UMinho | 2022

Eduardo Sousa

Custom Automotive Grade Linux image
for Production Diagnostics

Universidade do Minho
Escola de Engenharia

Eduardo João Fernandes de Sousa

Custom Automotive Grade Linux image for Production Diagnostics

julho de 2022

Universidade do Minho
Escola de Engenharia

Eduardo João Fernandes de Sousa

Custom Automotive Grade Linux image for Production Diagnostics

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos

Trabalho efetuado sob a orientação do
Professor Doutor Paulo Cardoso

julho de 2022

Acknowledgements

I would like to thank my advisor, Professor Paulo Cardoso, for helping me all the way through the writing process this dissertation. I would also like to express my gratitude towards Bosch Car Multimedia for taking me in as an intern and for letting me use their offices and technologies.

I would like to give special thanks to my colleagues of Bosch Car Multimedia for teaching me, for their extensive help through the development and spending their time with me when needed. A special thanks to Miguel Azevedo, my project superviser at Bosch Car Multimedia, for passing his knowledge onto me.

A huge thanks to my family for supporting me in every decision along the way, for giving me the tools that made me what I am and for always picking me up when I went down. A very big thank you to my girlfriend for helping me through the tough times this dissertation presented and for always being there for me.

To all sincerely, thank you!

STATEMENT OF INTEGRITY

Resumo

Os sistemas presentes nos automóveis de hoje em dia, têm vindo a tornar-se cada vez mais complexos conforme as tecnologias e as preferências dos consumidores têm evoluído. Os sistemas, tais como consolas centrais ou paineis de instrumentos, têm o papel não só de informar os passageiros como também de entertenimento e ajuda na navegação. Estas e outras capacidades traduziram-se numa maior complexidade destes sistemas, tanto a nível do *software* como do *hardware*.

A maior parte dos fornecedores de peças e equipamentos eletrónicos para a industria automóvel têm a necessidade de acelerar o processo de testes dos seus produtos, para poderem acompanhar a procura mantendo a qualidade. Para este fim, *software* de testes é usado nos sistemas desenvolvidos com o objetivo de testar o *hardware* e *software* desenvolvido para o cliente. No *software* de testes incluí-se o *Production Diagnostic Software (PDS)*.

A solução de *PDS* atualmente utilizada pela Bosch é composta por uma pilha de software, QNX a correr em paralelo com AUTOSAR da Vector. O AUTOSAR é um *Real-Time Operative System (RTOS)* usado como ponto de entrada dos comandos de diagnostico e oferece acesso ao hardware e à interface de comunicação. O QNX é um sistema *UNIX* de alto nível que lida com dispositivos externos, como ecrãs e audio.

O Automotive Grade Linux (AGL), em comparação com outros sistemas operativos direcionados à indústria automóvel, é mais recente e apresenta *software* totalmente *open-source*. Devido a isto, apresenta custos reduzidos quando comparado com soluções closed-source e um maior grau de personalização em termos de código.

Este projeto de dissertação implementou uma prova de conceito de um PDS usando o AGL como o unico sistema operativo, com o objetivo de competir e substituir, totalmente ou parcialmente, a pilha de software existente na solução atual da Bosch.

A conclusão deste projeto de dissertação é que o AGL não consegue substituir a pilha de software devido à falta de qualidades *real-time* por parte do AGL. Mas, por outro lado, é um possível candidato a substituir o QNX na pilha de software, pois o AGL apresenta as mesmas capacidades que o QNX a menor custo e com maior personalização.

Palavras-chave: Automotive, PDS, AGL, Diagnostico

Abstract

The systems in today's cars have become increasingly complex as technology and consumer preferences have evolved. Systems, such as center consoles or instrument clusters, have the role of not only inform passengers but also to entertain and aid navigation. These and other capabilities increased the complexity of these systems, in terms of software and hardware.

Most suppliers of electronic parts and equipment to the automotive industry have the need to speed up the process of testing their products so that they can meet demand while maintaining quality. For this purpose, test software is used with the objective of testing the hardware and software developed for the client. Test software includes Production Diagnostic Software (PDS).

The PDS solution currently used by Bosch is composed of a software stack, QNX running alongside Vectors AUTOSAR. AUTOSAR is a Real-Time Operative System (RTOS) used as the entry point for diagnostic commands and provides access to the hardware and communication interface. QNX is a high-level UNIX system that handles external devices, such as displays and audio.

The Automotive Grade Linux (AGL), compared to other Operative Systems targeted at the automotive industry, is newer and features a open-source approach. Because of this, it offers lower costs when compared to other closed-source solutions and a higher level of customization of code.

This dissertation project implemented a proof-of-concept of a PDS using AGL as the only operating system, with the aim to compete and replace, totally or partially, the existing software stack in the current Bosch solution.

The conclusion drawn from this dissertation project is that AGL cannot replace the software stack, due to it's lack of real-time capabilities. On the other hand, it is a possible candidate to replace QNX in the software stack, for the reason that AGL has the same capabilities at a lower cost and higher customization level.

Keywords: Automotive, AGL, PDS, Diagnostic

Contents

List of Figures

List of Tables

Acronyms

ADAS	Advanced Driver Assistance Systems
AGL	Automotive Grade Linux
ALSA	Advanced Linux Sound Architecture
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
CAN	Control Area Network
CAN FD	CAN Flexible Data-Rate
CID	Card Identification
COTS	Commercial of-the-shelf
DAC	Digital to Analog Converter
DID	Diagnostic ID
ECU	Electronic Control Unit
eMMC	embedded MultiMediaCard
EOL	End-of-Line
FIFO	First In First Out
GPIO	General Purpose Input Output
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HMI	Human Machine Interface

ISO	International Organization for Standardization
LSB	Least Significant Byte
MAC	Media Access Control
MSB	Most Significant Byte
OEM	Original Equipment Manufacturer
OS	Operative System
OSI	Open Systems Interconnection
PCM	Pulse-code Modulation
PDS	Production Diagnostic Software
PNG	Portable Network Graphics
RAM	Random Access Memory
RGB	Red Green Blue
RID	Routine ID
RTOS	Real-Time Operative System
SID	Service ID
SoC	System-on-Chip
UCB	Unified Code Base
UDS	Unified Diagnostic Services
USB	Universal Serial Bus

1

Introduction

This chapter introduces the scope of the dissertation and some basic concepts, such as Production Diagnostic Software (PDS) and automotive Operative System (OS). With all this information, the objectives of this dissertation project and its appeal to the automotive industry are presented. The chapter closes with a broad outlook of the dissertation. It is important to keep in mind this dissertation was developed inside a company, Bosch Car Multimedia.

1.1 Context

Nowadays the suppliers for the automotive industry have the challenge to keep up with high demands, in both quantities and technologies provided. To keep with such demands, the supplier industry came up with software solutions for faster testing while maintaining test quality. One of the software developed is the PDS that runs on the target system to be tested. The PDS is responsible for receiving test commands from a control station, executing them in sequence and send back a response to the test commands.

At Bosch Car Multimedia, the concept of PDS is software that runs diagnostics on the production line. It is used for the final tests and configurations of a product in the production line. At the production line, PDS is used for embedded systems, such as infotainment systems, and their associated ECU, before it is shipped to the car assembly factory.

The solution developed by Bosch uses as a basis the Unified Diagnostic Services (UDS) international standard. The test is conducted by sending UDS messages to the ECU and the PDS framework executing the specified tasks, be it read memory address or execute a routine, such as cycle display colors.

Bosch's PDS is characterized by having routines and functions that test the processing of external trigger signals form other ECUs in the car, the correct processing of information triggered externally from sensors or ECUs and the correct connection of all its devices, like the display.

Currently at Bosch, the solution is a composition of two OSs, Vector AUTOSAR running alongside QNX, with QNX having a custom, UDS compliant, diagnostic application running, and Vector AUTOSAR, a RTOS, with device control, communication interfaces and a native diagnostics engine also compliant with UDS. The AUTOSAR comes with a architecture dedicated to the automotive industry and a framework for diagnostics. The two OSs are running in different processors in the SoC and the communication between them is made through a custom inter-processor communication protocol, this is what synchronizes the whole system and makes it function as a single system. The inter-processor communication is used to send the messages received via CAN on AUTOSAR to the application running in QNX, and also to send responses from the diagnostics to AUTOSAR.

This solution presents some problems, due to its architecture and implementation. One of the problems is the usage of the same diagnostics engine for the production tests and the end user. A diagnostics engine is what interprets the UDS messages and executes the diagnostics, and the current PDS uses the same engine for both end user application and production diagnostics application. Since the engine is shared between use cases, it is very complicated and the majority of its functionalities go unused in the PDS application. Other problems are the non removal of the PDS and the utilization of two OS. The utilization of two OS increases the complexity of the system and makes developing for the system harder.

1.2 Motivation

A current trend in the automotive industry is the use of open-source software, this led to the development of the Automotive Grade Linux (AGL) in recent years to standardize the software present in the devices and accelerate the development of new applications. The main motivation in this dissertation is the exploration of AGL as a way to produce a custom environment for PDS.

1.3 Objectives

This dissertation, besides exploring a different OS, creates a PDS application that tackles some of the problems with the current solution.

The main objective of this dissertation is the exploration of AGL as a potential platform for PDS applications. For this end, it is developed a proof-of-concept system of a PDS application running in AGL, with the aim of testing and exploring this new OS.

1.4 Dissertation Structure

In this dissertation, the reader is presented with the Background chapter, which explains all the topics involved in the implementation of the project, allowing to understand the needed technologies and the subjects to be used or that help development.

The following chapter, PDS application specification, gazes upon the analysis of the system to be implemented, it's requirements and how this dissertation tries to tackle the problems presented. But also, the design, choices made to make the project happen and the decisions taken to make the project function on paper.

The Implementation and Tests chapter comes next and uncovers the implementation of the project. The chapter closes with tests and tools used to test the system to then evaluate the success of the developed project.

Finally, this dissertation ends with the Conclusion, where the final solution is warped up and summarized, terminating with the necessary work to perfect it and publish.

2

Background

In this chapter, the concepts used by this dissertation are explained, starting on more abstract concepts and narrowing it down to the core of the problem presented in this dissertation. Starting with a definition of Production Diagnostic Software (PDS) and Unified Diagnostic Services (UDS), and ending with the current solution used by Bosch and the new emerging Automotive Grade Linux (AGL) OS.

2.1 Production Diagnostics Software

Production Diagnostic Software is software designed to automate the tests necessary to validate the hardware before it is integrated in the final device [1, 2]. Test automation is a way to simplify and accelerate the diagnostics of faulty hardware and it is essential to realize time-sensitive action sequences [3]. With the increase in market demand, the PDS and its automation assume a greater relevance given the need to increase the testing cadence while maintaining the level of robustness of the tests [2].

Diagnostic software is software created and used to retrieve fault information stored in an ECU, this information will then be used to diagnose failures in the hardware [1]. This software follows the Unified Diagnostic Services (UDS) standard to support the communication part of the test system and the definition of diagnostic. Next, can be visualized the Figure 1 with a example EOL test system.

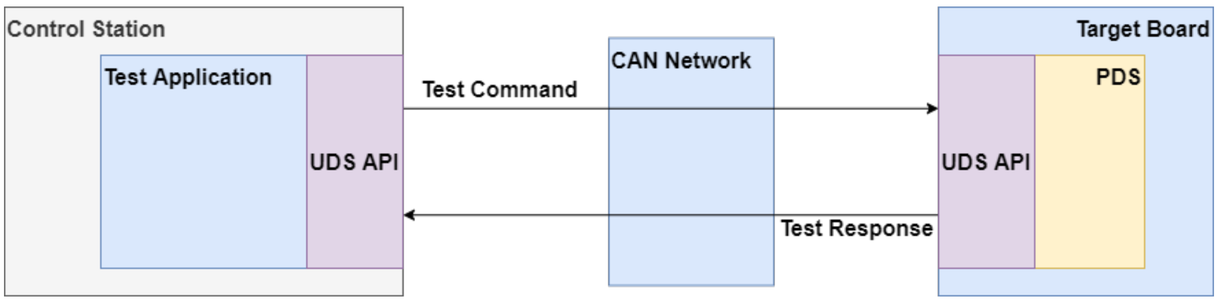


Figure 1: Typical test setup for EOL tests

2.1.1 Unified Diagnostics Service

Unified Diagnostic Services (UDS) is a communication protocol developed for the automotive industry, specified by the ISO 14229-1 [4], to define common diagnostics practices independently of company or car. For that purpose the UDS standard uses as a base the Open Systems Interconnection (OSI) basic reference module, which divides the communication systems into seven layers. At the top layer, or application layer, is where UDS resides. The standard also defines all the values for the information bytes, values for errors, error states and program sequences to be implemented.

The UDS standard specifies that all the diagnostics in a car are comprised of diagnostic services, that are predefined by the standard and have a Service ID (SID), and their respective parameters and Routine ID (RID), if available [4].

A diagnostic service, as per the specification, is an exchange of information started by a client in order to retrieve relevant information from a server. Here, a client is the tester or the equipment that sends the diagnostic messages and the server is the hardware being tested. The UDS specifies a multitude of diagnostic services. Some of the services specified are SessionControl, ECUReset, DiagnosticSessionControl, RoutineControl.

The UDS standard, not only defines the available services but also the message structure and a plethora of codes used in the response messages. The request messages are defined as starting with the SID, next the Service Sub function (if applicable), then the Diagnostic ID, and ending with the Diagnostic parameters, as Figure 2 shows. The response is similar to the request message, it is specified that if the execution of the diagnostic is correct the response shall start with the SID value plus 40 in hexadecimal (i.e. for the SID 0x31, the positive response will be 0x31+0x40 = 0x71), this is followed by the Diagnostic ID and by the data, as Figure 2 demonstrates. For the negative responses, the specification created a special SID called Negative Response Code (NRC) ID used for the purpose of sending negative responses to the tester. This NRC ID as the value of 0x7F, and in the negative response format, is followed by the SID that was called and ends with the negative response code, as Figure 2 shows.

