Thesis Dissertation

AN OBD DATA MONITORING PLATFORM IN ANDROID AUTO

Andreas Savva

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

January 2021

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

An OBD Data Monitoring Platform
in Android Auto

Andreas Savva

Advisor
Associate Professor Demetris Zeinalipour

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

January 2021

Acknowledgments

First of all, I would like to thank my advisor, Mr. Zeinalipour, who has given me the opportunity to study a subject that intrigued me. It was an honor to be working with a professor of his magnitude and through the course of this thesis, I learned a lot from him. Through his recommendations and guidance, I was able to complete my Thesis and learn about aspects of computer science that previously I could not reach.

I would also like to thank my family, who, during this period were always by my side supporting me and making me stronger. My family played one of the most important roles in helping me complete my Thesis and keeping my faith high even through the worst of times.

Lastly, a big thank to all of my friends who have been by my side all this time. My friends have not only given me support but they have also helped me by giving me recommendations through reading this Thesis. In addition, without my friends, I would not have been able to do real-world testing of the application developed in this Thesis.

Abstract

Nowadays, cars get more advanced with the use of modern computer technology. Cars now contain multiple Electronic Control Units (ECU) that are used for many different reasons. Apart from that, innovation continues with the development of the connected car to the grid and other cars. The car today is a machine that requires multiple thousand lines of code in order to function.

Now, although cars contain more software than ever before, information is hidden from the driver. Drivers and passengers want to see insights, analytics, and information about the vehicle, but this has been traditionally hidden by the manufacturers. In addition, using smartphones while driving is extremely dangerous and prohibited. The only solution to this problem is a 3rd party application that could run on an infotainment system and show all the vehicle data a driver may want while offering customization.

In this thesis, our goal has been to learn about how the modern car uses all this code and what it has to offer. We would like to know what the available systems are, so that we can develop code and improve the experience, not only for the driver but for the passengers as well. We have chosen to create an application for a modern system called Android Auto. Our application runs on an Android smartphone and connects to the car through an OBD port and allows the user to project the vehicle's information on the infotainment system. We believe that this application is the solution to the hidden information problem in modern cars.

Through our process, we have found that this application is very simple to setup and use. It provides all the necessary information of the car and once setup, there is no need to use the phone at all. In addition, it allows users to create a custom list with their choices of vehicle properties so as to make it easier to find desired information. But of course, it does not come without drawbacks since it cannot display information in more custom lists or allow the user to customize the objects projected. Lastly, there are small delays in receiving information from the vehicle meaning that data are slightly late.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In recent years, we have seen car technology advance rapidly. Car manufacturers have implemented the use of modern computer technology and are now using the processing power available to them in order to improve and evolve the car as we know it.

Nowadays cars use many different computers to control various components, offer luxury and comfort, and most importantly safety. But as the car is an ever-changing machine, it has now come to a point where the computer systems that are shipped with it are incredibly complex and may be even considered as the main selling point. We are now in an era when the car will be able to connect to the grid, receive and provide information to enhance the experience, provide suggestions, and take critical decisions. We are moving towards a future where the driver will have less impact on the journey than the software used in the car as autonomous driving technology is advancing and data from all the different parts of the car are available to manufacturers to use for improving the systems.

In addition to this, we are now at a critical pivot point where the car is shifting from the traditional combustion engine to the use of electric power and renewable energy sources. This has a significant impact on the process because the whole vehicle is designed from the ground up because it needs to cooperate with new systems in place.

Furthermore, traditionally information is hidden in vehicles. Drivers and passengers want to view more information and analytics about their cars. Manufacturers hide this information and the only way to extract it is by using OBD2 devices connected to phones. This is extremely dangerous and against the law while driving because of high safety risks.

These are some of the reasons that moved us to study the new technologies put into our everyday means of transport and also develop an application that is safe to use while in a vehicle to view all the information you may want. Our application works on Android Auto and projects all the information on the infotainment system in a non-distractive way.

1.2 Thesis Overview

Our goal is to study the available automotive operating systems and find their capabilities. By doing this, it would allow us to understand the main characteristics of each system, what the requirements are for this type of environment, and what the focus is for the future. In addition to this, we want to learn about the protocols that are going to be used in order to implement the exciting new features of the car, the governing bodies that regulate these features, and the security concerns and tactics. This is especially important, because, as we all know the car is governed by strict policies not only in the European Union but everywhere in the world. On the one hand, these policies restrict the ability of the manufacturers to create something completely new and revolutionary as they need to follow certain guidelines but on the other hand, the provide safety, not only to the customers but the entire world as the car is the most usable means of transport.

