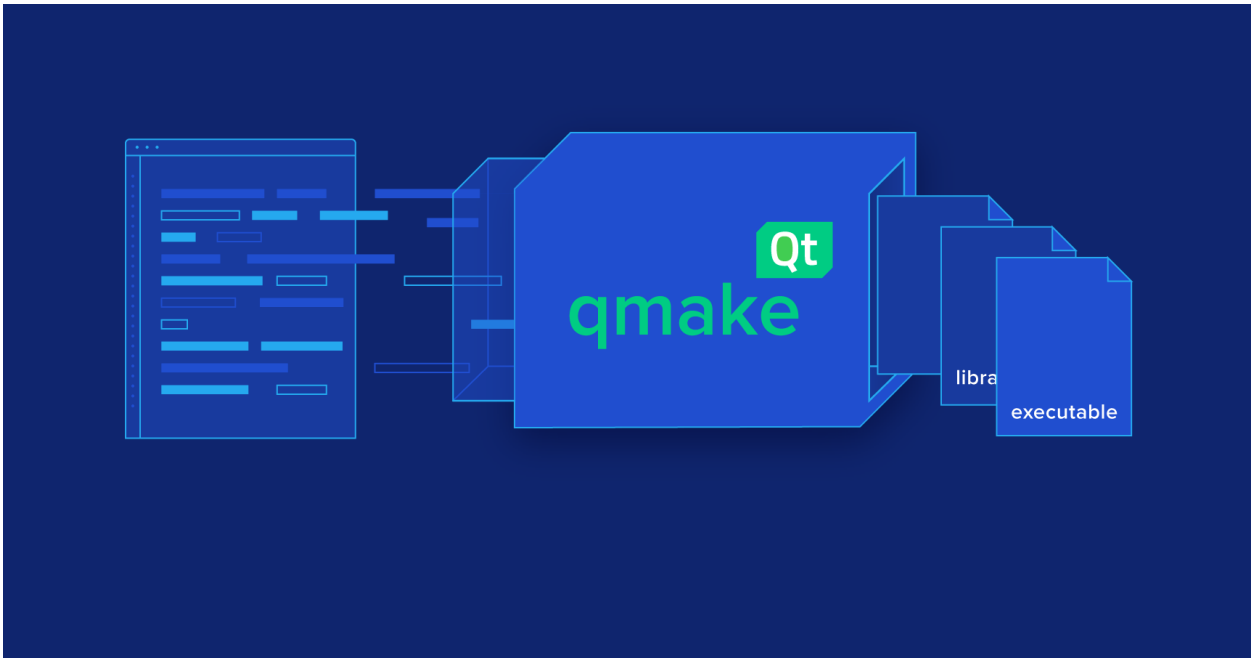

Figure 155 QMake

QMake is a build system used in the Qt application framework. It is a powerful and flexible tool that generates Makefiles for building Qt-based applications on different platforms. QMake is designed to be platform-independent, making it easier to develop cross-platform applications.

QMake uses a simple and easy-to-understand syntax, which makes it an ideal tool for novice developers. The syntax is based on variables and assignment statements, similar to the syntax used in Makefiles. This makes it easy to read and write QMake project files.

QMake is an essential tool for developing applications using the Qt framework. It provides a streamlined approach to building applications, allowing developers to focus on the development process rather than the build system. With QMake, developers can quickly and easily generate Makefiles and build their applications with ease.

A QMake project file is a simple text file that contains information about the application being built. The project file has a .pro extension and is used by QMake to generate the Makefile. The project file specifies the source files, libraries, and other dependencies required for building the application.

The QMake project file is written in the QMake language, which is a simple and easy-to-understand language based on variables and assignment statements. The project file is organized into sections, each of which contains a set of variables and assignment statements.

The most important variables in the QMake project file are TEMPLATE, TARGET, SOURCES, and HEADERS. The TEMPLATE variable specifies the type of project, which can be either app or lib. The TARGET variable specifies the name of the application or library being built. The SOURCES variable specifies the source files required for building the application, while the HEADERS variable specifies the header files required for building the application.

In addition to these variables, the QMake project file can also contain other variables that specify the build configuration, compiler flags, and other build options. These variables can be used to customize the build process and to optimize the performance of the application.

QMake provides a number of advanced features for building Qt applications. Some of the features include:

- Adding Libraries: To add libraries to your application, use the LIBS variable in the project file. This variable specifies a list of libraries to link with the application. For example, to link with the Qt Core library, you can add the following line to your project file:

```
LIBS += -lQtCore
```

- Adding Include Paths: To add include paths to your application, use the INCLUDEPATH variable in the project file. This variable specifies a list of directories to search for header files. For example, to add the directory /usr/include to the include path, you can add the following line to your project file:

```
INCLUDEPATH += /usr/include
```

- Adding Compiler Flags: To add compiler flags to your application, use the QMAKE_CXXFLAGS variable in the project file. This variable specifies a list of flags to pass to the C++ compiler. For example, to enable debugging symbols, you can add the following line to your project file:

```
QMAKE_CXXFLAGS += -g
```

- Using Precompiled Headers: Precompiled headers can speed up the compilation process by caching commonly used header files. To use precompiled headers in your application, use the PRECOMPILED_HEADER variable in the project file. This variable specifies the name of the precompiled header file. For example, to use the precompiled header file stdafx.h, you can add the following line to your project file:

```
PRECOMPILED_HEADER = stdafx.h
```

- Generating Additional Files: QMake can generate additional files during the build process using the CONFIG variable in the project file. This variable specifies a list of configuration

options for the build process. For example, to generate a resource file from a .qrc file, you can add the following line to your project file:

```
CONFIG += resources
```

9.6 Weston and Wayland

Wayland is a protocol for a display server, designed to replace the X Window System, which has been the dominant display server on Linux and other Unix-like operating systems for many years.

Unlike X, which is a complex and monolithic system, Wayland is designed to be a simpler and more modular architecture that can provide better performance and security. It provides a communication protocol between the display server (the "compositor") and the client applications, which can be written in any programming language and can use various graphics toolkits such as Qt, GTK, and SDL.

In Wayland, each client application has its own "surface" that is managed by the compositor, rather than being managed by the X server. This allows for more efficient rendering and eliminates the need for the server to manage window decorations, which are instead handled by the client application itself.

Wayland also provides improved security features, such as sandboxing and input event filtering, that are designed to prevent malicious applications from accessing sensitive information or interfering with other applications.

Wayland has been adopted by many Linux distributions, including Fedora, Ubuntu, and Debian, and is used as the default display server in several desktop environments, such as GNOME, KDE Plasma, and Enlightenment.

Weston is an open-source compositor for Wayland, a protocol for a display server that is intended to replace the X Window System. A compositor is responsible for combining graphical elements such as windows, menus, and icons into a single output frame that can be displayed on a screen or sent over a network to another device.

Weston provides a reference implementation of a Wayland compositor, and it is designed to be lightweight, modular, and easy to use. It includes a number of built-in plugins for handling input devices, creating surfaces, and providing other functionality, and it can be extended through the use of additional plugins.

Weston is used in a variety of systems, including embedded devices, desktop environments, and virtual machines, and it is supported by a number of Linux distributions and other operating systems. It is licensed under the MIT License, which allows for free use, modification, and distribution of the software.

Chapter 10: Applications Installation on Yocto Image

In this chapter, we will showcase the installation procedures of the previously stated applications on the customized Linux image created by Yocto.

Weston Compositor is used instead of the X11 display server to open all the applications and it is like a Desktop GUI where all the applications will be opened above it.

Each application will follow the same procedure to be installed which is like the following:

1. Prepare the application files including source codes, header files, video files, …etc.

2. Get the required dependencies for each application to run including Python Packages, C++ Libraries or any other package needed by the application to run.