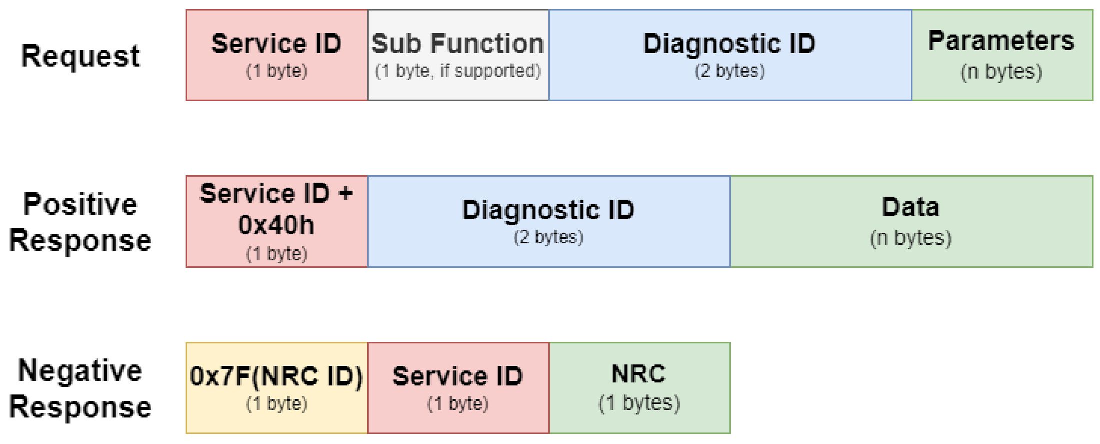


Figure 2: UDS messages format

For this dissertation, the most important service specified by the standard for diagnostics purposes is the RoutineControl service. The RoutineControl service is used by the tester to execute a sequence of defined steps and obtain results. It is a service of high flexibility, since the implementation of the steps is defined by the project requirements and it is used for complex control sequences. The Routine Control is defined as having an ID of 0x31h and having one parameter to control if the routine is start, stop or request result, thus giving control to the tester if he wants to start, stop or receive a result from a given routine service.

Table 1 helps visualize the request message for a RoutineControl service. The message contains the SID for RoutineControl, and then, the RoutineControlType variable that controls the type of execution, either start (0x1h), stop (0x2h) or request result (0x3h). After that comes the RID, that defines the sub-function to execute in the diagnostics program and has a two byte length ID, the next part of the request message is optional depending on the need for execution parameters or not and it is the parameters section, every parameter is one byte.

Data byte	**Parameter Name**	**Value**
#1	Service ID	0x31h
#2	RoutineControlType	0x1h, 0x2h, 0x3h
#3 #4	Rotuine ID [] = [MSB, LSB]	0x0h - 0xFFFFh
#5 - #n	Optional Parameters[] = [param1,..., paramN]	0x0h - 0xFFh ... 0x0h - 0xFFh

Table 1: Routine Control Request message format [4]

2.2 Support Software for PDS

The PDS, as with all applications, runs on a OS and has software that supports its implementation and functionality. For this end, it is usually developed a software stack composed of two OSs.

Currently, at Bosch, the software stack is composed of QNX on top of AUTOSAR. Next subsection gives a brief explanation and insight of this solution.

This subsection also shows some technical concepts and state of the art about AGL, as a newly emerging automotive focused OS.

2.2.1 Current Bosch solution

As explained in the contextualization, the current solution, as seen in Figure 3, uses two OS running in different processors on the SoC and are connected via a custom inter-node communication, or INC, used for communication between processors. The two OS are Vector AUTOSAR and QNX, and each have their own tasks. The AUTOSAR deals with hardware directly, offers implementation of device drivers, communication interfaces, offers abstraction from hardware, implements the majority of the communication part and it's responsible for the main state machine that controls the flow of execution of the diagnostics. AUTOSAR also has a diagnostic engine that it is native to the OS. QNX offers access to a file system, control of displays and audio and a custom diagnostic engine to execute UDS diagnostics.

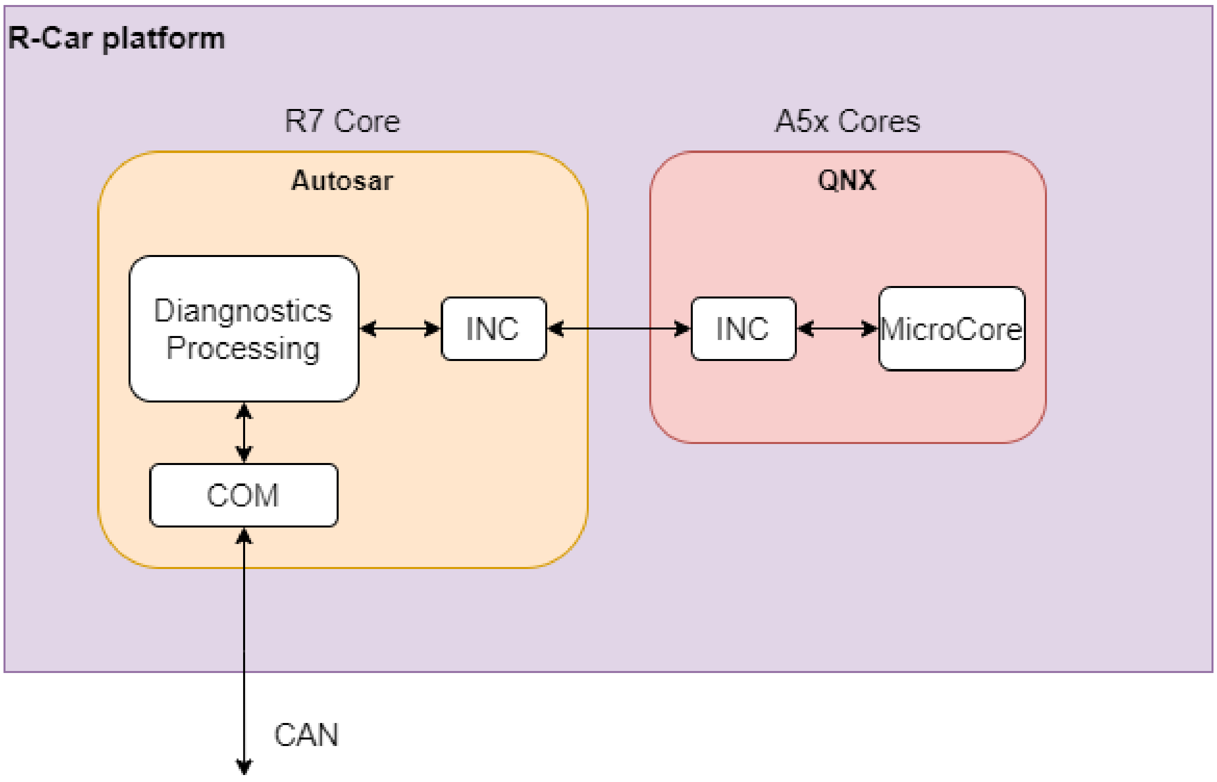


Figure 3: Block Diagram of Bosch Solution

The current booting process is composed of two bootchains A and B, each containing two instances of AUTOSAR and QNX. The bootchain B is used for production line tests and the other for the end user software. The control of what bootchain is booted is done by the bootloader, through a custom boot flag. By default, the boot flag enables the bootchain B. The problem with this solution is that, when the tests are done, the tests bootchain is not deleted but is, however, deactivated through the change and subsequent locking of the boot flag to boot the end user bootchain.

The current solution uses the Vector AUTOSAR for the communication and diagnostics part of the UDS specification, and QNX only for specific diagnostics. On the QNX part of the system, the diagnostics engine used for the production tests is the same as the one that it is used for the end user software. This diagnostics engine, called MicroCore, has more functionalities implemented than what is needed for production tests, so, it presents the developer with high complexity. Also, the MicroCore does not have any communication capabilities apart from the inter-node communication, or INC, used for communication between processors, and no simple way of adding more communication interfaces.

The flow of execution of the current solution is as follows. Upon receiving a UDS message in the AUTOSAR communication interface, the AUTOSAR application parses the message and checks if the diagnostic exists and if the diagnostic is supposed to run on AUTOSAR or QNX. If the diagnostic is supposed to run on QNX, the UDS message is sent, through the inter-node communication interface, to QNX's MicroCore application. Afterwards, MicroCore upon receiving the message proceeds to parse it and execute the diagnostic. Upon finishing the diagnostic execution, MicroCore sends the response, via the inter-node communication to AUTOSAR, that then sends the response to the CAN communication bus.

As with all known systems, the system used by Bosch currently contains problems. The largest problem currently is the high complexity of the architecture. The architecture as it stands is the same as the end-user architecture, which for production diagnostics is highly complicated and goes mostly unused. The high complexity also means more development time for new features and a less flexible system.

Other problem with the current system is the usage of closed-source software, which limits the flexibility of the system, brings with it high licensing costs and the access to the code is limited.

2.2.2 Automotive Grade Linux

Automotive Grade Linux (AGL) is an open-source operating-system targeted to the automotive industry based on the Linux kernel. The main focus of AGL is to create OSs for infotainment systems and a framework to build automotive software applications [5].

Automotive software dates back to the 1970s where systems where programmed in assembly and only in the 1990s, the C language became a industry standard for systems programming, but with the introduction of new technologies and increased complexity, software became complex too and the creation of AUTOSAR and recently AGL came to help with the problem [6, 7].

Since the complexity of cars tends to increase, the need for a more complex and robust OS and hardware to keep up with the increased demand for safer, faster and reliable automotive systems. Recent production vehicles can have upwards of 100 million lines of code across all systems and take 2 to 4 years to develop, compared to smartphones with only 12 to 15 million lines of code [6]. Nowadays automakers are using Windows Embedded Automotive, QNX or Ubuntu for the OS of the systems, but, each car on the lineup can use a different customization of an OS. AGL comes to fix this lack of standardization [8].

AGL became available in 2012 [8], but some articles from 2005 already refer to some automotive-grade Linux, not AGL but a concept or idea of a Linux distribution for automotive [9].

The main concept of the AGL is that the OS acts as a framework that can be used by OEMs and suppliers to make their own versions by integrating their own modules. The main target applications for

AGL are telemetry, HMI and instrument clusters. The base building blocks can also be reused from other Linux embedded distributions due to the similar kernel [8].

The AGL uses Unified Code Base (UCB) and the ”code first”approach to reduce time-to-market for the product and does this by implementing 70-80% of the necessary software bases for the supported boards, thus reducing the workload to the developers to only 20-30%, which consists of specific application software and customization [10].

AGL in the real world

AGL has already been used in the automotive world, with some new cars infotainment being totally developed with this OS, as it is been showed, the Toyota Camry was the first Toyota car in the United States market with an infotainment system totally developed in AGL [5].

Other proof-of-concepts have been made with this OS, one of them presented in [7], which goes into the concept of connected vehicles by binding adaptive AUTOSAR with AGL. The author goes into exploring the usage of well documented and well implemented AUTOSAR to develop the low level of the project, but developing the whole Graphical User Interface (GUI) with AGL. The benefit is that the AUTOSAR software can focus activities into secure software and time sensitive networks, while AGL focuses on GUI.

More uses of AGL in the automotive world were explored by papers such as [11]. In this paper the authors develop AGL applications for Advanced Driver Assistance Systems (ADAS) using Linux as a basis, but refer AGL as a candidate to develop ADAS with the use of a GPU for acceleration. The authors also mention the lack of reports on the real-time performance of such systems developed with AGL.

AGL software architecture

AGL presents the user with 4 layers in its software architecture [8]. The top layer, and most abstract, is the application/HMI layer. The layer is composed of the things the driver and passengers can interact with, mainly the infotainment apps, radio, GPS, etc. The next layer in the architecture is the application framework, this layer provides all the necessary Application Programming Interface (API) to develop applications. The layer bellow is the service layer which contains all the user-space resources and automotive services. At the bottom of the stack, the OS layer includes the Linux kernel and all the device drivers.