Furthermore, we have been intrigued by the idea of developing an application that can potentially be used in such a modern system and that would allow us to understand

the magnitude of the size of data being produced by the car but also potentially shared with the manufacturers in the concept of the connected car.

The application we have developed can be used with Android Auto, which is a system that we will analyze in detail and is named Auto Insights Dashboard. It allows the user to connect to an OBD dongle and project all the useful data of the car to its android auto compatible infotainment system for monitoring. Also, this application allows us to understand all the limitations and concerns of the companies to ensure travelers' safety.

1.3 Thesis Contribution

This thesis will allow future work for the development of software running on vehicles by providing useful information about modern systems in cars and how one can develop for them. During my thesis project, I made the following two important contributions:

1. Studied and analyzed modern systems used in cars today and showed how applications can be developed for them.

2. Designed, developed, and tested a useful application that runs on a modern system found in cars today that allows users to view their car’s information through an OBD2 device.

Also, during this thesis, we carried out a user study to evaluate the effectiveness and functionality of our proposed application. Our selected users were asked to test our application on their Android Auto enabled cars while following the Highway Code and staying focused on the road. Our users then generated a small report outlining the positive and negative points of the application. Through this study, we found that our application fulfills all the functionality desired by the user but needs more customization to allow for smaller lists of data to minimize the amount of time needed to navigate through a list and find the desired property along with its value.

1.4 Thesis Outline

This Thesis begins with Chapter 1, where we give a brief introduction in order to give motivation and explain what the main point of the project was. This will allow the

reader to understand the subject, difficulties, and how the information can be used in the future. It also contains a brief overview of the following chapters.

In Chapter 2, we will begin by introducing the main concepts used in the project. This will include all the related work done during the project and the background needed in order to continue to the next chapters. The reader will get familiar with the modern systems in cars and how those can be used for development. It provides information for software and hardware included in cars and also how one can ensure the code running in cars is safe.

Chapter 3, “Android Auto”, aims to give a better view of the system used in this Thesis to design and develop an application to be used in modern cars. It gives the architecture of the system, analyses the different types of applications that can be developed while giving sample applications for understanding them, and gives a look at what is available in the market for applications on this system.

In Chapter 4, we introduce the application we developed in this thesis. In this chapter, the reader will learn about what the motivation for our application was, how the UI works both on the smartphone and on the car.

In Chapter 5 we will explain the architecture of the application. We break the application into different layers, explain how each layer works and how the classes of the application work together to deliver the end result, and work with the system we selected.

Chapter 6 will show the reader how AID was evaluated. The reader will learn how the evaluation was done both on the computer and in the real world. We will show the results and explain them.

Finally, in Chapter 7 we will draw our conclusions from the Thesis. This will include everything covered in the project and what the future work could be in order to expand and improve the application and what could be done with the knowledge gained for other automotive systems introduced.

Chapter 2

Related Work & Background

2.1 Technological Background

We are now going to introduce systems and technologies that during this thesis we have studied in order to understand the underlying technology, the use cases for them and the potential advancements.

2.1.1 Introduction to Automotive Systems

A traditional operating system on a personal computer is a program that is first loaded when the computer boots and is responsible for managing all the programs,

communicating with the underlying hardware and the user. One of the main functions of the OS is to schedule the programs that are simultaneously executing, both on the background and on the foreground. This is done as it is needed to have multiple processes running in what may seem parallel fashion, but the processes are switched out from the CPU to allow a different process to run. An OS is also responsible for allocating all the memory needed to the processes, keeping track which processes are using which portion of memory, and not allowing processes to interfere with each other.

The purpose of an Automotive Operating system is to be able to control many different programs that are needed simultaneously in a car. Since a car has many different control units but also because it is a real-time environment where decisions are needed to be taken instantly to adapt to certain situations, these operating systems are designed to schedule and control various components of the car regarding how critical each of the tasks is.
Due to this critical rating standpoint, certain control units require operating systems that undergo a certification process, to ensure safety and responsiveness. Because of this, most operating systems used only control a subset of the car and are being developed to be able to get certified and potentially be used throughout it. As a consequence, there is a situation when a car's engine control unit (ECU) is using a different operating system than the infotainment and cluster system, simply because the infotainment system doesn't have any real-time or critical functions.