3. Check whether these dependencies are compatible with the currently working Yocto branch or not. If it is not compatible with the branch, then you will have to write the recipe for that dependency for the application to work.

4. Decide whether you want to upload the application to a GitHub repository or save the files locally for the recipe to read it.

5. Write a recipe for each application depending on the used programming language to create the application (Python, C++, or Qt C++) including the dependencies which is required to be installed on the system for the application to run.

6. Add this variable to the image recipe: IMAGE_INSTALL += “application_name”.

7. Bitbake the image recipe and the application will be installed on the system depending on the configurations provided in the recipe.

10.1 Qt GUI

The GUI source code with its related files is uploaded to a GitHub repository. The GUI’s Makefile is generated either from the CMakeLists.txt or the (.pro) file which contains all the required configurations for the Qt Project to work properly. The (.pro) file is uploaded with the GUI source code to GitHub.

10.1.1 Dependencies

This application depends on many libraries that must be compiled and installed on the system for the application to run properly.

The libraries used are: Qtbase, Qtdeclarative, Qtimageformats, Qtmultimedia, Qtquickcontrols, Qtquickcontrols2, Qtbase-plugins, Qtwayland, Liberation-fonts openssl.

These libraries are found in “meta-qt5” layer which is added in the BBLAYERS file in our project. Each library has its own recipe included in the layer. There is no need to make a custom-made recipe for a non-existent library as all the required libraries are provided by the layer. OpenSSL is required to run the APIs which are used in the GUI.

The application dependencies are put in DEPENDS or RDEPENDS variables. DEPENDS variable means that any package included in this variable is required to build the application and RDEPENDS variable means that any package included is required during the run time of the application.

10.1.2 The Recipe

The recipe used to install the GUI will be like this:

```
SUMMARY = "A recipe for The Qt GUI Application"
LICENSE = "CLOSED"
FILESEXTRAPATHS:prepend := "${THISDIR}/systemd:"
RDEPENDS:${PN}:append = " qtbase-tools qtbase qtdeclarative qtimageformats
qtmultimedia qtquickcontrols2 qtquickcontrols qtbase-plugins liberation-
fonts qtwayland openssl"
DEPENDS:append = " qtbase"
SRC_URI:append = " file://CarDashboard.service \
                   file://CarDashboard.env"
SRC_URI:append                              =                                  "
git://github.com/Hussam82/CarDashboard;protocol=https;branch=master"
SRCREV = "afc42e58e44d7c8414f5960e0ccfba45f7659443"

S = "${WORKDIR}/git"

inherit systemd
inherit qmake5

do_install:append(){
install -d ${D}${bindir}
install -m 0755 CarDashboard ${D}${bindir}

install -d ${D}${systemd_system_unitdir}
install -m 0644 ../CarDashboard.service ${D}${systemd_system_unitdir}
```

```
install -d ${D}/etc
install -m 0644 ../CarDashboard.env ${D}/etc

}
SYSTEMD_SERVICE:${PN} = "CarDashboard.service"
SYSTEMD_AUTO_ENABLE:${PN} = "enable"


FILES:${PN}:append = " ${systemd_system_unitdir}/CarDashboard.service"
FILES:${PN}:append = " /etc/CarDashboard.env"
FILES:${PN}:append = " ${bindir}/CarDashboard"
```

Systemd and qmake classes are inherited in this recipe to tell bitbake that this project uses QMake to run the (.pro) file and generate its Makefile while systemd is used to make the application run on startup by using the service file (qtapp.service) through these two variables SYSTEMD_AUTO_ENABLE and SYSTEMD_SERVICE:${PN}.

The used service is like the following:

```
[Unit]
Description=My Qt5 App
After=multi-user.target local-fs.target weston.service
Requires=weston.service

[Service]
User=root
Restart=on-failure
Type=oneshot
EnvironmentFile=/etc/CarDashboard.env
ExecStart=/usr/bin/CarDashboard
StandardOutput=console

[Install]
WantedBy=multi-user.target weston.service
```

- Requires variable contains the service which is to be started before this service starts which is the Weston service.
- User variable contains the user owner of this service file.
- Restart variable contains the condition on which the service restarts.
- EnvironmentFile variable contains the environment file directory which has all the environments variables required to be exported for the application to work properly.
- ExecStart variable contains the directory the binary file of the application to be executed.

The rest of the variables will be discussed later.

The used Environment File contains some environment variables that have to be exported during the startup of the GUI App:

```
QT_QPA_PLATFORM=wayland
XDG_RUNTIME_DIR=/run/user/1000
WAYLAND_DISPLAY=wayland-1
```

- QT_QPA_PLATFORM is used by the Qt application framework to specify the platform plugin. The platform plugin is responsible for providing the Qt application with access to the underlying windowing system, input devices, and other platform-specific features.
- XDG_RUNTIME_DIR specifies the location of the runtime directory for the current user. The runtime directory is a temporary directory that is created when the user logs in and is used to store user-specific runtime files and other data that is required by running applications.
- WAYLAND_DISPLAY specifies the name of the Wayland display socket that an application should use to connect to the Wayland compositor. Wayland is a display server protocol that is used on modern Linux systems as an alternative to the older X Window System.

The GitHub repository link is added to the SRC_URI variable specifying the branch name and protocol used.

The files downloaded from the repository will have sha256sum which will be provided in the variable SRC_URI[sha256sum].

SRCREV variable takes the commit ID, and it is obligatory to provide it to specify at which state the repository will be downloaded.

LICENSE variable takes the License used in the application and if there is none, CLOSED is used and if there is a License for example MIT, the license file must be provided with its checksum.

These two commands installs the binary produced from the compilation process of the GUI.

```
install -d ${D}${bindir}
install -m 0755 CarDashboard ${D}${bindir}
```

The first line "install -d ${D}${bindir}" creates a new directory in the destination (D) directory that corresponds to the system's binary directory (bindir). The ${D} variable is used to specify the root of the target filesystem that the files will be installed into. The "-d" option tells the "install" command to create intermediate directories if they don't exist.

The second line "install -m 0755 CarDashboard ${D}${bindir}" installs the executable file "CarDashboard" into the specified binary directory with the permission mode "0755", which allows the owner to read, write, and execute the file, and other users to execute it. The "-m" option specifies the file permission mode to be set during installation.

The same procedure is used with the service file and the environment file to be installed in the specified directories.

QMake uses the .pro file and generated a Makefile which is used to compile the application producing the executable which will be installed in the specified directory.

10.2 Lane Detection

10.2.1 Dependencies

The Lane Detection application is a C++ code that uses opencv to detect lanes. It depends on opencv and some libraries that are used by the application in run time.

These libraries are opencv libopencv-core libopencv-imgproc libopencv-highgui libopencv-videoio libopencv-imgcodecs.

They are provided by "meta-oe" layer with a version of 4.5.5 in kirkstone branch. There was no need to make any custom-made recipes for the needed libraries as they are all provided by meta-oe.

10.2.2 The Recipe

The files of the application are stored locally in the files directory inside the recipe folder.