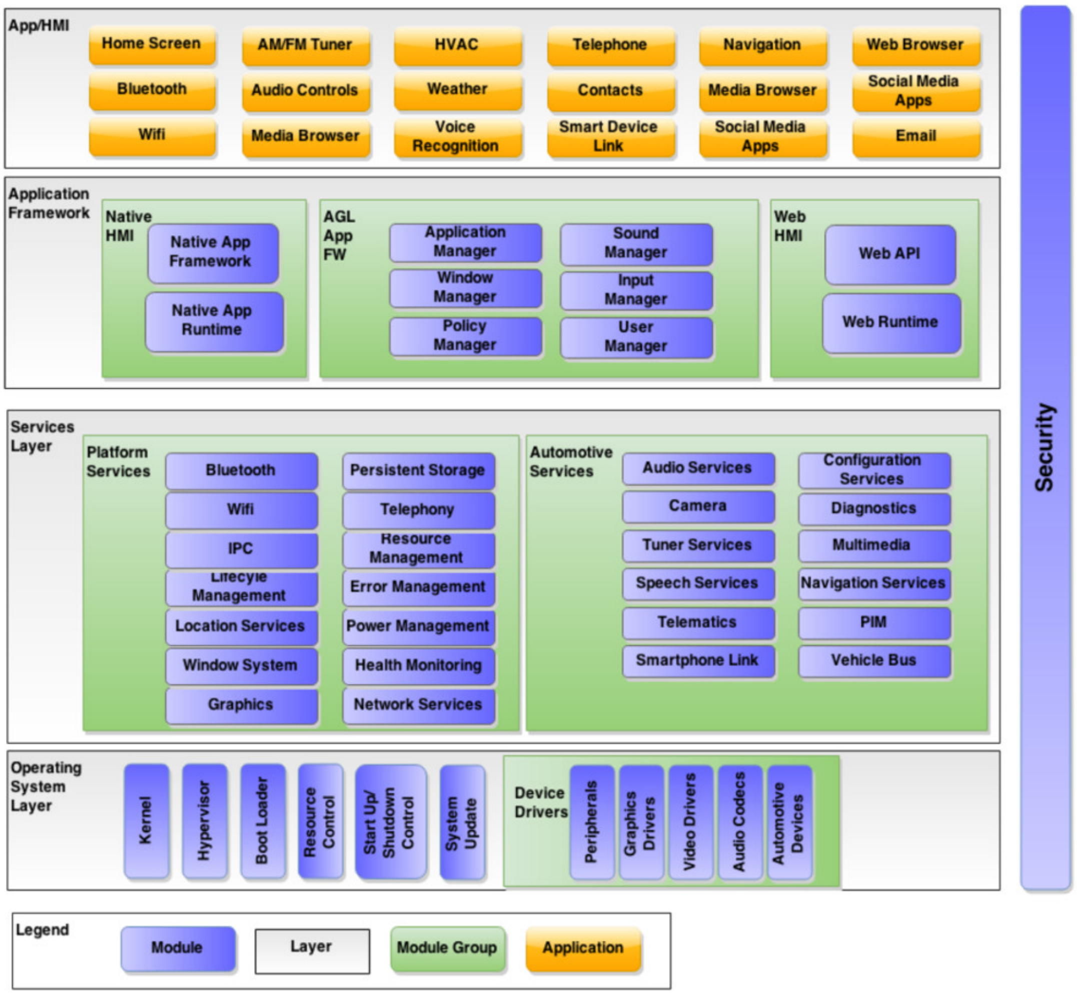


Figure 4: AGL software architecture [8]

3

PDS application specification

3.1 Problem analysis

To validate AGL as a possible platform for PDS applications, it is intended to use a custom developed PDS application. This PDS application must contain different diagnostics that will be used to test the viability of AGL, as well as, simulate a real PDS application in the real world.

As requirements for this application the diagnostics to be conducted need to mimic the Bosch solution including using the UDS specification. The communication must use CAN protocol.

The set of diagnostics that are going to be implemented, as a proof-of-concept, seen in Table 2, are a subset of the tests performed by the current Bosch solution.

Diagnostics
Read MAC Address
Read embedded MultiMediaCard (eMMC) Card Identification (CID)
Play/Stop Audio
Read/Write General Purpose Input Output (GPIO) pin
Read/Write Memory
Display Test Picture
Display Generated Image

Table 2: Diagnostics to be implemented

The diagnostics to be implemented are presented in the next paragraphs.

Read MACAddress

This diagnostic reads the MACAddress of the main Ethernet interface and should return error either if the Ethernet interface is not configured or reading its MACAddress fails.

Read eMMC CID

This is a diagnostic used to read a special register present in the eMMC used. This register contains manufacturer specifications and identifcations for that eMMC chip. For this diagnostic error shall be returned if the register is not accessible or no eMMC is present.

Read/Write GPIO

Two diagnostics, one for reading a value of a given port of the GPIO, and the other for setting a given value of a given port of the GPIO. Errors are returned when the pin is not accessible or out of bounds.

Read/Write Memory

Two diagnostics with similar functions as the Read/Write GPIO, but in this case it reads or writes a value from a Random Access Memory (RAM) address. The diagnostics should return error if the memory address is out of bounds, protected or it is not possible to open due to concurrency.

Play/Stop Audio

It is used to play and stop the output of a sine wave of a given frequency at a given volume, the frequency and volume are parameters defined by the tester in the request message. The diagnostic shall throw a error if the audio device is not accessible, wrongly configured or some problem occurs with the diagnostic execution.

Display Test Picture

Shows, on the screen, a picture that is stored in memory. This diagnostic returns error if the screen is not present or the picture is not available.

Display Generated Picture

Creates a quadrilateral through two vertices positions and gives the background and the quadrilateral a color, all this vertices and colors are received from the tester. This diagnostic returns error if the values sent for the size of the geometry are bigger than the display pixel size.

3.1.1 Diagnostics Behaviour

The diagnostics to be implemented for this application follow the same behaviour, present on Figure 5. All the diagnostics requests are received by the Communication Interface entity, that forwards it to the Message Processor entity. the request passes through a Message Processor to extract relevant data from the message. After this, the contents of the message arrive at the Diagnostics Processor that, with the contents of the message, will send the diagnostic ID to the Lookup table to retrieve the diagnostic and proceed to execute the diagnostic with the given parameters.

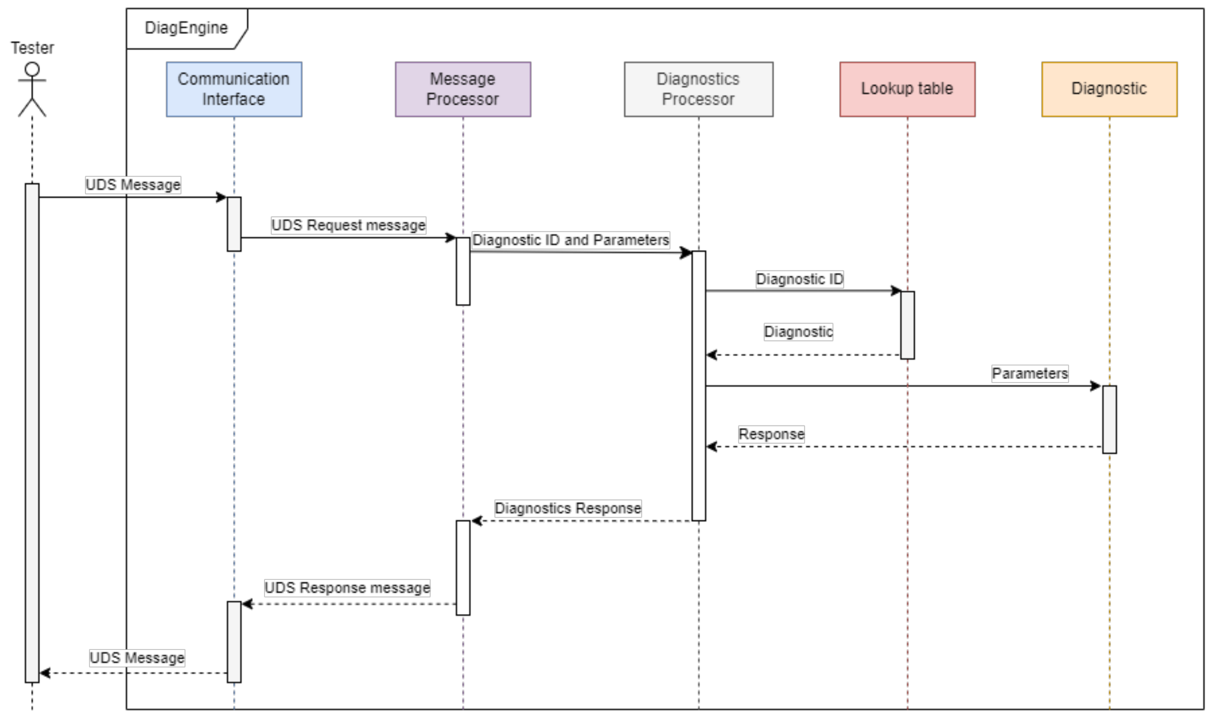


Figure 5: System sequence diagram.

Next it's described each of the entities identified by the sequence diagram.

Communication Interface

This entity is the entry point of the system. It gives the system the ability of receiving UDS messages from the tester and send responses out. This entity receives UDS compliant commands from the tester. These commands are transferred directly to the Message Processor without any pre-processing.

Message Processor

This entity is responsible for the processing of the raw UDS messages. This entity extracts the relevant information of the message such as the SID, the RID and the parameters, the Diagnostic ID is a composition of both the SID and RID.

The Message Processor after its execution returns all the information to the Diagnostics Processor, which itself proceeds to lookup the Diagnostic ID through the Diagnostics Lookup Table.

Diagnostics Processor

This entity is responsible for receiving the processed messages from the Message Processor, for sending an ID to the Diagnostic Lookup table and for executing the needed Diagnostic.

The most important requirement for this module is the queuing of Diagnostics, meaning that if a Diagnostic is executing the system can receive other commands and store them in a queue. Other requirements are the separation of the Diagnostic execution and the other parts (Message Processing and Diagnostic Lookup).

Diagnostics Lookup

This entity is responsible for storing all the diagnostics available for execution and, when prompted, to return a specific diagnostic with the given Diagnostic ID. The Diagnostic ID is sent to the module via the Diagnostics Processor.

The process through which the Diagnostics Lookup module knows the available diagnostics and their IDs is through a configuration file. The Diagnostics Lookup then puts the information into a table.

3.1.2 Block Diagram

The Figure 6 shows a block diagram of the proof-of-concept application to be developed. As it can be seen, the Diagnostics Processor entity is the core of the PDS application and the one with the task of connecting all the other blocks together.

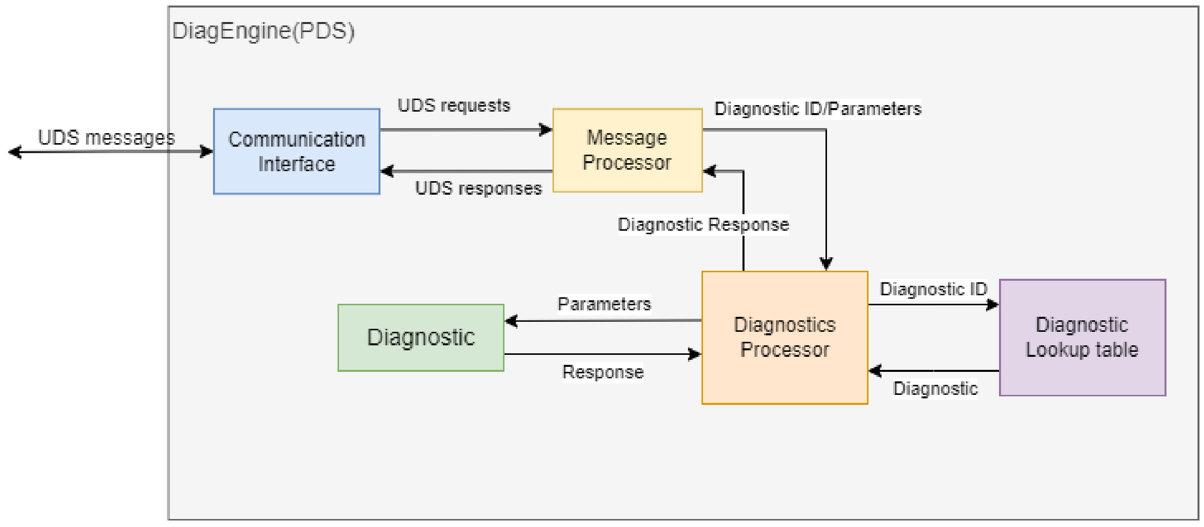


Figure 6: System block diagram.

3.2 Application Design

Based on the block diagram presented earlier in Figure 6, this section provides the detailed design of each of the entities as well as their integration into a PDS-driven computing system.

3.2.1 Entities Specification

This following section is dedicated to the design of the PDS application or DiagEngine used in the project. It starts with the core part of the application and its modules, like the Scheduler and Communication Interface, and the choices taken to make them possible, like the usage of Commercial of-the-shelf (COTS) modules or the creation of a algorithm. This section closes with the design of the different diagnostics.

The base of the DiagEngine is the UDS specification. Most of the design of the application was influenced by the specification, specially the diagnostics and message processor part.

The requirements of high flexibility inposed to the DiagEngine shaped its design. To make the DiagEngine flexible it is going to have a modular approach. This means all the diagnostics and communication parts of the application will be shared library modules, that can be added, in runtime, to the application through a set of configuration files and compilation options. To also make the DiagEngine more flexible, in terms of development, it implements abstraction from the communication and diagnostics. This means that from the DiagEngine all communication protocols are routed through a abstraction class and all diagnostics are UDS Services (Basic UDS specification for diagnostics types).

To understand the abstraction used for the diagnostics is important to understand the UDS specification for the diagnostic. The specification provides various services, but for this project all the diagnostics are going to use a Routine Control service, since it is the best for execution of specific functions. This service, as per the specification, offers a start, stop and return result subfunctions. This subfunctions are what the diagnostics execute. Most diagnostics use only the start subfunction, with one, the play/stop audio, using the start and stop subfunctions.

Communication Interface

The Communication Interface of the DiagEngine will use a hybrid design, meaning it will take some COTS libraries and integrate them into a custom algorithm. Since the DiagEngine is modular, so is the Communication. This means that the proposed design for this module is a abstract class that makes a interface with the outside and the different classes for each communication protocol inheriting the base class and implementing the necessary functions. This creates a abstraction from the communication protocol.

It was set that the Communication part of the DiagEngine needs to support serial as well as CAN communication.

For the serial communication it will be used a custom algorithm. This algorithm reads the contents of the command line after a carriage return. This then is copied to a vector of bytes that then is sent to the Scheduler.

For the CAN communication a COTS library called SocketCAN will be used. SocketCAN offers APIs for creating and managing CAN sockets, but also, to configure the communication to enable CAN FD and filters. The communication will start by creating a socket to open a communication in the CAN transceiver and configure it to enable CAN FD. The reading part will be managed by a thread, this thread will be checking if a message is received by the transceiver and send it to the scheduler. The sending will be managed synchronously by the Scheduler by calling an API from the communication interface.

The Figures 7 and 8 show the flowcharts for the two types of communication, serial and CAN.

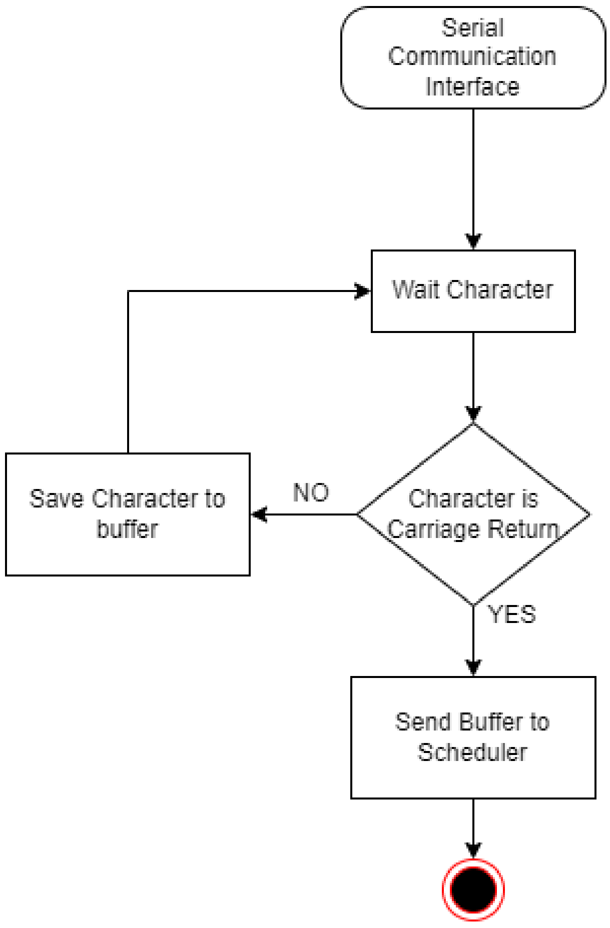


Figure 7: Serial communication flowchart

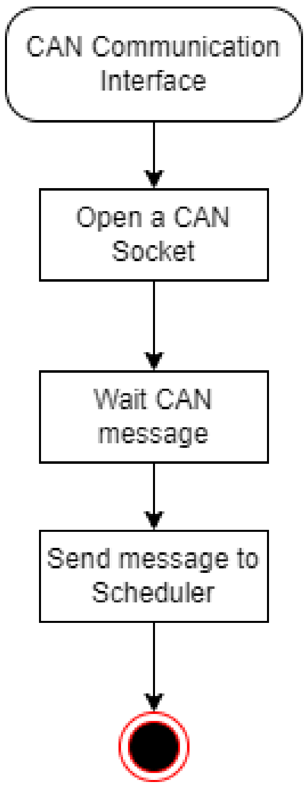


Figure 8: CAN communication flowchart

Message Processor

The Message Processor will be done using a custom algorithm. Since there is no available COTS algorithms for UDS message parsing, a algorithm needed to be made.

The algorithm works as follows, first it starts by analysing the first byte of the message, this byte contains the information about the SID, as per the UDS specification. Next, the algorithm sees if the SID contains any parameters, for example the RoutineControl service contains a one byte parameter (see Table 1). After the algorithm stores the SID and, if they exist, the parameters, it proceeds to parse the service sub-function ID. This sub-function ID has a length of two bytes. This sub-function ID and the SID are stored in a data structure, which is then used to search the table of diagnostics. Finally the algorithm parses the parameters of the service sub-function and saves them to a buffer to be later used by the diagnostic. Figure 9 shows a flowchart of the algorithm designed.

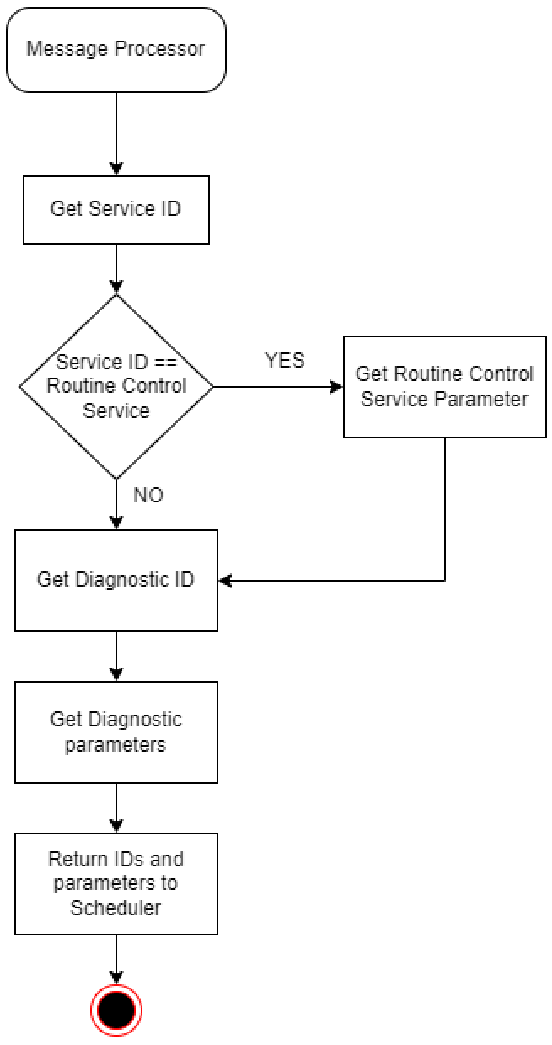

Message Processor
Get Service ID
Service ID == Routine Control Service
YES
Get Routine Control Service Parameter
NO
Get Diagnostic ID
Get Diagnostic parameters
Return IDs and parameters to Scheduler

Figure 9: Message Processor algorithm flowchart

Diagnostics Lookup table

For the diagnostics lookup table module it is mainly used COTS data structures and functions, that are already present in the programming language used. The COTS will then be encapsulated in custom code and use custom data structures that take leverage of said COTS.

For the storage of diagnostics it is used the C++ class map. A map is a specific table that can store and retrieve content using a ID or key. The content and key can be whatever data structure the developer wants (i.e. strings, classes, integers, etc.), this enables high flexibility and the ability to use complex data structures, either for keys or content.

For this module the map is used because of its properties, such as, unique keys (no two elements can have the same key), self dynamic memory allocation and associative nature. Also, it enables the storage of diagnostics by using a custom data structure for the map key. This custom data structure is called UDS ID and it is composed by the SID and DID. This was done to make searching for diagnostics easy, by just searching its SID and DID.

For the insertion of elements in the Diagnostics map a COTS algorithm and design, previously developed by Bosch, is used. This works by compiling the diagnostics as a shared object library and storing them in a directory inside the target hardware filesystem. The libraries are then fetched in run-time by a algorithm that creates a pointer to the class in the map and an association is then made between the Diagnostic ID and the Diagnostic by using configuration text files, that associate the ID with the Diagnostic Name. The flowchart for the storage of diagnostics can be seen in the following Figure 10.

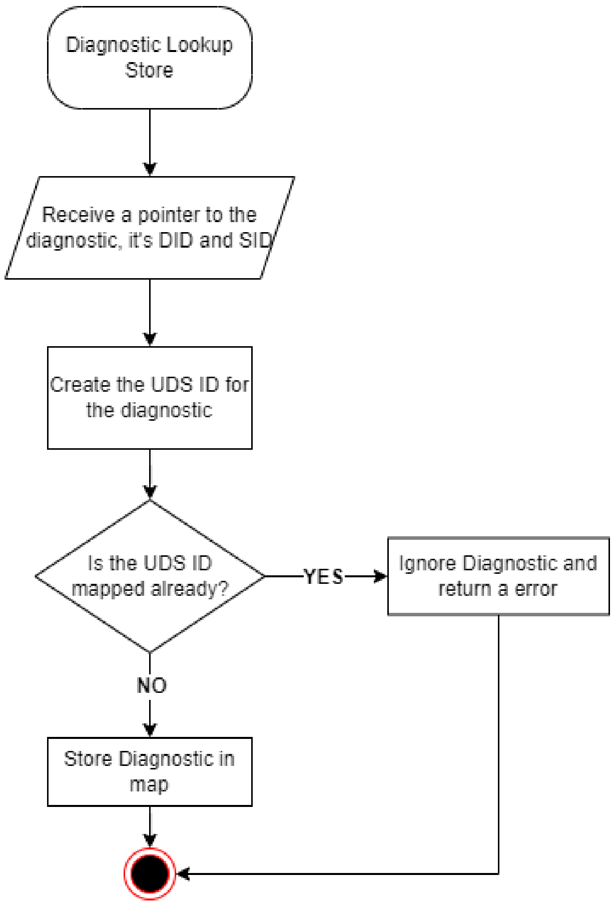

Diagnostic Lookup Store
Receive a pointer to the diagnostic, it's DID and SID
Create the UDS ID for the diagnostic
Is the UDS ID mapped already?
YES
Ignore Diagnostic and return a error
NO
Store Diagnostic in map

Figure 10: Diagnostic Lookup Store diagnostic flowchart

For searching and retrieving the diagnostic, it is used the UDS ID and the map function find, this function takes in the map key as a parameter (in this case the UDS ID) and returns an iterator to the specific map position where the key was found. This iterator contains the key and the value it stores, in this case the value will be the diagnostic.

The map will be encapsulated in a class, that contains a find and a store function, and the UDS ID structure and the map as class members. The store function will be used on at the start of the application to store all the available diagnostics. The find function will be the one used by the scheduler to retrieve the diagnostic. The searching and get algorithm can be visualized further with the flowchart presented next in the Figure 11.

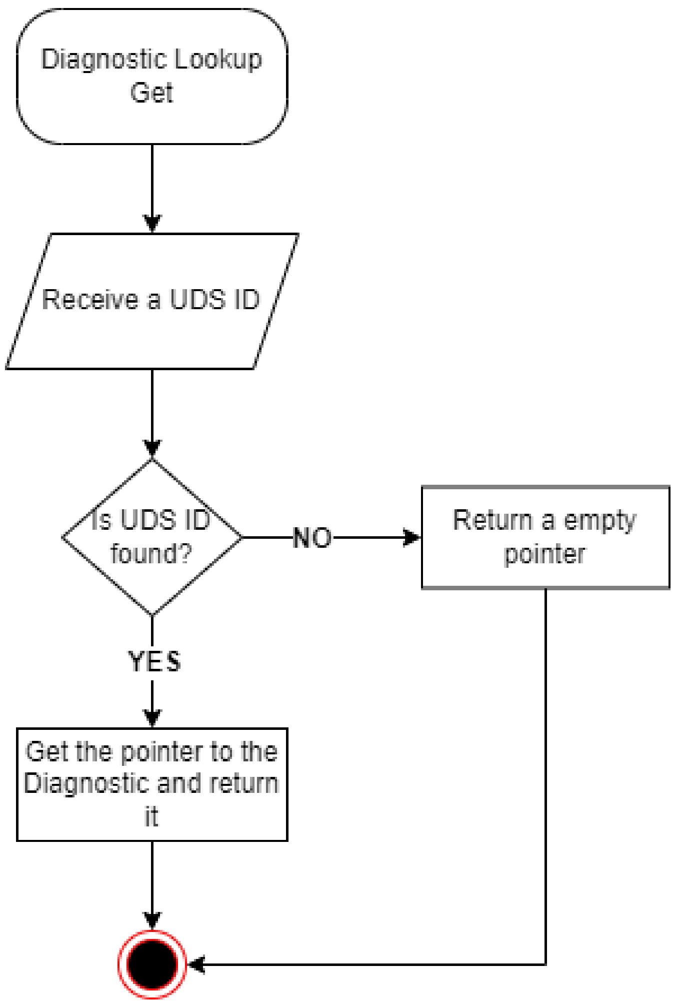


Figure 11: Diagnostic Lookup Get diagnostic flowchart

Diagnostics Processor

This entity is responsible for the retrieval and execution of diagnostics. This entity is divided into two parts, a synchronously called part for retrieving the diagnostics and a asynchronous part for executing diagnostics. The asynchronous part is where the diagnostics are called to execute and where the response is sent after their execution is terminated and it is totally independent of the rest of the application. The synchronous part retrieves the diagnostic from the table and places it in a queue to later be executed.

The design of this module uses custom algorithms for both the synchronous and asynchronous parts. For the synchronous part, the algorithm receives the diagnostics IDs and parameters from the Message Processor and receives the corresponding diagnostic from the Diagnostics Lookup table entity.

The algorithm for the asynchronous part uses a thread and a queue with the First In First Out (FIFO) method of organization maintaining the events in the same order of reception. When this part enters execution it retrieves the diagnostic from the queue and executes it. As the Figure 13 shows, the thread starts by checking if the queue of events is empty, if it is not, it pops the top of the queue and proceeds to call the diagnostic execution synchronously. After the diagnostic executes, its response will be sent to the communication part to later be sent via the communication bus. After all this, the thread executes again from the top of the algorithm.