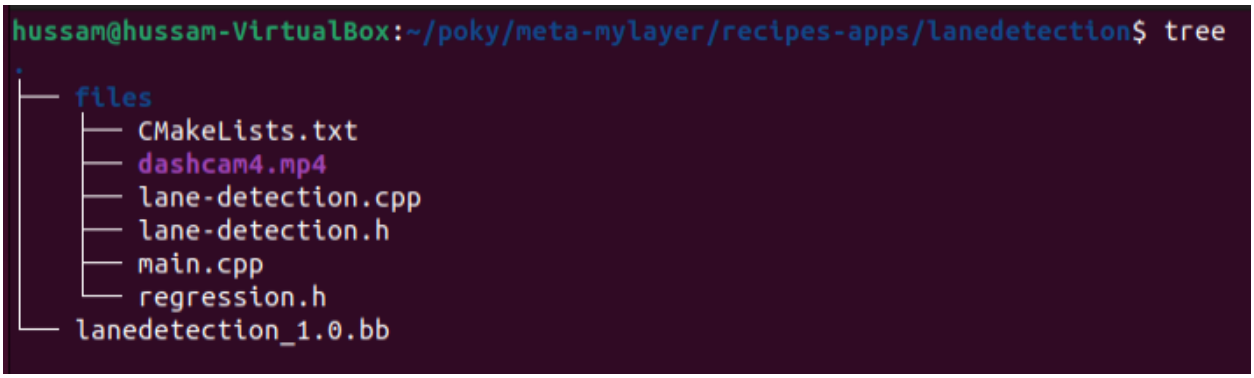

Figure 156 Files inside lanedetection recipe folder

The application source code files are lane-detection.cpp, main.cpp, regression.h and lane-detection.h. dashcam4.mp4 is the video which will be used to display the detections on it. CMakeLists.txt file will contain the commands which tell bitbake to compile and install the executable in the specified directory.

The CMakeLists.txt file contains the following:

```
cmake_minimum_required(VERSION 3.15)
project(lanedetection)

set(CMAKE_CXX_STANDARD 17)
```

```
find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} "main.cpp" "lane-detection.cpp")
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
install(TARGETS ${PROJECT_NAME} DESTINATION "/usr/bin/")
install(FILES dashcam4.mp4 DESTINATION "/etc/")
```

Overall, the CMakeLists.txt file sets up the C++ project for lane detection, finds and includes the required OpenCV library, creates an executable target and links the OpenCV library to it, and installs the resulting executable and a video file to specific directories on the system.

The recipe will be like the following:

```
DESCRIPTION   = " This is The Lane Detection Application Recipe"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://main.cpp \
       file://lane-detection.cpp \
       file://lane-detection.h \
       file://regression.h \
       file://CMakeLists.txt \
       file://dashcam4.mp4 "



DEPENDS = "opencv"
RDEPENDS_${PN} = "libopencv-core libopencv-imgproc libopencv-highgui libopencv-
videoio libopencv-imgcodecs"
S = "${WORKDIR}"

inherit cmake


FILES_${PN} = "/etc/Models/*"
```

10.3 Vehicles Detection

This application uses OpenCV and TensorFlow Light model to detect vehicles. It depends only on the tensorflow-lite Python package, and it is provided in the meta-tensorflow-lite layer. The recipe which is used to install the application into the system is like the following:

```
SUMMARY = "A recipe to build and install vehicle detection application"
LICENSE = "CLOSED"
RDEPENDS:${PN}:append = " python3-tensorflow-lite python3-opencv"
DEPENDS:append = " opencv"
SRC_URI:append = " file://vehicledetection.py \
                   file://videoplayback_2.mp4 \
                   file://custom_model_lite"

S = "${WORKDIR}"


do_install:append(){
install -d ${D/etc/Models
cp ${S}/vehicledetection.py ${D/etc/Models

install -d ${D}/etc/VehicleDetection
cp -r ${S}/custom_model_lite ${D}/etc/VehicleDetection
cp ${S}/videoplayback_2.mp4 ${D}/etc/VehicleDetection

}
FILES:${PN}:append = " /etc/Models/*"
FILES:${PN}:append = " /etc/VehicleDetection/*"
```

This recipe simply copies the Python code into the “/usr/bin” directory and the rest of the files used by the code into the “/etc/VehicleDetection” directory.

10.4 Yolov5 Applications

10.4.1 Dependencies

In the upcoming applications, they are all treated in the same way as they all use OpenCV and Yolov5.

They depend on many Python packages such as numpy, pillow, matplotlib, psutils, pyyaml, requests and tqdm. These Python packages are provided by OpenEmbedded-Core layer, but Yolov5 depends on more Python packages that are not provided by any meta layer. Therefore, we had to do our own recipes to install these Python packages such as scipy, ultralytics, PyTorch, torchvision and huggingface-hub.

Also, we had to make a recipe that installs the Yolov5 package into the directory where the Python packages are installed which is “/usr/lib/python3.10/site-packages”.

There are two ways to install a Python package through a recipe either by downloading the source code and installing it with setup.py or by downloading the already built package in the form of a (.whl) file which is like a zipped file. The (.whl) file contains the package file which are ready to be installed in the Python packages directory directly without the need for building the package.

The recipe for Yolov5 package will be like the following:

```
LICENSE = "GPL"
LIC_FILES_CHKSUM            =            "file://${S}/yolov5-7.0.12.dist-
info/LICENSE;md5=97ff9683aa36f333c7d2295d6520090f"

PYTHON_PACKAGE = "yolov5-7.0.12-py37.py38.py39.py310-none-any.whl"

SRC_URI                                                                 =
"https://files.pythonhosted.org/packages/ba/6f/52449e79074fc4a319ec384e02b
fbb812e3d5ca0a14a5ebc6c2dd04aa089/${PYTHON_PACKAGE};subdir=${BP}"
SRC_URI[md5sum] = "cc965558e2c0d929810a608272c466c6"


DEPENDS = "python3 python3-pip-native python3-wheel-native"

RDEPENDS:${PN} = "${PYTHON_PN} \
                  ${PYTHON_PN}-numpy \
                  ${PYTHON_PN}-pillow \
                  ${PYTHON_PN}-matplotlib \
                  ${PYTHON_PN}-psutil \
                  ${PYTHON_PN}-pyyaml \
                  ${PYTHON_PN}-scipy \
                  ${PYTHON_PN}-requests \
                  ${PYTHON_PN}-tqdm \
                  pip-ultralytics \
                  pip-torch \
                  pip-huggingface-hub \
                  pip-torchvision"


RPROVIDES:${PN} = "/bin/bash"
inherit python3native

do_unpack[depends] += "unzip-native:do_populate_sysroot"

do_unpack_extra(){
    [ -d ${S} ] || mkdir -p ${S}
    cd ${S}
    unzip -q -o ${S}/${PYTHON_PACKAGE} -d ${S}
}
```

```
addtask unpack_extra after do_unpack before do_patch

do_install() {
    # Install pip package
    install -d ${D}/${PYTHON_SITEPACKAGES_DIR}
    cp -r ${S}/yolov5 ${D}/${PYTHON_SITEPACKAGES_DIR}
    cp -r ${S}/yolov5/models ${D}/${PYTHON_SITEPACKAGES_DIR}
    cp -r ${S}/yolov5-7.0.12.dist-info ${D}/${PYTHON_SITEPACKAGES_DIR}

}


FILES:${PN} += "\
    ${libdir}/${PYTHON_DIR}/site-packages/* \
"

INSANE_SKIP:${PN} += "already-stripped"

COMPATIBLE_MACHINE = "(-)"
COMPATIBLE_MACHINE:aarch64 = "(.*)"
```

As we can see in the above recipe, the Yolov5 package is in the form of a .whl file, so all we must do is just copy the contents into the packages directory which is the variable PYTHON_SITEPACKAGES_DIR.

As for the rest of the Python packages, a recipe for one of them is called ultralytics which uses the source code is shown next:

```
SUMMARY = "Recipe to embedded the Python PiP Package ultralytics"
HOMEPAGE ="https://pypi.org/project/ultralytics"
LICENSE = "AGPL-3.0"
LIC_FILES_CHKSUM = "file://LICENSE;md5=eb1e647870add0502f8f010b19de32af"

inherit pypi setuptools3
PYPI_PACKAGE = "ultralytics"
SRC_URI[md5sum] = "2f55967df7645bd153f6ea9f31ed0d6b"
SRC_URI[sha256sum]                                                    =
"bedf43e38e0baa99f4283e157174ab2af58faeeaf0566e1cf241575afd2c7ee5"
```

This recipe downloads the source code which has the setup.py file which builds and installs the Python package like using pip3 command. This happens in Yocto by inheriting pypi and setuptools3 classes which are responsible for installing the Python package after downloading the source code.