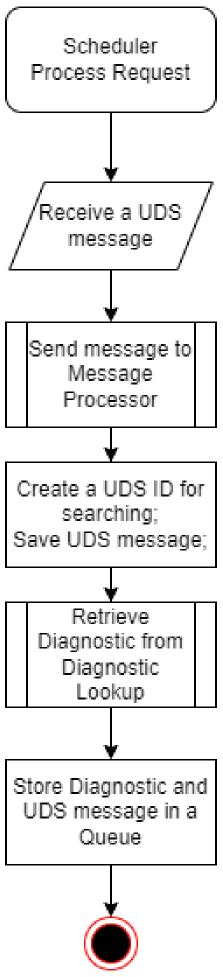

Scheduler Process Request
Receive a UDS message
Send message to Message Processor
Create a UDS ID for searching; Save UDS message;
Retrieve Diagnostic from Diagnostic Lookup
Store Diagnostic and UDS message in a Queue

Figure 12: Diagnostics Processor synchronous part flowchart

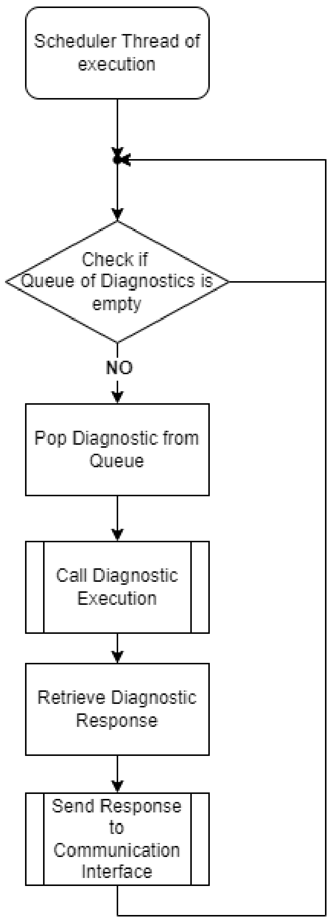

Scheduler Thread of execution
Check if Queue of Diagnostics is empty
NO
Pop Diagnostic from Queue
Call Diagnostic Execution
Retrieve Diagnostic Response
Send Response to Communication Interface

Figure 13: Diagnostics Processor thread flowchart

3.2.2 Diagnostics

This entity is based on a COTS design that is already used by Bosch and it is based on UDS protocol specifications for diagnostics and services.

This design consists of a hierarchy of classes, presented on Figure 14. It is based on a fully abstract class that represents the concept of Service in the UDS protocol specification. This class is inherited by different classes that represent the different Services available and each Service can, in turn, be inherited by several classes that implement different diagnostics.

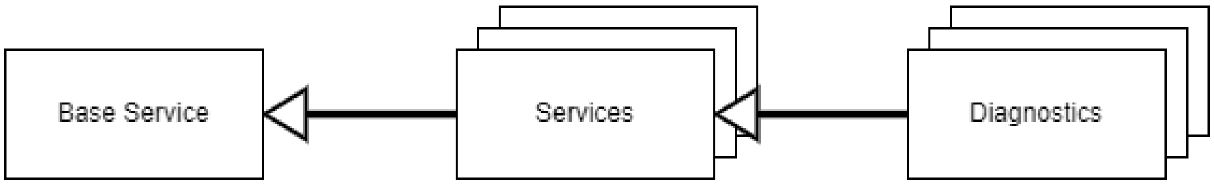


Figure 14: Diagnostics Class diagram

Diagnostics Specification

Each diagnostic class, referred above and in Figure 14, will equate to the diagnostics to be executed by the system, referred on table 2.

Read MACAddress

This diagnostic reads the MACAddress of the Ethernet device present on the board and retrieving that information to the tester. This diagnostic will take leverage of the Linux method of presenting devices as files. For this diagnostic it will be used COTS functions that are available in the programming language, to read the file that contains the MACAddress of the main Ethernet device of the board.

Read eMMC CID

For this diagnostic it was designed a custom algorithm that takes advantage of the Linux way of showing devices as files in the filesystem. The algorithm for this diagnostic works by searching a specific file on the sys folder of Linux, this file is called ”cid”and it contains the CID value for the specific eMMC card. This diagnostic on call will proceed to open the cid file and read its contents. Later it parses the contents to the response buffer to be sent to the tester. The response of this diagnostic will contain a 16 byte value of the register. The CID is a special register on every eMMC device that contains relevant manufacturer data.

Play/Stop Audio

This diagnostic uses a COTS library, namely the ALSA library, for controlling the audio device. The ALSA library, or Advanced Linux Sound Architecture library, offers APIs to control the sound devices of a Linux system, it can be used to control volume or send specific values to the DAC. For this diagnostic it will be used the PCM interface from ALSA library.

This diagnostic is composed of two actions, a start play audio and a stop audio. For this the parameter of the Routine Control Service will be used.

For the start play audio, the diagnostic works by opening and configuring the target device first. Next is setting the volume given by the parameters. Afterwards it is calculated a sine wave with the frequency parameter and the values are saved to a buffer. After all the setup is complete the diagnostic proceeds to create a thread to send the values of the sine wave to the device by using the ALSA API function snd_pcm_writei. A thread is used to enable the output of sound without blocking the system execution and to output sound until a stop command arrives.

For the stop audio, the diagnostic works by signaling the thread to stop, using an atomic variable, this makes the thread finish sending the pending values and stop its execution. It then ends by cleaning up the configurations made to the device and closing all the open devices.

Read/Write GPIO

For the design of this diagnostic it will be used a COTS API for Linux called libgpiod. This library offers an abstraction layer from the ioctl functions to control the GPIO pins of a board.

For this diagnostic it will only be used the start function of the Routine Control Service. The parameters for the read, as explained in the previous chapter, are the GPIO channel number and the pin number. The parameters for the write are the same as the read with one extra, the value to set the pin to, which is either 0 or 1 which translates to either high or low.

The way the reading diagnostic works is by first opening the gpio chip whose number is received in the parameters by using the libgpiod APIs, afterwards, it is opened the pin by using the number received in the parameters. If all the processing is done correctly and does not raise error, the diagnostic shall return the value that the specified pin in the specified gpio chip is set at.

For the writing the diagnostic sends the channel, pin and value to the GPIO driver. The response should contain the value the tester sent.

Read/Write Memory

This diagnostic is mostly used to read/write values from certain addresses on the RAM. For this the read receives the size of the values (8, 16 or 32 bit) to be read and the memory address. The write receives the size, the memory address and the value to write.

The design of the read memory diagnostic is simple, basically it starts by opening the system RAM file descriptor in read mode and mapping its address to a pointer. Afterwards it just basically reads the value of the pointed address and passes it to a byte array.

For the write memory diagnostic, it also opens the system RAM file descriptor, but, in write mode. It maps the address to a pointer and then simply proceeds to copy the received value from the parameters to the values of the pointed address.

All of the operations for opening, mapping, reading and writing will be encapsulated in a custom class to help the portability of the code and to not repeat code, since this is used in two diagnostics.

Display Test Picture

This diagnostic is responsible for reading the pictures inside a specified folder in the filesystem, index them, process them and show them accordingly in the display connected to the development board.

The chosen file type for the pictures is Portable Network Graphics (PNG), since there are COTS libraries available to process this file type in Linux, and for this diagnostic the libpng and png++ wrapper are used. The libpng is the reference library for PNG manipulation and supports almost all PNG features. The png++ is a C++ wrapper interface for the libpng that offers better C++ compatiblity and reduces complexity of data manipulation and handles initialization/deinitialization of structures. This COTS modules are used due to the immediate availability and ease of integration they presented.

For storing the pictures in the filesystem it is used a specific folder to only contain PNG pictures. The diagnostic will read the pictures in the directory and index them in a array. To make it possible to show the same picture every time the same index is received, the pictures are stored in a orderly fashion by using their file name to store the index number (i.e. "1-TestPicture.png").

When a diagnostic start is called the diagnostic will search for the specified picture index, retrieve it and process it to convert from PNG to raw binary data buffer. The conversion to raw is done through the png++ wrapper functions. After the image is transformed from PNG to raw binary, the raw binary buffer is sent to the display driver to be displayed in the screen.

Display Generated Image

This diagnostic is responsible for taking the position of two vertices of a quadrilateral and display it on the screen, with a background color and a color for the quadrilateral. The diagnostic will receive a 3 bytes for the RGB values of the quadrilateral, followed by 3 bytes for the RGB values of the background. The vertices are given as pairs of x and y values of pixels, each x and y is given by 2 bytes each, forming a value like 0xXXXX 0xYYYY for one vertex. The following Figure 15 helps visualize this idea.

For this diagnostic, the design uses a COTS code developed by Bosch to convert the vertices position to a final image. Since the vertices are received as the absolute position on the screen, the calculations involved are not complicated. The algorithm works by representing the screen as a matrix with the size of the screen in pixels, each coordinate in the matrix containing the RGB value of the pixel.

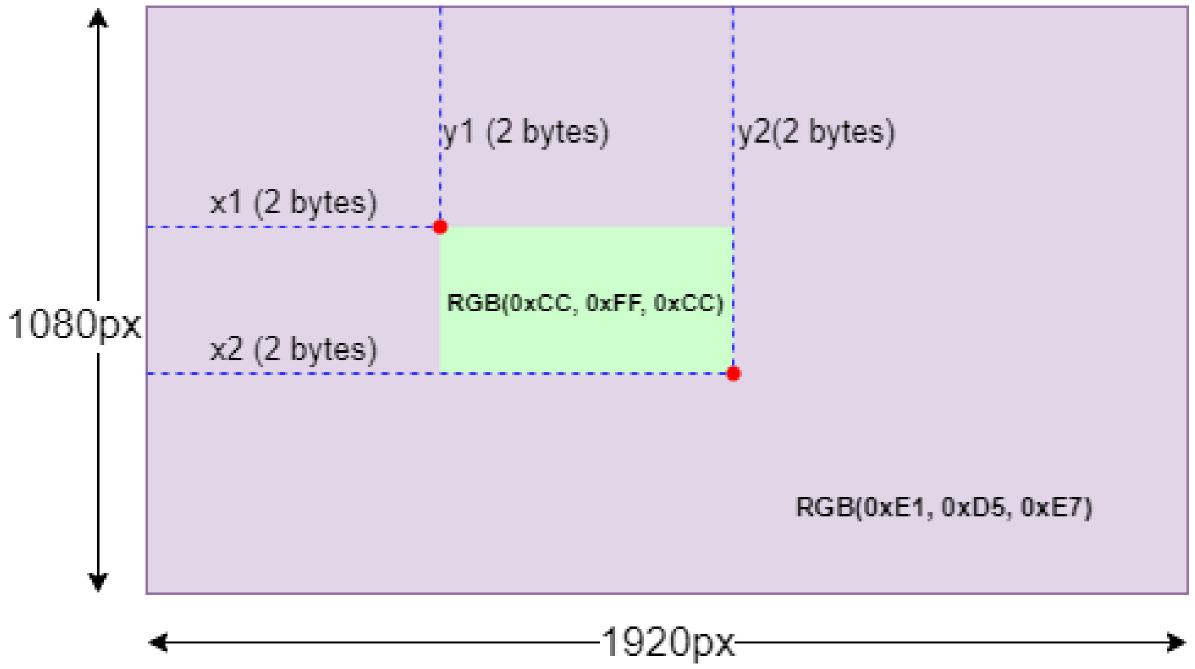


Figure 15: Display Generated Image diagram

On execution, the algorithm, first, fills the matrix with the background color values, afterwards, it goes through the whole matrix and checks if the coordinates are between the x values and the y values. If the previous condition is true, the algorithm fills that pixel with the RGB values for the quadrilateral. Upon finishing filling the matrix the algorithm then converts the matrix to a single dimension array to be dispatched to display driver to display the final image on the screen.

4 Implementation and Tests

This chapter discusses the aspects of the implementation, as well as, the validation tests of the developed application. First it is shown the implementation of the application and its entities. Next, the configuration of the AGL as a platform for the application. This chapter ends with the tests made on this dissertation project. The testing section shows the testing environment used and what tests were conducted on the project. This enables one to gauge the feasibility of the project when deployed on real hardware, and draw conclusions.

4.1 PDS Application

The setup used to develop and implement this system was a computer running Ubuntu Focal Fossa, since it's lightweight and offers a lot of flexibility and documentation. For the programming language, it was used C++ version 14 since it is supported by AGL, it is very powerful and flexible and one of the most used programming languages for embedded systems.

To implement the application, in accordance with the requirements, the solution reached was a core shared library and executable, with different modules implemented through shared libraries and fetching of those plugin libraries in runtime. This solution enables modularity since only the shared library for the plugin needs to be compiled to be added to the program. This modularity works by compiling the core modules into the executable and its library, and compiling the diagnostics and communication plugins as separate libraries. The core application and library is composed of the following entities, the Diagnostics Processor, the Diagnostics Lookup, Message Processor, and the Communication Interface. The plugins for the app are the individual diagnostics and the specific communication protocol plugin (for example serial communication plugin).

The basic structure of the program is a core DiagEngine class that contains and initializes all other classes. The main classes of the program called by the DiagEngine are, the UDSConfig class, that reads the ".conf"configuration files used to configure plugins, and configures the program accordingly, the UDS-Binding class, this class fetches and initializes the shared libraries in runtime, and, the UDSComLibrary class, this class configures the communication layer of the program and has a pointer to the abstract communication class.

The startup of the final iteration of the PDS application, as seen in Figure 16, is conducted as follows. When started, the application searches the predefined path for the configuration files, that contain the diagnostics names and DIDs, saves these values to a map to later be synced with the diagnostic libraries. Subsequently, the UDSBinding class searches a predefined path for the diagnostics libraries (in this case .so or shared object files), and proceeds to bind the map created by the configuration files with pointers to the diagnostics classes. Afterwards, the UDSCommLibrary starts the communication protocol, by searching a predefined path for the library of the communication interface wanted (for example, CAN or serial), and runs the communication thread. When all the previous steps are done, the application waits for UDS messages to arrive at the communication interface to execute the diagnostics. All this can be further visualized in the flowchart presented in the Figure 16.

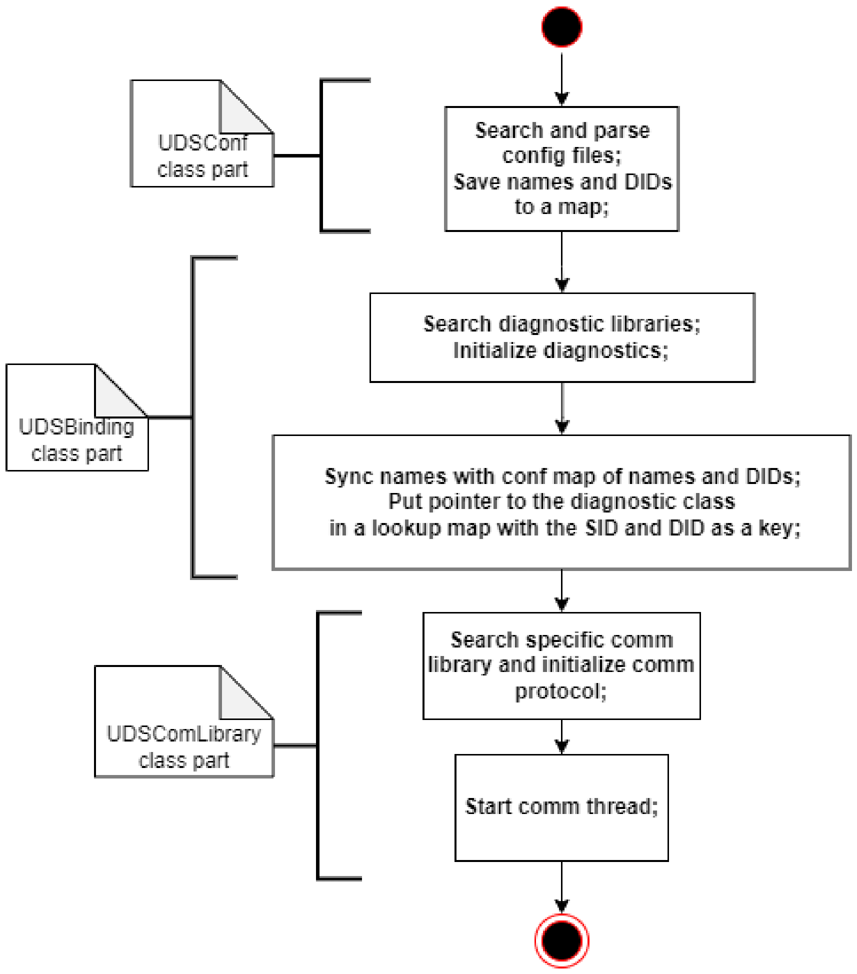

UDSConf class part
Search and parse config files; Save names and DIDs to a map;
Search diagnostic libraries; Initialize diagnostics;
UDSBinding class part
Sync names with conf map of names and DIDs; Put pointer to the diagnostic class in a lookup map with the SID and DID as a key;
Search specific comm library and initialize comm protocol;
UDSComLibrary class part
Start comm thread;

Figure 16: Application startup diagram

4.1.1 DiagEngine class

This class, is the main class of the application, used to initialize all the other modules of the application and to contain all the other classes.

For the initialization of the application, it starts with the configuration of the system by reading the diagnosis.default.conf file that includes other .conf files for the diagnostics, this ".conf"files for the diagnostics contain the diagnostic name and SID. First it is configured the supported diagnostics and their respective names and SIDs via the diagnosis.default.conf file. Afterwards, the search for the shared libraries of all the diagnostics and the binding of them to the lookup class used to store and fetch the pointers of the diagnostics. Lastly the configuration, initialization and start of the communication protocol, which can be either using CAN or serial console. All the configurations use user defined macros that contain the absolute path of the configuration folders, communication and diagnostics library paths.

4.1.2 Communication Interface

The implementation of the Communication Interface, taking the requirements into account, it is done with the usage of three classes and inheritance. The three classes are a base abstract class, and both the CAN communication class and the serial communication class. The specific communication classes, CAN and serial, inherit from the base class.

For the base abstract class the idea is to create an interface with the DiagEngine class and abstract it from the communication protocol, for this, the base class has functions to retrieve messages and to send messages, as well as an abstract initialization function. This way any class that inherits the base class has the control of what is done for the initialization, reception and relay of the UDS messages.

For the implementation of the CAN communication class, it is first done a configuration on the class constructor to open a socket and to configure the baudrate and datarate of the CAN communication, this ends by starting the execution thread. This thread calls the *recv* function from the Linux CAN libraries that waits until a message is received. After the message is received, it is copied to a byte vector that it is sent to the Scheduler using the Scheduler's *setRequest* function.

For the serial communication class, the process is similar, only that the serial communication does not need configuring and the thread uses different methods to retrieve information from the serial console. The contents from the serial console are retrieved using the C++ libraries function *fgets* and checking if the string is a digit or not and passing the ASCII digits to their value in bytes. This information is passed to a buffer that then is copied to a byte vector and sent to the Scheduler, using the *setRequest* function.

4.1.3 Message Processor

The implementation of the message processor, contrary to the other modules, is not a class but a set of simple lines of code to retrieve the relevant information from the UDS message. These lines of code use standard C++ functions to, first, remove the first byte which is the SID. Next, it checks if the SID extracted contains a sub-function parameter, if this checks true, it retrieves the byte as a Service parameter and proceeds to extract the two bytes that form the DID, if the condition does not check true, it proceeds right away to extract the DID. The SID and DID are stored in a data structure that contains the unique ID of the diagnostic. This data structure is what is used for the keys in the Diagnostic map and are used to search for the diagnostics.

4.1.4 Diagnostics Lookup

The Diagnostics Lookup module, is implemented as a class that is used as an interface for storing and retrieving diagnostics. The diagnostics are stored in a map, using a custom data structure that contains the SID and DID as a key and a pointer for the Base Diagnostic class as the elements stored. This class is run on the startup of the application to store the available diagnostics in the map, by using the configuration files explained before.

Since this class needs to be shared through most of the other application classes it is instantiated as a pointer that is passed to the other classes through the constructor, this enables the application to share the same map between different processes.

This class contains a C++ map to store the diagnostics, this map uses a custom structure to use as key and a pointer to the Diagnostic as the mapped value. The custom structure is a simple C++ struct that contains two variables, a *uint8_t* for the SID and a *uint16_t* for the RID. The class also contains functions to interface with the map, functions such as *getDiagnostic*, *addDiagnostic*, *clearMap* and *searchDiagnostic*.

The addition of the diagnostics in the map is made through a COTS library from Bosch that searches the filesystem for a .so library file. This is done by fetching the name given in the configuration file and searching for "lib[diagnostic name].so". The library then calls upon the *dlopen()* C++ function to initialize the shared object libraries. This ends with the library file calling a EntryPoint function, with the diagnostic class name and it's DID, that starts the diagnostic class and inserts the into the map by using the *insert()* function of the C++ map class.

It is important for the configuration file to have the same name as the shared object library and to have the same DID, since it is what is used to open the library and to index the diagnostic class correctly.

4.1.5 Diagnostics Processor

The implementation for this entity is a class that is composed of a standard C++ thread to execute diagnostics and a function to store diagnostics in a queue. The thread is used to continuously check a diagnostics queue for elements and proceed to call the execution of said elements, the queue's elements are pointers to the Base Service class and these elements are executed by a function from the Service Class.

The queuing of diagnostics is called by the Communication Interface module when a message is received. This synchronous function, called *setRequest*, takes the message and sends it to the message processor algorithm and Diagnostics Lookup entity to retrieve parameters and a pointer to the diagnostic. Upon receiving the data it is stored in a queue for diagnostics and in a queue for parameters.

The thread of execution of the scheduler first checks a atomic variable used to stop the thread. After this first check, it checks if the queue of diagnostics is empty, if it is not, it proceeds to pass the diagnostic to a smart pointer of the Base Diagnostic class called *CurrentExec* and pass the parameters to a vector and pops both queues. Next, it calls the method *onRequest* of the *RoutineControl* Class using the *CurrentExec* pointer that proceeds to parse the parameters for the sub-function and call the proper sub-function, either *onStop*, *onStart* or *onResult*, and ends with the execution and subsequent sending of the diagnostic response to a response queue. The diagnostic response is sent after the *onRequest* function returns. The thread of execution calls a class function called *responseHandle* that retrieves the response from the queue and calls the Communication Interface function *sendResponse* and gives the response vector as a parameter.

4.1.6 Diagnostics

The implementation of this entity follows a stripped down version of the design proposed in the previous chapter, as seen in Figure 14. This was done because all of the implemented diagnostics use the same service, the Routine Control. In the end, the implementation contains the Base Service class, a single Service and all the diagnostics.

The Base Service implemented is called UdsService and contains basic functions to get the SID, initialize itself and purely virtual functions that other classes will inherit. The class for the only service is called *RoutineControl*. It implements a *onRequest* function, that is as an entry point and is used to parse the sub-service parameter, and also the methods of start, stop and request result. Each diagnostic will, in turn, implement their specific start, stop and request result functions.

The implementation was done this way to make the program only store a lookup map composed of the SID and DID as keys and a pointer to a Base Service class. This implementation takes leverage of the C++ inheritance and makes the program much more flexible, enabling not only the storage of Routine Control services but other services, such as Session Control, by using the abstract class Service. The relationship, methods and variables of the *UdsService*, *RoutineControl* and a example diagnostic can be seen on the Figure 17.

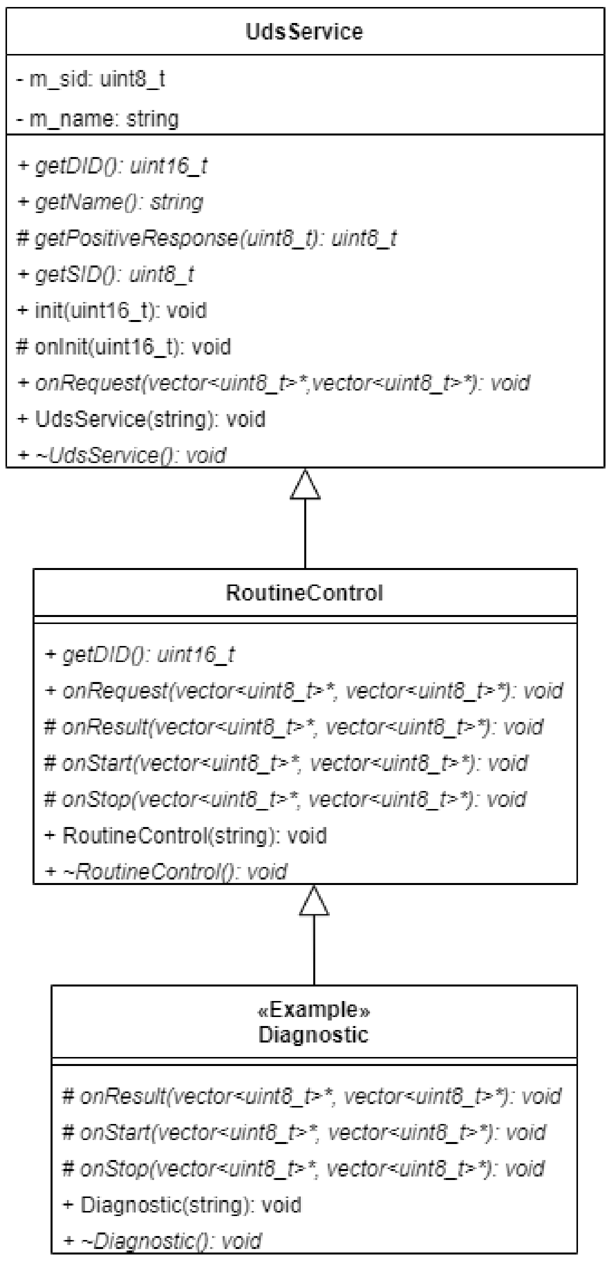

UdsService
- m_sid: uint8_t
- m_name: string
+ getDID(): uint16_t
+ getName(): string
# getPositiveResponse(uint8_t): uint8_t
+ getSID(): uint8_t
+ init(uint16_t): void
# onInit(uint16_t): void
+ onRequest(vector<uint8_t>*,vector<uint8_t>*): void
+ UdsService(string): void
+ ~UdsService(): void
RoutineControl
+ getDID(): uint16_t
+ onRequest(vector<uint8_t>*, vector<uint8_t>*): void
# onResult(vector<uint8_t>*, vector<uint8_t>*): void
# onStart(vector<uint8_t>*, vector<uint8_t>*): void
# onStop(vector<uint8_t>*, vector<uint8_t>*): void
+ RoutineControl(string): void
+ ~RoutineControl(): void
«Example»
Diagnostic
# onResult(vector<uint8_t>*, vector<uint8_t>*): void
# onStart(vector<uint8_t>*, vector<uint8_t>*): void
# onStop(vector<uint8_t>*, vector<uint8_t>*): void
+ Diagnostic(string): void
+ ~Diagnostic(): void

Figure 17: Diagnostic, Service and RoutineControl class diagram

Read MACAddress

The implementation of this diagnostic, uses the Linux way of presenting the hardware as files in the filesystem. The diagnostic, when *RoutineControl* is called, consists of searching the Linux filesystem for the given Ethernet interface files (in this case searching "/sys/class/net/"interface name"/address"), copy the address to a temporary file, check if it is in the correct format and ending by sending the response back with the positive response plus the address. The result should come back as "71 01 FD 10 Address" in hexadecimal.

Read eMMC CID

As other diagnostics, this diagnostic takes leverage on the Linux presentation of the hardware as files in the filesystem. This diagnostic searches the hardware files for the eMMC registers and parses the CID register of the eMMC. Furthermore the diagnostic divides the value into its components, for example, manufacturer ID and production date, and sends it in the response. The result of this diagnostic is a positive answer with the CID as the parameters of the response.

Read and Write GPIO

For this diagnostic it is used the libgpiod library that comes with AGL as an extra package. The libgpiod library offers a great interface with the Linux hardware functions, like ioctl, and offers two ways to control the GPIO pins, either by full configuration by the developer and with utilization of a object pointing to the GPIO controller, or, a contextless way with simpler configuration and no objects. The solution chosen was the contextless way, since the requirements of this diagnostic is to simply test if certain lines are working and not to block the GPIO pin for the scope of the diagnostic.