10.4.2 Pedestrian Detection

The recipe which installs this application into the system is shown next:

```
SUMMARY = "A recipe to build and install pedestrian detection application"
LICENSE = "CLOSED"
RDEPENDS:${PN}:append = " python3-opencv pip-yolov5"
DEPENDS:append = " opencv"
SRC_URI:append = " file://pedestrian.py \
                   file://best_ped.pt \
                   file://cross.mp4"

S = "${WORKDIR}"


do_install:append(){
install -d ${D/etc/Models
cp ${S}/pedestrian.py ${D/etc/Models

install -d ${D}/etc/PedestrianDetection
cp -r ${S}/best_ped.pt ${D}/etc/PedestrianDetection
cp ${S}/cross.mp4 ${D}/etc/PedestrianDetection

}
FILES:${PN}:append = " /etc/Models/*"
FILES:${PN}:append = " /etc/PedestrianDetection/*"
```

This recipe simply copies the Python code into the “/etc/Models” directory and the rest of the files used by the code into the “/etc/PedestrianDetection” directory.

10.4.3 Traffic Lights Detection

The recipe which installs this application into the system is shown next:

```
SUMMARY = "A recipe to build and install traffic lights detection
application"
LICENSE = "CLOSED"
RDEPENDS:${PN}:append = " python3-opencv pip-yolov5"
DEPENDS:append = " opencv"
SRC_URI:append = " file://trafficlights.py \
                   file://best_tl2.pt \
                   file://videoplayback_6.mp4"

S = "${WORKDIR}"
do_install:append(){
install -d ${D}/etc/Models
cp ${S}/trafficlights.py ${D}/etc/Models
```

```
install -d ${D}/etc/TrafficLightsDetection
cp -r ${S}/best_tl2.pt ${D}/etc/TrafficLightsDetection
cp ${S}/videoplayback_6.mp4 ${D}/etc/TrafficLightsDetection

}
FILES:${PN}:append = " /etc/Models/*"
FILES:${PN}:append = " /etc/TrafficLightsDetection/*"
```

This recipe simply copies the Python code into the "/etc/Models" directory and the rest of the files used by the code into the "/etc/TrafficLightsDetection" directory.

10.4.4 Traffic Signs Detection

The recipe which installs this application into the system is shown next:

```
SUMMARY = "A recipe to build and install traffic signs detection application"
LICENSE = "CLOSED"
RDEPENDS:${PN}:append = " python3-opencv pip-yolov5"
DEPENDS:append = " opencv"
SRC_URI:append = " file://trafficsigns.py \
                   file://best_ts.pt \
                   file://videoplayback_5_Trim.mp4"

S = "${WORKDIR}"


do_install:append(){
install -d ${D}/etc/Models
cp ${S}/trafficsigns.py ${D}/etc/Models

install -d ${D}/etc/TrafficSignsDetection
cp -r ${S}/best_ts.pt ${D}/etc/TrafficSignsDetection
cp ${S}/videoplayback_5_Trim.mp4 ${D}/etc/TrafficSignsDetection
}
FILES:${PN}:append = " /etc/Models/*"
FILES:${PN}:append = " /etc/TrafficSignsDetection/*"
```

This recipe simply copies the Python code into the "/etc/Models" directory and the rest of the files used by the code into the "/etc/TrafficSignsDetection" directory.

10.5 The Final Image Recipe

The image recipe will contain all the installed packages, libraries, and applications. It is shown next:

```
#This image is based on core-image-base
include recipes-core/images/core-image-base.bb

# The resulting SDK will include the required development tools for cross-
compiling a Qt application
inherit populate_sdk_qt5

# Only produce the "rpi-sdimg" image format
IMAGE_FSTYPES = "rpi-sdimg"

#Add support for ssh and tools-sdk you will find the rest in core-image
class
IMAGE_FEATURES += "ssh-server-dropbear"

#Add python3 and its packages
IMAGE_INSTALL += "python3 python3-pip python3-modules"

#Add nano and git
IMAGE_INSTALL += "nano git"

#Add our apps
IMAGE_INSTALL += "cardashboard"
IMAGE_INSTALL += "lanedetection"
IMAGE_INSTALL += "trafficsigns"
IMAGE_INSTALL += "trafficlights"
IMAGE_INSTALL += "vehicledetection"
IMAGE_INSTALL += "pedestrian"


#Add support for  OpenCV and Camera
IMAGE_INSTALL += "ffmpeg"
IMAGE_INSTALL += "gstreamer1.0 gstreamer1.0-libav gstreamer1.0-plugins-base
gstreamer1.0-meta-base  gstreamer1.0-plugins-bad  gstreamer1.0-plugins-ugly
gstreamer1.0-plugins-good"
IMAGE_INSTALL += "fswebcam"
IMAGE_INSTALL += "v4l-utils"
IMAGE_INSTALL += "userland"
IMAGE_INSTALL += "weston weston-init"
MACHINE_FEATURES:append = " xf86-video-fbdev"

#Add support for wifi and bluetooth
```

```
IMAGE_INSTALL += "linux-firmware-bcm43430"

#Use network manager
IMAGE_INSTALL     +=     "networkmanager     networkmanager-bash-completion
networkmanager-nmtui"

#Add kernel modules
IMAGE_INSTALL += "dht11km"

# Remove old builds
RM_OLD_IMAGE = "1"

# Customize the splash screen or disable
IMAGE_FEATURES:remove = "splash"
```

The image will be based on core-image-base which is provided by the OpenEmbedded-Core layer.

0*

Chapter 11: Hardware

11.1 Raspberry Pi

Raspberry Pi is a series of small single-board computers (SBCs) developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom.

The Raspberry Pi is a very cheap computer that runs Linux, but it also provides a set of GPIO (general purpose input/output) pins, allowing you to control electronic components for physical computing and explore the Internet of Things (IoT).

Figure 157 Raspberry Pi board

The Raspberry Pi 4 Model B, which is the one we're using, is the latest version of the low-cost Raspberry Pi computer released in Jun 2019. Raspberry Pi isn't like your typical device, in its cheapest form it doesn't have a case, and is simply a credit-card sized electronic board of the type you might find inside a PC or laptop, but much smaller. It offers ground-breaking increases in processor speed, multimedia performance, memory, and connectivity compared to the prior-generation Raspberry Pi 3 Model B+, while retaining backwards compatibility and similar

power consumption. For the end user, Raspberry Pi 4 Model B provides desktop performance comparable to entry-level x86 PC systems.

There are a lot of things we can do with Raspberry Pi, some people buy a Raspberry Pi to learn to code, and people who can already code use the Pi to learn to code electronics for physical projects. The Raspberry Pi can open opportunities for you to create your own home automation projects, which is popular among people in the open-source community because it puts you in control, rather than using a proprietary closed system. In our case, Raspberry Pi will be used to run the applications mentioned in the previous chapters, that being object detection, lane detection, sign detection and finally the graphical user interface (GUI) controlling all said applications.

11.1.1 Raspberry Pi 4 Specifications

Table 4 Raspberry Pi 4 Specs

Processor	Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
Memory	1GB, 2GB or 4GB LPDDR4 (depending on model)
Connectivity	2.4 GHz and 5.0 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 5.0, BLE Gigabit Ethernet 2 × USB 3.0 ports 2 × USB 2.0 ports.
GPIO	Standard 40-pin GPIO header (fully backwards-compatible with previous boards)
Video and sound	2 × micro HDMI ports (up to 4Kp60 supported) 2-lane MIPI DSI display port 2-lane MIPI CSI camera port 4-pole stereo audio and composite video port
Multimedia	H.265 (4Kp60 decode); H.264 (1080p60 decode, 1080p30 encode); OpenGL ES, 3.0 graphics
SD card support	Micro SD card slot for loading operating system and data storage
Input power	5V DC via USB-C connector (minimum 3A1) 5V DC via GPIO header (minimum 3A1) Power over Ethernet (PoE)–enabled (requires separate PoE HAT)
Environment	Operating temperature 0–50ºC
Compliance	For a full list of local and regional product approvals, please visit

11.1.2 Raspberry Pi 4 Booting Sequence

The booting sequence starts with GPU until the kernel is loaded in RAM and then CPU takes over.