The read GPIO diagnostic starts by checking the number of parameters received, then it parses the parameters, the first one being the gpiochip number from 1-8 and the gpiopin, after this it calls the *gpiod_ctxless_get_value()* function with the parameters received and it returns the GPIO value of the pin in the gpiochip and sends the value back to the communication.

The write GPIO diagnostic is similar to the read but receives an extra parameter, the value of either 1 or 0 to set the pin to. The procedure is also similar, it checks the number of parameters, parses the parameters and then checks if it can set the value by using the *gpiod_ctxless_set_value()* function, if all goes well it sends the value it sets the GPIO pin back to the communication interface.

Read and Write Memory

In this diagnostic it is used the basic Linux ioctl and memory read libraries to read a certain memory address and return its values, it is the simplest of diagnostics. It consists of opening the RAM file in the Linux filesystem, using fopen functions, and using the ioctl function controlling the memory either to read or write to it.

The read function, as said above, opens the RAM file in the Linux filesystem, and uses ioctl to open the ram as read-only, after that the contents retrieved from the address are sent in the parameters.

The write function is similar to the read, only it writes to the RAM. It starts by opening the RAM file and set it to read and write. After this, it uses a pointer to write the sent values to the address specified in the parameters.

Play and Stop Audio

This diagnostic uses as a base the Advanced Linux Sound Architecture (ALSA) library. It was created a abstraction class to setup, play audio and control the audio driver through ALSA, this class is initialized inside the audio diagnostic as a object named "ctl". The created class basically uses the ALSA Pulse-code Modulation (PCM) functions to open the default audio device and setting it up to receive a sine wave at the tester specified frequency that is given in the parameters, the volume is controlled by the ALSA library functions also. The most important functions of this class are the Audio_Setup, Audio_Start, Audio_Stop and the thread to send the sine to be played(called audio_output). The Setup function opens the device and pcm_handle and setups the sample frequency and other parameters. The Audio_Start function checks if the thread is configured correctly, sets the audio volume and calculates the sine wave and starts the thread. The Audio_Stop function checks if the thread is running and, if so, sets the stop_thread variable to true and joins the thread to wait it to finish and ends by dropping the pending sine wave values. The thread, audio_output, is basically infinitely sending the values of the calculated sine wave stored in a buffer to the pcm_handle with the usage of the snd_pcm_writei function, this function receives the handler for the audio device, a buffer of values to be sent and the size of the values.

When a start routine signal is received, through the communication, the diagnostic proceeds check if it received the correct number of parameters, in this case 3 parameters(frequency MSB, frequency LSB and volume), and checks if the setup of the audio stack was done properly. Afterwards, checks the value of the volume to truncate it between the 0 and 100% values. Next, it starts to calculate the sine wave values to put into the buffer and after this it calls the Audio_Start function from the AlsaCTL class to start playing the audio and sends a positive response code.

When a stop routine signal is received, in the communication, the diagnostic checks if the audio stack is properly configured and calls the Audio_Stop function form the AlsaCTL class to stop the audio playing and sends a positive response.

Show Generated Picture and Test Picture

The Generate Picture diagnostic upon entering it calculates and creates a quadrilateral in the screen with the top left x and y parameters and the bottom right x and y parameters, the other parameters are the background colour and the quadrilateral colour. All this is using basic arithmetic's to calculate the size of the quadrilateral and the pixel offsets to put it in the right position. The result is a byte array containing the raw binary pixel colours to be display and sent to the Display Control Wrapper function for writing to the display.

The Test Picture is similar to the Generate Picture, only it needs to convert a .png file stored in a location in the filesystem into raw binary pixel data. To explain this better let's start with the searching the .png file.

The search for the .png is made by giving the pictures in the specified directory, in this case "/usr/local/TestPictures/", a index in their name(i.e. "1-TestPicture.png"), this is important since the image to be displayed is given by a index from the UDS message parameters, the Linux library dir functions are used to search all the available indexes in the directory and parse them to a array. This array is then used to retrieve the specified image. Once the specified image is retrieved it's passed from a .png folder to a raw binary representation via the png library available in Linux. Afterwards it's passed into the Display Control Wrapper write function effectively displaying the picture.

Display Control Wrapper

As the implementation progressed the need for a unified display control driver arose, this was done in order to not write the same code for both the display diagnostics and prevent concurrent accesses to the display.

For the display control there is COTS implementation of Bosch that was taken as a base. It wasn't possible to use it, as is, in AGL since the COTS was made for QNX and utilized drivers and functions not available in Linux, but it's core design and functions are very similar to what is possible and available in Linux.

The display class is implemented as a singleton to prevent multiple objects and to prevent the addressing of the framebuffer in different locations, it offers a simple interface for receiving what's supposed to be displayed, a picture in a raw array format. The way it's implemented is, with the fb.h header we have access to the framebuffer structure, soo we start by using the open function to open the framebuffer file, with ioctl function we get the framebuffer information into the fb_var_screeninfo structure, with all this it proceeds to memory map the framebuffer, this allows to memset the framebuffer with the intended information.

The wrapper will function as follows, first it acquires and opens all the available framebuffer devices of the system and uses their number to index them in a array (i.e. "fb0"in the filesystem is indexed in the position 0 of the array). The diagnostics will then call a function to retrieve the instance of the framebuffer they need (i.e. framebuffer in index 1 or 2).

The wrapper offers the initialization of the framebuffer devices, by utilizing the ioctl functions of Linux to retrieve screen size and number of bytes per pixel, and maps the device to memory so it's not using the framebuffer memory directly and enable seamless and easy control of the pixel buffer through a pointer.

The displaying of content in the framebuffer is done through write functions that copy, line by line, the contents of a raw image buffer to the pointed framebuffer mapped memory. The wrapper also offers functions to set background and foreground color to abstract the Display Generated Image diagnostic.

4.2 AGL Image

The AGL image built is based on the AGL jellyfish version 10.0.3 for the Renesas RCar H3 Salvator-XS board. The Renesas RCar H3 Salvator-XS board is a development board based on the RCar H3 SoC, this processor is a nine core processor with four cortex A-57, four cortex A-53 and a cortex R7 core, this board was used because of it's availability and it's similarity with a custom board developed by Bosch that it's based on this development board.

To explore more the AGL build environment and capabilities a yocto layer was created. The layer is based on the already available layer for the Renesas RCar platform provided by Renesas to AGL, but, with less clutter, less configurations and only supporting the development board used for this dissertation. In the end the layer provided the capability of building a image, based on a pre-configured default image, for the development board with the CAN drivers and interface activated and the libraries necessary for compiling and use the PDS application.

The base image choice that the created layer provides is based on the ”agl-cluster-demo”image. This image is a very good starting point with most of the functionalities needed for the project, but it needed some fine tweaks. It was added the option of having the GPIO pins present on the filesystem to enable the configuration of the Control Area Network (CAN) hardware module through scripts provided by Renesas. It was also added to the image the libgpiod library, this enables the implementation of simpler GPIO controls through a abstraction layer provided by that library. After the GPIO modifications, it needs to have enabled the support for Control Area Network (CAN) communication and to have enabled the specific System-on-Chip (SoC) CAN device drivers, which is done in the kernel configuration menus.

4.3 Tests

This is dedicated to the tests and the results of the tests conducted on the system. First there's a brief exposition of the test environment used, what tools and software was used to conduct said tests. After the tests done are shown and explained, it's exposed the results of the tests, to later be discussed in the Conclusion chapter.

It's important to keep in mind that this dissertation project is a proof-of-concept, so its tests are mostly functional with little to no real world test conducted, since it's difficult to do something in such a embryonic solution.

4.3.1 Test environment

The test environment used was the CANoe application by Vector running on a Windows machine. This application is used through out Bosch to simulate CAN devices and busses and the interactions between the hardware and a simulated car. The application offers a UDS diagnostics test framework to introduce the diagnostics the tester wants to run on the target and their respective responses. This tool was used since it has a simple configuration interface and it supports the CAN interface hardware available.

The usage of a CAN simulation application is not enough, it is also needed a interface to physically connect to the hardware and to transform the messages into CAN messages. For this, it is used the Vector VN1630A CAN interface. This interface supports the transmission and reception of CAN and CAN Flexible Data-Rate (CAN FD).

To configure the test environment, first it was configured the diagnostics messages to be sent and their positive responses. Then it was created a BasicDiagnosticECU, which is a virtualization of the ECU under test and it's used to send the diagnostics to the board and to receive the responses. Afterwards it was configured the CAN timings, baudrate and datarates of the CAN bus. The chosen baudrate and datarate are 500000 and 2000000 bits per second. After, it was needed to enable CAN FD and choose the CAN IDs.

The test environment connections is as follows, the VN1630A is connected to the computer via Universal Serial Bus (USB) and to the board via a 9 pin DSUB connector. The next figure 18 is a diagram of the test connections.

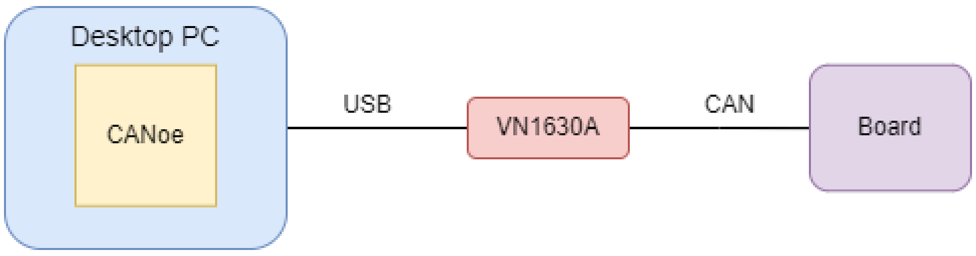


Figure 18: Test environment connection diagram.

4.3.2 Tests and Results

Since the project is a proof-of-concept the tests conducted on the board are mainly functional tests, to check if the implementation does what is supposed to and functions in accordance with the requirements and design. First it will be explained the tests conducted to the communication of the system with the test environment. Afterwards it will be shown the tests made for each of the system's modules, like the Scheduler. The tests end with the tests conducted to the individual diagnostics.

Communication tests

For the communication tests it was used a set of applications for Linux CAN subsystem, the CAN-utils, as well as the CANoe application, for testing the sending of CAN messages from the board to the tester computer. CAN-utils offers many utilities, the ones used for the test where cangen, candump and cansend. The cangen and cansend are used to send CAN messages to the bus, cangen sends random messages and cansend sends a message specified in it's parameters. Candump is used to dump received messages to the terminal. These utilities where used to test the correct configuration and connection of the CAN interface.

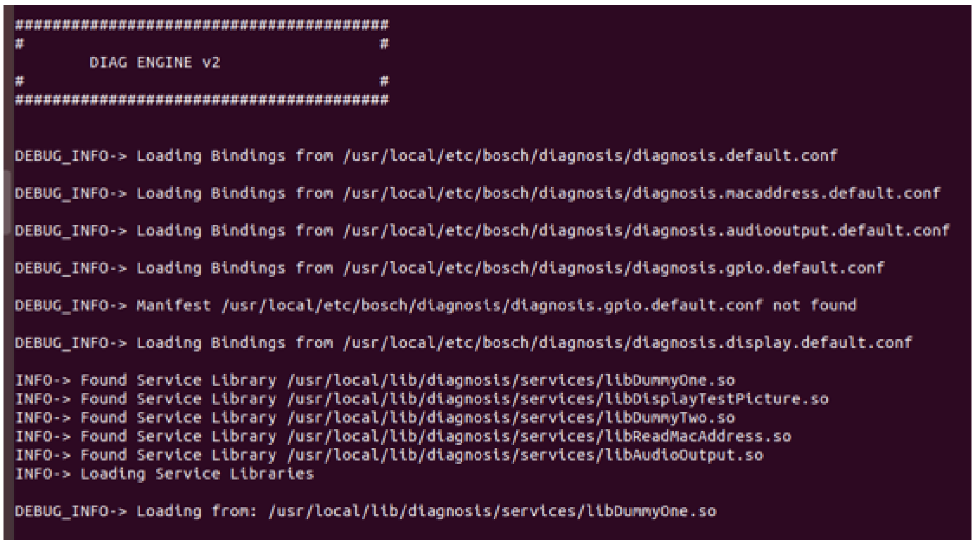
The results of this test were good and as expected. The CANoe application on the tester computer received all of the messages sent from the board and the board also received and displayed the messages received from the tester computer correctly on the command line, with the use of the candump application.

The last tests where also on the full system, but this time, testing the diagnostics itself. The diagnostics where tested to know if the execution is compliant with their requirements and tested against errors. Some diagnostics tests where short and others more intensive. To show better the tests conducted on each of the diagnostics, the next part goes into each of the diagnostics and shows what was done.

System Tests

This tests simulate the reception of diagnostic messages and the initialization of the system. The system was tested to know if it interpreted the message correctly, what would happen if it received a non compliant message or unknown SID or DID. The initialization tested the system against lack of libraries for each of the components, lack of configuration files and what would happen if it tried to fetch a diagnostic without it's library present in the filesystem.

The results of the tests mediocre, the system, when poorly configured or lacking it's base libraries, doesn't initialize, as expected. On the other hand, if the system initializes it's base libraries but lacks the component libraries it enter a error state and doesn't recover, which is non ideal. For the reception of non compliant messages or unknown SID or DID the system performed well, it returned a error message to the tester showing that either the message is incorrect or the IDs are not supported by the system. In the Figure 19 it can be seen a snipped of the output of the initialization of the application.



```
########################################
#                                      #
        DIAG ENGINE v2
#                                      #
########################################


DEBUG_INFO-> Loading Bindings from /usr/local/etc/bosch/diagnosis/diagnosis.default.conf

DEBUG_INFO-> Loading Bindings from /usr/local/etc/bosch/diagnosis/diagnosis.macaddress.default.conf

DEBUG_INFO-> Loading Bindings from /usr/local/etc/bosch/diagnosis/diagnosis.audiooutput.default.conf

DEBUG_INFO-> Loading Bindings from /usr/local/etc/bosch/diagnosis/diagnosis.gpio.default.conf

DEBUG_INFO-> Manifest /usr/local/etc/bosch/diagnosis/diagnosis.gpio.default.conf not found

DEBUG_INFO-> Loading Bindings from /usr/local/etc/bosch/diagnosis/diagnosis.display.default.conf

INFO-> Found Service Library /usr/local/lib/diagnosis/services/libDummyOne.so
INFO-> Found Service Library /usr/local/lib/diagnosis/services/libDisplayTestPicture.so
INFO-> Found Service Library /usr/local/lib/diagnosis/services/libDummyTwo.so
INFO-> Found Service Library /usr/local/lib/diagnosis/services/libReadMacAddress.so
INFO-> Found Service Library /usr/local/lib/diagnosis/services/libAudioOutput.so
INFO-> Loading Service Libraries

DEBUG_INFO-> Loading from: /usr/local/lib/diagnosis/services/libDummyOne.so
```

Figure 19: Terminal output of the initialization of the application

Diagnostic Tests

This test is characterized by checking the individual diagnostics supported by the system against bad received information or some midway problem with the information that it's passed between entities. This also tests all the interactions in the system, starting at receiving a message and ending in calling a diagnostic. This is done to check the interaction between the entities, how the modules perform and what happens if a error occurs in the middle of execution.

Read MACAddress

For this diagnostic the tests where simple, just test what happened if the diagnostic receives parameters or the ethernet interface doesn't exist. The results are good and as expected, the diagnostic returns the UDS protocol error code for incorrect message length when the number of parameters is higher than the needed number. When the ethernet interface or any processing between the acquisition of the MACAddress and the return of it fails the system correctly returned the UDS protocol error code for ”conditions not correct”.

Read eMMC CID

This diagnostic is like the Read MACAddress one, just check what happens if the diagnostic receives parameters when it should receive none and one extra test check if the response message contains the exact same values as the CID register. The results are good, the diagnostic sends the correct negative responses to the incorrect number of parameters and if the CID is not available.

Read/Write GPIO

The tests for this diagnostic check the parameters and test the design and implementation against errors of reading or writing pins that are locked by other programs or processes or do not exist. The results are as expected, like read MACAddress the diagnostic responds correctly to a incorrect number of parameters and responds correctly to non existent GPIO pins, thanks to the libgpiod API error handling. When a pin is locked by other processes of the system the diagnostic also responded correctly with the ”conditions not correct”negative code, due to the libgpiod API excellent error handling.

Read/Write Memory

The Read/Write Memory tests are checking what happens when restricted memory is read or written to and check the outcome of write by reading the value after the write operation. The results are unexpected and bad, the system could not recover and reset when a restricted memory address was accessed in read or write, this can be a problem with the algorithm used as well as the functions used to retrieve the contents of the memory. On the other hand, when something is written into a address the reading of that same address returned the contents written to it.

Play/Stop Audio

In this diagnostic, the test where about what would happen if the interface is not correctly configured, what happens if the thread doesn't start and what would happen if the audio could not be stopped. The results of this tests are good and bad. If the diagnostic fails at configuring the interface the system responded with the correct negative code and the system didn't crash, as well as, when the thread could not be started the response was correct. On the other hand, if the audio thread somehow failed, the system would play the audio indefinitely and could only stop when a reset is triggered.

Display Test Picture

This diagnostic tests are straight forward, connect the display and see what's outputted, but also, check the outcome of searching a non existent image index and what happens if the image is too big for the screen. The results are as expected, the diagnostic when a out of bounds index is sent returns the UDS error code "conditions not correct". When a selected image is bigger than the display, which a occurrence that should not happen since the developer controls the pictures present in the system, the image is displayed, although, it gets cut since it has more information than the display can output. The Figure 20 shows the display output of a given image and the Figure 21 shows some debug output of the diagnostic being called.

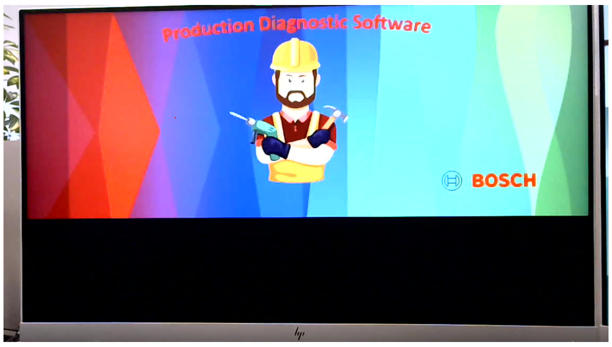
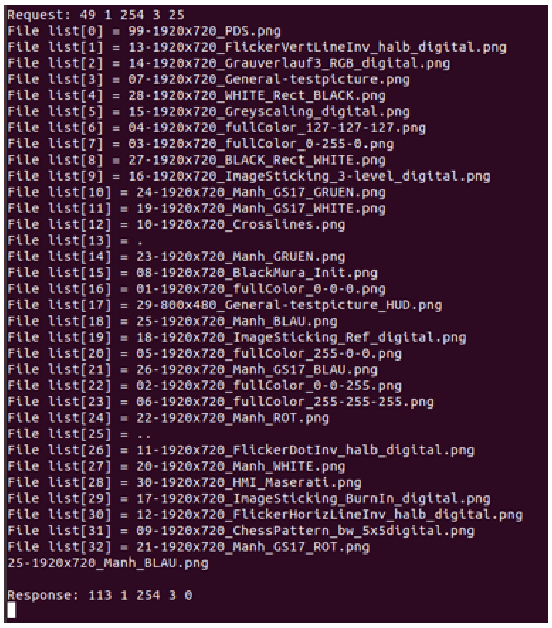


Figure 20: Display output to a give image



```
Request: 49 1 254 3 25
File list[0] = 99-1920x720_PDS.png
File list[1] = 13-1920x720_FlickerVertLineInv_halb_digital.png
File list[2] = 14-1920x720_Grauverlauf3_RGB_digital.png
File list[3] = 07-1920x720_General-testpicture.png
File list[4] = 28-1920x720_WHITE_Rect_BLACK.png
File list[5] = 15-1920x720_Greyscaling_digital.png
File list[6] = 04-1920x720_fullColor_127-127-127.png
File list[7] = 03-1920x720_fullColor_0-255-0.png
File list[8] = 27-1920x720_BLACK_Rect_WHITE.png
File list[9] = 16-1920x720_ImageSticking_3-level_digital.png
File list[10] = 24-1920x720_Manh_GS17_GRUEN.png
File list[11] = 19-1920x720_Manh_GS17_WHITE.png
File list[12] = 10-1920x720_Crosslines.png
File list[13] = .
File list[14] = 23-1920x720_Manh_GRUEN.png
File list[15] = 08-1920x720_BlackMura_Init.png
File list[16] = 01-1920x720_fullColor_0-0-0.png
File list[17] = 29-800x480_General-testpicture_HUD.png
File list[18] = 25-1920x720_Manh_BLAU.png
File list[19] = 18-1920x720_ImageSticking_Ref_digital.png
File list[20] = 05-1920x720_fullColor_255-0-0.png
File list[21] = 26-1920x720_Manh_GS17_BLAU.png
File list[22] = 02-1920x720_fullColor_0-0-255.png
File list[23] = 06-1920x720_fullColor_255-255-255.png
File list[24] = 22-1920x720_Manh_ROT.png
File list[25] = ..
File list[26] = 11-1920x720_FlickerDotInv_halb_digital.png
File list[27] = 20-1920x720_Manh_WHITE.png
File list[28] = 30-1920x720_HMI_Maserati.png
File list[29] = 17-1920x720_ImageSticking_BurnIn_digital.png
File list[30] = 12-1920x720_FlickerHorizLineInv_halb_digital.png
File list[31] = 09-1920x720_ChessPattern_bw_5x5digital.png
File list[32] = 21-1920x720_Manh_GS17_ROT.png
25-1920x720_Manh_BLAU.png

Response: 113 1 254 3 0
```

Figure 21: Terminal output of some debug lines showing the available images

Display Generated Image

The Display Generated Image tests include check the displayed image if it's correct with the parameters, check the output of parameters out of range and check if what happens if the Display Control Wrapper is not working properly. The results of this diagnostic tests are great, the diagnostic deals with the corner cases as expected. If the Display Control Wrapper is not working the diagnostic since it could not retrieve a non null pointer throws a error and a negative UDS response, as expected.

5 Conclusions

This is the closing chapter of this dissertation. Here, the project will be wrapped up and the results to the tests will be discussed to find an answer to the dissertations questions.

Currently in the automotive world of Bosch the usage of a open source Linux based OS like AGL can be of high interest. The company can reduce the costs of using closed source options and have a more flexible solution due to open source nature. The solution here presented, a PDS application running on AGL, has proved to be very promising in this field of production diagnostics.

5.1 Conclusions

The purpose of this dissertation project was to create a proof-of-concept Production Diagnostics System that uses AGL as a OS, in order to be a competing platform to the solution currently used by Bosch. Nowadays, the existing PDS applications at Bosch are supported by a QNX on AUTOSAR stack. This dissertation aims to verify if it is possible to remove this software stack and replace it with AGL and also, it aims of exploring the automotive features of AGL, for example the implementation of a diagnostics engine compliant with UDS.

AGL, as a Linux project, offers high levels of customization, as well as a repository of open-source software packets and COTS modules to support development of new applications. On the other hand, AGL being a recent technology does not follow some automotive standards and lacks the code base and maturity for certain functionalities when comparing with other operating systems like QNX.

A proof-of-concept was made, in this dissertation, of the use of AGL to support the implementation of a UDS compliant application interacting with AGL. The proof-of-concept developed integrated a PDS application with AGL. This system is capable of receiving UDS requests, execute Production Diagnostics and send back UDS responses to the tester.

The finding of this dissertation is that the proof-of-concept developed cannot replace the current solution used by Bosch. This is because AGL doesn't offer the same features, have the same behaviour and be compliant with the same automotive standards that Vector AUTOSAR offers, this can be attributed to the fact that AUTOSAR is a RTOS with low level hardware interfaces and a ecosystem made for automotive development.

On the other hand, AGL can be seen as a competitor to QNX in this software stack. AGL being a open-source Linux project, aimed for the automotive industry, offers a cost reduction and higher customization when compared to QNX. The challenge, however, is to expand its code base and reach a higher level of maturity to be viable option to future solutions and replace QNX.

5.2 Future Work

As mentioned above, AGL is a viable solution for incorporating the PDS support software stack.

Although the PDS application developed in this dissertation is functionally correct, it needs to implement a good error handling framework and mechanisms to prevent system lockups in certain corner cases. The application also needs testing against all the automotive ISO specifications, specially ASIL. Other good features to implement in late iterations are, data logging to help tackle errors and understand what the system is doing, more services implemented to increase the number of diagnostics and a fully functional UDS specification framework.

AGL OS it is a promising alternative to QNX on the software stack. For the future it will be very important to explore the integration of AGL into the software stack, taking the place of QNX.

Bibliography

[1] R. Ramya, S. Sripada, and K. B. Binu. “Breakthrough approach to automotive diagnostic verification”. In: *SAE Technical Papers* (2011). issn: 26883627. doi: 10.4271/2011-28-0030.

[2] Y. Li, D. Zhan, and H. Li. “Eol testing method based on ecu diagnostic function”. In: *FISITA 2016 World Automotive Congress - Proceedings*. 2016.

[3] E. Bringmann and A. Krämer. “Model-based testing of automotive systems”. In: *Proceedings of the 1st International Conference on Software Testing, Verification and Validation, ICST 2008*. 2008, pp. 485–493. isbn: 076953127X. doi: 10.1109/ICST.2008.45.

[4] I. Standard. “INTERNATIONAL STANDARD Road vehicles — Unified diagnostic”. In: 2006 (2008).

[5] J. Knox. *Automotive grade linux platform debuts on the 2018 toyota camry*. 2017. url: https://www.automotivelinux.org/announcements/automotive-grade-linux-platform-debuts-on-the-2018-toyota-camry/.

[6] J. Mössinger. “Software in automotive systems”. In: *IEEE Software* 27.2 (Mar. 2010), pp. 92–94. issn: 07407459. doi: 10.1109/MS.2010.55.

[7] S. Aust. “Paving the way for connected cars with adaptive AUTOSAR and AGL”. In: *Proceedings of the 43rd Annual IEEE Conference on Local Computer Networks, LCN Workshops 2018* (2019), pp. 53–58. doi: 10.1109/LCNW.2018.8628558.

[8] P. Sivakumar et al. “Automotive Grade Linux Software Architecture for Automotive Infotainment System”. In: *Proceedings of the 5th International Conference on Inventive Computation Technologies, ICICT 2020*. 2020, pp. 391–395. isbn: 9781728146850. doi: 10.1109/ICICT48043.2020.9112556.

[9] M. O’Donnell. *Automotive telematics: Open-source automotive-grade linux is the future*. July 2005.

[10] AGL. *Automotive Grade Linux*. 2014. url: https://www.automotivelinux.org/.

[11] S. Mizuno et al. “Real-time Performance of Embedded Platform for Autonomous Driving Using Linux and a GPU”. In: *International Journal of Informatics Society* 10.1 (2018), pp. 31–40. url: `http://www.infsoc.org/journal/vol10/IJIS%7B%5C_%7D10%7B%5C_%7D1%7B%5C_%7D031-040.pdf`.