It consists of 5 main stages:

- First Stage Bootloader (ROM Bootloader or RBL).
- Second Stage Bootloader.
- Start.elf.
- Linux Kernel.
- Root File System and init Process.

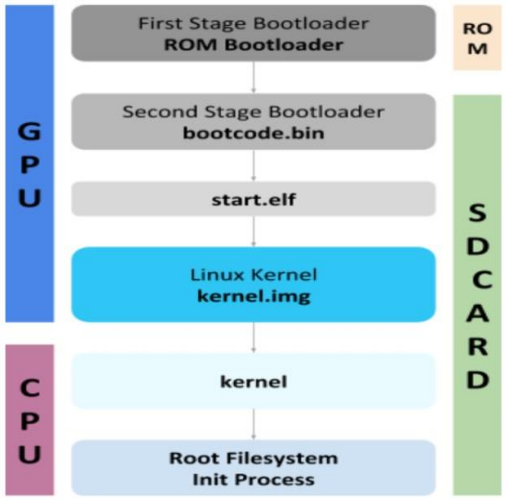

Figure 158 Raspberry Pi 4 Booting Sequence

The booting sequence runs as following:

1. When the system powers on, the ARM CPU is off, the GPU is powered up and starts the booting sequence.
2. The First Stage Bootloader starts executing from the GPU ROM (Boot ROM).
    a. Performs some integrity checks on H.W.
    b. Searches for the bootable device whether it is SD Card, USB or EMMC (Embedded Memory Card).
    c. Loads the Second Stage Bootloader (bootcode.bin).
3. The Second Stage Bootloader starts, and it is responsible for:
    a. Enabling the SDRAM.
    b. Initializing some peripherals.
    c. Loading The Third Stage Bootloader (start.elf).
4. The Third Stage Bootloader (start.elf) loads The Linux Kernel into RAM and reads these files:
    a. config.txt
    b. cmdline.txt
    c. Device Tree Binary File (.dtb)

These files are essential for running the Kernel.

5. By now, the kernel is loaded into RAM and CPU takes over and starts executing The Linux Kernel.
6. The Linux Kernel starts executing until it reaches The Init Process.

11.1.3 Raspberry Pi image structure

The SD Card which is inserted into RPi is divided into two partitions:

- Boot Partition, which is in the format of FAT32, and it contains all the files needed in the booting sequence.
- Rootfs Partition, which is in the format of EXT4, and this is where The Root File System is located.

BOOT	RASPBERRY (rootfs)		
FAT32	EXT4		
• bootcode.bin • start.elf • kernel7.img • fixup.dat • config.txt • cmdline.txt • *.dtb	• /bin • /boot • /dev • /etc • /home • /lib	• /media • /mnt • /opt • /proc • /root • /run	• /sbin • /srv • /sys • /tmp • /usr • /var

Figure 159 Image file structure after burning it on the SD Card

11.2 DHT-11 Sensor

In our project, we will be using DHT-11 Sensor which is a low-cost, digital temperature and humidity sensor. It is a popular sensor used in various projects and applications, including home automation, weather stations, and HVAC systems. The DHT11 sensor uses a capacitive humidity sensor and a thermistor to measure humidity and temperature, respectively. It has a measurement range of 0-99% relative humidity (RH) and a temperature range of 0-50°C. The DHT11 sensor communicates with a microcontroller or other digital device using a single-wire serial interface, and it is powered by a 3-5V DC power supply. The sensor's data output is a 40-bit digital signal that includes both humidity and temperature readings. The DHT11 sensor is a cost-effective solution for measuring temperature and humidity in various applications.

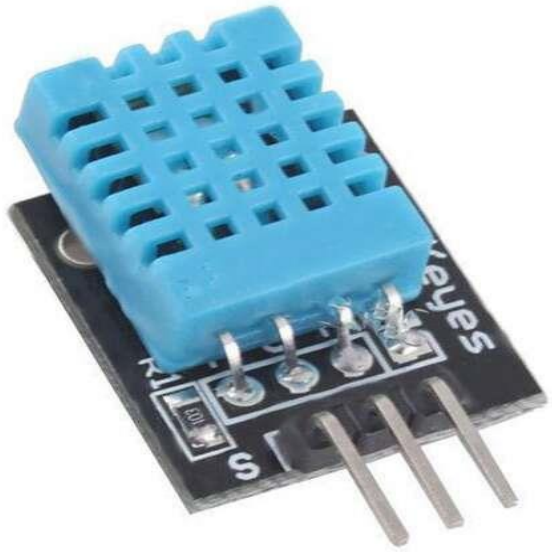

Figure 160 DHT-11 Sensor

11.2.1 DHT-11 Sensor Pinout

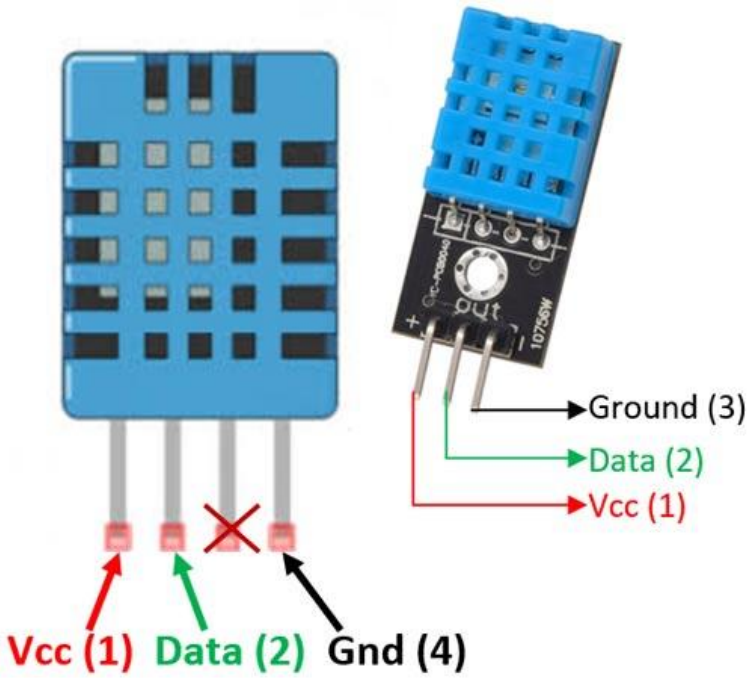

Figure 161 DHT-11 Sensor Pinout

Table 5 DHT-11 Pinout Configuration

Pin No.	Pin Name	Description
DHT-11 Sensor		
1	Vcc	Power supply 3.5V to 5.5V
2	Data	Outputs both Temperature and Humidity through serial Data
3	NC	No Connection and hence not used
4	GND	Connected to the ground of the circuit
DHT-11 Sensor Module		
1	Vcc	Power supply 3.5V to 5.5V
2	Data	Outputs both Temperature and Humidity through serial Data
3	GND	Connected to the ground of the circuit

11.2.2 DHT-11 Sensor Specifications

- Operating Voltage: 3.5V to 5.5V
- Operating current: 0.3mA (measuring) 60uA (standby)
- Output: Serial data
- Temperature Range: 0°C to 50°C
- Humidity Range: 20% to 90%
- Resolution: Temperature and Humidity both are 16-bit
- Accuracy: ±1°C and ±1%

11.2.3 DHT-11 Sensor Connection with RPi4

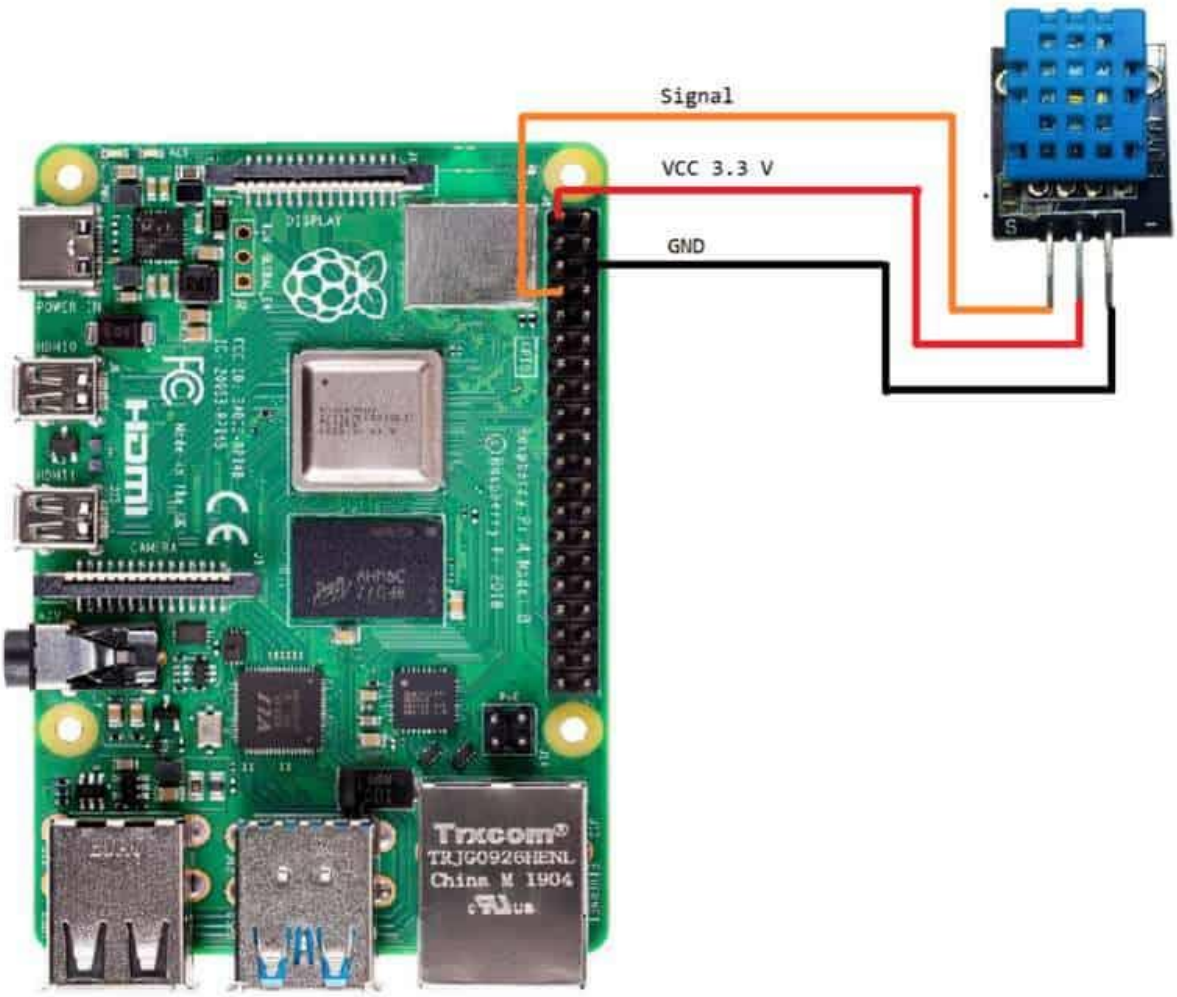

Figure 162 DHT-11 Connection with RPi4

Here the sensor is connected to Pin7 which is GPIO4 of the Raspberry Pi.

11.2.3 DHT-11 Sensor Working

The data pin of the DHT11 module will provide output in the form of an 8-bit integer for humidity data, an 8-bit decimal for humidity, an 8-bit integer for temperature data, an 8-bit fractional temperature data, and an 8-bit parity bit. To initiate the transmission of this data, the I/O pin must be briefly set to low and then held high, as illustrated in the timing diagram below.

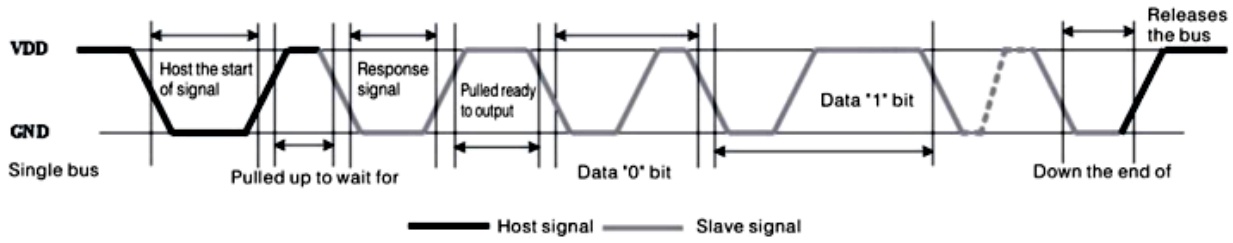

Figure 163 DHT-11 Frame

These timings are explained step by step in the datasheet.

11.3 Touchscreen LCD

Figure 164 7 Inch Touchscreen LCD

11.3.1 Overview

The Waveshare 7 inch touchscreen LCD screen is a high-quality display screen designed for use with various devices such as single-board computers, microcontrollers, and other embedded systems. It features a resistive touch screen that allows for easy and precise input, and it has a resolution of 1024x600 pixels, which provides sharp and clear images.

11.3.2 Features

The Waveshare 7 inch touchscreen LCD screen has several features that make it a great choice for many applications. Some of these features include:

- Resistive Touch Screen: The resistive touch screen allows for easy and precise input, even with a gloved hand or stylus.
- High resolution: The 1024x600 pixel resolution provides clear and sharp images.
- Wide Viewing Angle: The wide viewing angle of the display screen ensures that the images are visible from various angles.
- Easy to Use: The screen is easy to use and can be easily connected to various devices using the provided cables.
- Compatible with Various Devices: The screen is compatible with various devices such as single-board computers, microcontrollers, and other embedded systems.
- Low Power Consumption: The screen has a low power consumption of only 0.7W, making it ideal for battery-powered applications.
- Integrated Drivers: The screen comes with integrated drivers, which simplify the setup process and make it easy to use.

11.3.3 Setup Instructions

To set up the Waveshare 7 inch touchscreen LCD screen, follow the instructions below:

- Connect the HDMI cable to the HDMI port on the screen and the other end to the HDMI port on your device.
- Connect the USB cable to the USB port on the screen and the other end to a USB port on your device.
- Turn on your device and the screen should display the output.

Figure 165 Touchscreen connection with Raspberry Pi 4

Chapter 12: Results and Conclusions

In conclusion, this project provided an effective ADAS to improve the driving safety and reduce the risk of accidents. Through the previous chapters, we started exploring the making of the system from the very bottom layers by the means of Embedded Linux through tools like Yocto to tailor an operating system that would fit the requirements of our system, no more, no less.

In addition, we developed and integrated our own set of ADAS features into the system, starting with the graphical user interface or the GUI which would act as the main interface between the user and the system represented in the various other features that we implemented. The GUI was designed to be user-friendly and intuitive.

We also integrated advanced features such as drowsiness detection, lane detection, traffic signs detection, traffic lights detection, and pedestrian detection, which all utilize the camera to give live feedback. These features were designed with the intention to provide a seamless and safe driving experience, reducing the risk of accidents and improving overall driving safety.

Throughout the development process, we faced several challenges related to software compatibility, optimization and system integration. However, through diligent testing and refinement, we were able to overcome these challenges and deliver a reliable and effective ADAS system.

Chapter 13: Future Work

Our goal from the very begging was to create a fully industrial system that we could one day use in real life. But the limitations from our hardware and our lake of experience in regards to the industry may have held us back a bit.

Throughout this project, we have learned a lot and achieved so much, but we know that there is always room for improvement and more work to do. So in this section, we introduce some of the future work.

13.1 Upgrade the Hardware

One of our ongoing efforts to improve ADAS is upgrading the hardware used to run these systems. One example of this is the Raspberry Pi 4, which is a popular single-board computer widely used in ADAS development.

While the Raspberry Pi 4 is a compact, affordable, and powerful computer that can run machine learning models for ADAS. However, as the demand for more accurate and complex models increases, the limitations of the Raspberry Pi 4 become more apparent as it does lack a GPU or a graphical processing unit. One of the main limitations is the frame rate per second (FPS) that the Raspberry Pi 4 can handle, which can impact the accuracy and responsiveness of ADAS features.

To address this limitation, upgrading the hardware used in ADAS development would seem like the optimal decision. And by upgrading the hardware, we mean either of the following:

1. Add a TPU (Tensor Processing Unit)
2. Use a more powerful platform (such as the Jetson Nano).

13.1.1 Add a TPU

A TPU is a specialized hardware accelerator designed to perform high-speed matrix operations, which are common in machine learning models. By adding a TPU to an ADAS development system, it can significantly increase the speed and efficiency of running complex machine learning models. This, in turn, can improve the accuracy and responsiveness of ADAS features and overall net more frame per second.

13.1.2 Use a different Developer Kit

Alternatively, using a dedicated platform such as the Jetson Nano can also provide significant benefits. The Jetson Nano is another compact, low-power, and affordable platform designed specifically for running machine learning models in ADAS applications. With its powerful GPU, which our Raspberry lacks, and specialized hardware accelerators, the Jetson Nano can handle multiple camera inputs and run complex models with high efficiency and accuracy.

Figure 166 Jetson Nano Board

13.2 Increase the accuracy of the Models

13.2.1 Use more accurate models

As we said in the previous section, upgrading the hardware would allow us to run models that we would otherwise not be able to. Our hardware of choice has put many restrictions on us to balance between the accuracy and the speed and it might have tipped the scales in favor of the speed to get the models going.

13.2.2 Add more Cameras

Another approach to increasing the accuracy of ADAS is to add more cameras to the system. By adding more cameras and using image fusion techniques, the system can capture a more comprehensive view of the road and surrounding environment, enabling it to detect potential hazards with greater accuracy. For example, some ADAS systems use a multi-camera setup to provide a 360-degree view of the vehicle, which can improve the accuracy of features such as lane departure warnings and blind spot detection.

Adding sensors to an ADAS system can also help increase its accuracy. Sensors provide additional data to the system, enabling it to better detect and respond to potential hazards on the road.

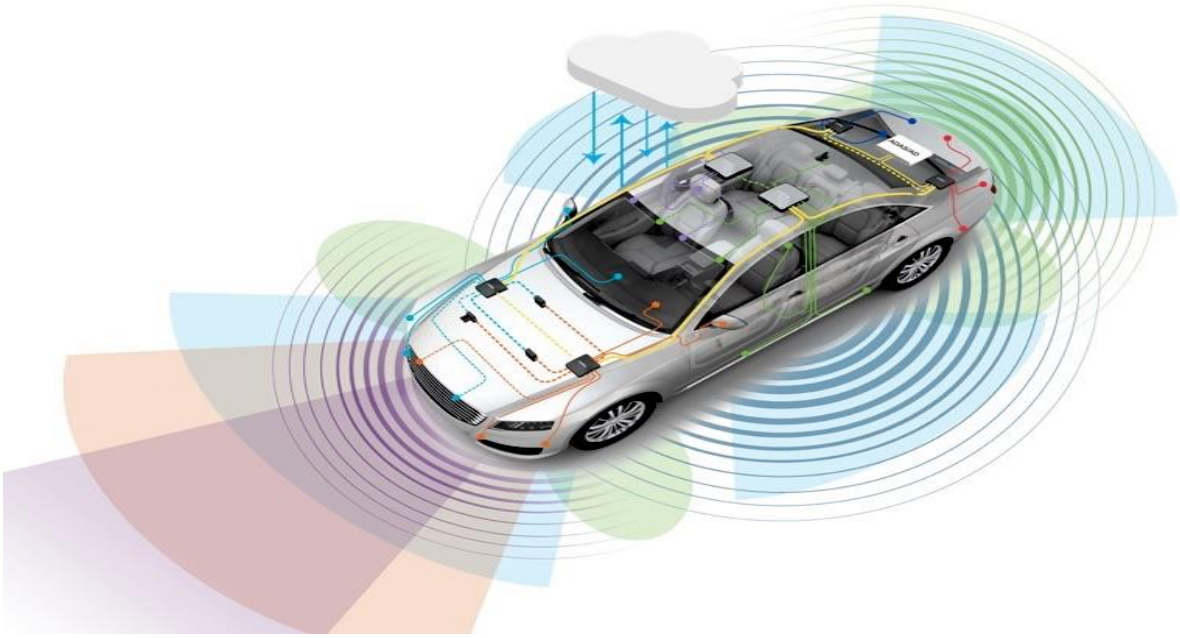

Figure 167 Vehicle Surroundings

13.2.3 Add Sensors

One example of a sensor commonly used in ADAS is radar. Radar sensors can detect the distance and speed of objects in front of the vehicle, enabling features such as adaptive cruise control and automatic emergency braking. By adding radar sensors to an ADAS system, the system can better detect and respond to potential collisions.

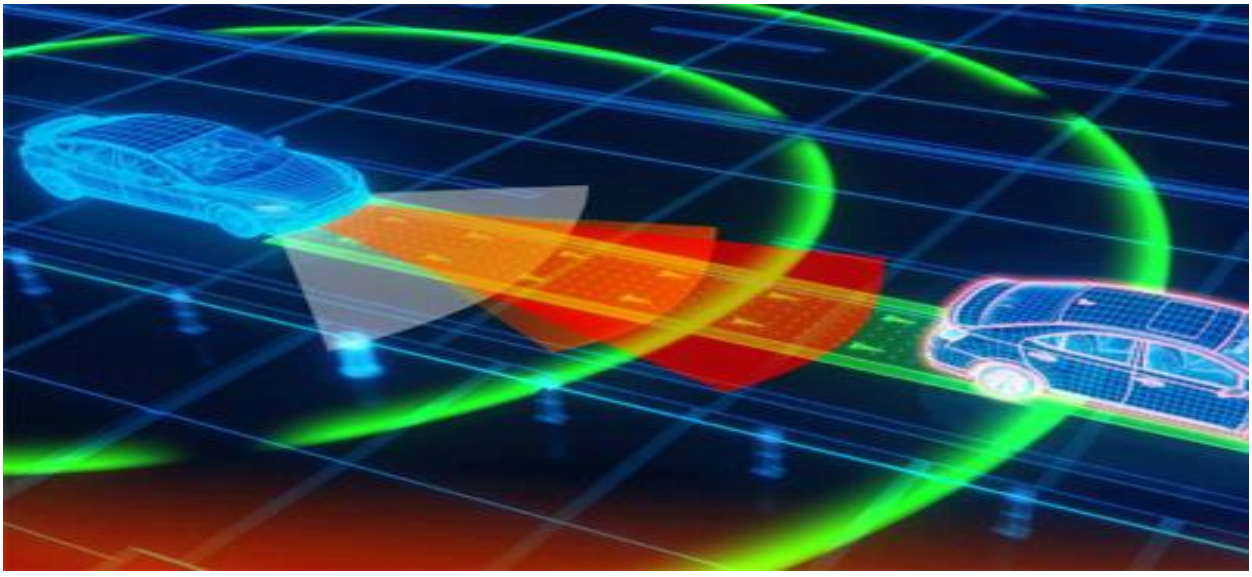

Figure 168 Radar Sensing

Another example of a sensor used in ADAS is lidar. Lidar sensors use laser beams to create a 3D map of the surrounding environment, enabling the system to detect objects with greater accuracy and precision. By adding lidar sensors to an ADAS system, the system can better detect and respond to potential hazards, including pedestrians and cyclists.

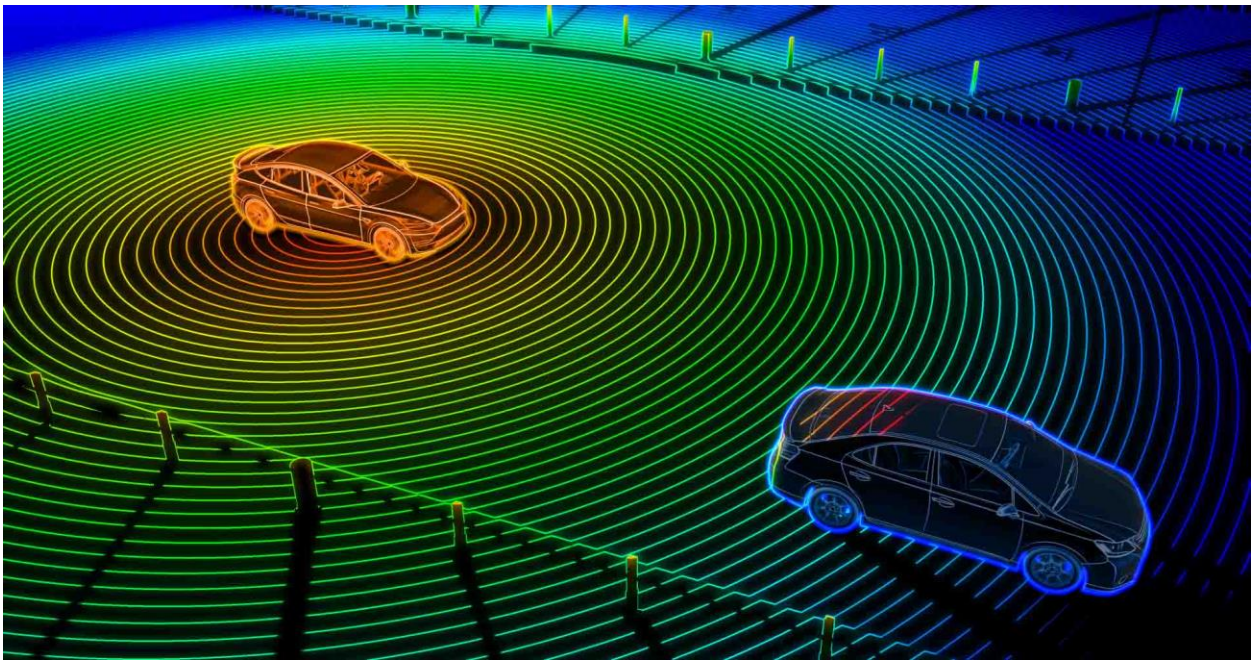

Figure 169 Lidar Sensing

References

[1] A. Vaduva, Learning Embedded Linux Using the Yocto Project, Packt Publishing Ltd, 2015.

[2] E. a. G. H. Upton, Raspberry Pi user guide, John Wiley & Sons, 2016.

[3] C. Simmonds, Mastering embedded Linux programming., 2015: Packt Publishing Ltd,.

[4] O. a. D. A. Salvador, Embedded Linux Development with Yocto Project, Packt Publishing Ltd, 2014.

[5] O. a. D. A. Salvador, Embedded Linux Development using Yocto Projects: Learn to leverage the power of Yocto Project to build efficient Linux-based products, Packt Publishing Ltd, 2017.

[6] M. a. S. W. Richardson, Getting started with raspberry PI., 2012: O'Reilly Media, Inc.

[7] P. A. L. a. S. N. Raghavan, Embedded Linux system design and development., 2005: CRC press,.

[8] D. Molloy, Exploring Raspberry Pi: interfacing to the real world with embedded Linux., John Wiley & Sons,, 2016.

[9] J. Lombardo, Embedded linux, Sams Publishing, 2001.

[10] R. Klette, Computer Vision for Driver, Auckland, New Zealand: © Springer International Publishing AG, 2018.

[11] A. a. M. M. S. Kadav, Understanding modern device drivers, ACM SIGPLAN Notices 47.4, 2012.

[12] D. Huong, *Development of Linux Distribution using Yocto Project*, 2022.

[13] B. e. a. Fu, A novel intelligent garbage classification system based on deep learning and an embedded linux system, 2021: IEEE Access 9.

[14] J. A. R. a. G. K.-H. Corbet, Linux device drivers, O'Reilly Media, Inc., 2005.

[15] "THE YOCTO PROJECT. IT'S NOT AN EMBEDDED LINUX DISTRIBUTION,," [Online]. Available: https://www.yoctoproject.org/.

[16] "Welcome to OpenEmbedded, the build framework for embedded Linux.," 2013. [Online]. Available: https://www.openembedded.org/wiki/Main_Page.

[17] A. G. H. M. Z. B. C. D. Kalenichenko, MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, Google Inc., 2017.

[18] C.-Y. T. M.-D. R. G.-Y. S. a. T.-T. L. Yu-Chen Chiu, Mobilenet-SSDv2: An Improved Object Detection Model for Embedded Systems, Department of Electrical and Computer Engineering Tamkang University.

[19] P. S. R. K. P. B. Rakkshab Varadharajan Iyer1, Comparison of YOLOv3, YOLOv5s and MobileNet-SSD V2 for Real-Time Mask Detection, 2021.

[20] A. B. I. T. F. C. A. Bradley*, YOLO-Z: Improving small object detection in YOLOv5 for autonomous vehicles, 2023.

[21] P. B. S, DRIVER DROWSINESS DETECTION, 2022.

[22] M. P. S. *. M. A.-W. a. A. S. Elena MagánORCID, Driver Drowsiness Detection by Applying Deep Learning Techniques to Sequences of Images, Spain: Appl. Sci., 2022.

[23] *Temperature and Humidity Module DHT11 Product Manual*, Aosong Guangzhou Electronics Co., Ltd.

[24] S. J. a. C. J. S. Schmalz, *Using YOLOv5 Object Detection and a Raspberry Pi to Improve the Safety of Drivers,* 2022.

[25] G. e. a. Jocher, *ultralytics/yolov5: v6. 0-YOLOv5n'Nano'models, Roboflow integration, TensorFlow export, OpenCV DNN support*, Zenodo, 2021.

[26] Y. J. T. a. Y. L. An, *A mobilenet SSDLite model with improved FPN for forest fire detection*, Singapore: Springer Nature Singapore, 2022.

[27] P. e. a. Dubey, *Deep Learning-Powered Visual Inspection Using SSD Mobile Net V1 with FPN*, Springer International Publishing, 2022.

[28] M. e. a. Ramzan, *A survey on state-of-the-art drowsiness detection techniques,* IEEE Access 7, 2019.

[29] B. e. a. Alshaqaqi, *Driver drowsiness detection system,* IEEE, 2013.

[30] A. a. S. B. Mohanty, *Drowsiness detection system using KNN and OpenCV*, Springer Singapore, 2021.

[31] L. Z. Eng, *Qt5 C++ GUI Programming Cookbook: Practical recipes for building cross-platform GUI applications, widgets, and animations with Qt 5,* Packt Publishing Ltd,, 2019.

[32] "Qt Documentation," Qt Company, 2023. [Online]. Available: https://doc.qt.io/.