In this section, we analyze in detail the automotive systems studied in this project and what each of these has to offer, where they are used, what their capabilities are and what they can control in the car, and lastly how they allow developers to create their own applications and most importantly how they work.

2.1.2 Automotive Grade Linux

Automotive Grade Linux[8] is a project of the Linux Foundation. It is a collaborative open-source project that aims to bring manufacturers together to develop the connected car. The goal is that the platform will provide a starting point for the infotainment platform that will allow manufacturers to customize the system to their

needs and focus their resources on their product. In the long term, this project will implement more additional features such as the instrument cluster and telematics system. AGL is based on the Linux operating system core which has been modified to meet the tight requirements of the automotive systems and will provide an embedded distribution, which will allow the developers to produce for the platform through its framework.

The project's architecture consists of four main different components. The first component is the App/HMI layer which contains the applications that run on the system along with their logic. The second layer is the Application Framework which contains all the necessary APIs to create and run an application on the system. The third layer is the Services component that is responsible for all the application accessible services provided by the OS. These include connectivity services such as Bluetooth and WIFI, Audio services for applications that need to stream audio in the car and Speech services. Last but not least, the Operating System Layer which is built based on the Linux kernel provides the drivers for the hardware and all the other OS-specific features such as resource management.

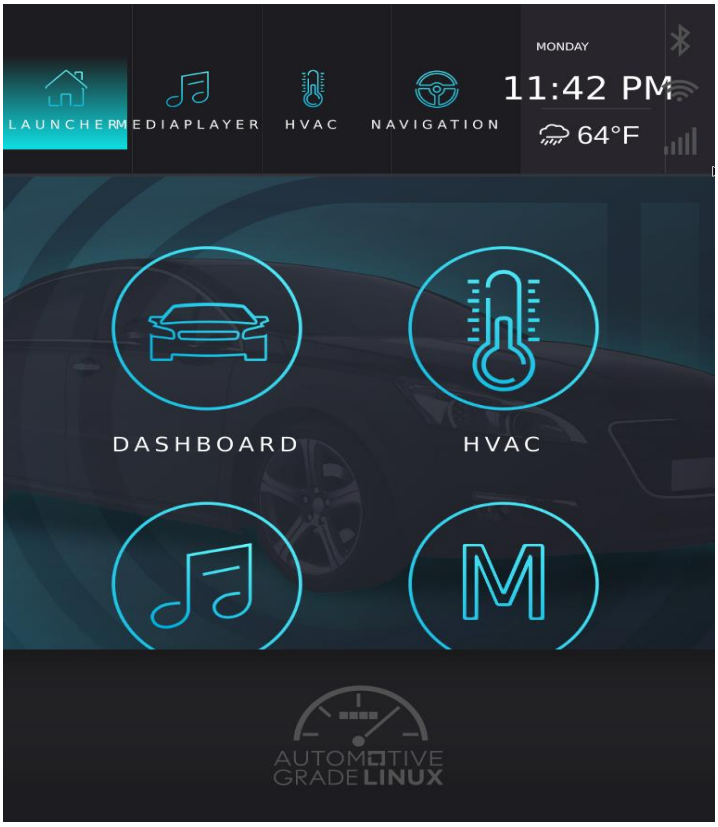


Figure 2-1 Automotive Linux emulated on x86

2.1.3 Android Automotive

One of the most successful smartphone operating systems, Android, now has a version that can be used with cars. Google[5] has created a version of their mobile operating system, to be used in modern infotainment systems in cars and it is able to run selected third-party applications of Android and pre-installed applications by the manufacturer. It is very important to note that this system is not separate from the Android OS running on mobile devices, but rather an extension of the system with automotive-specific features.

A big advantage of this system is that users are already familiar with Android. This allows users to work with their infotainment system seamlessly and they can expect to have a wide range of applications known to them available for in-car use. Well established applications such as Google Maps or a wide range of media players means the user now has the ability to select the preferred interface and features he/she wants to use while driving. In addition, users and manufacturers can exploit the Google's Assistant

abilities in the car, allowing them to stay more focused on the road, enhancing safety and overall providing a pleasant experience through the journey. With the use of this technology, users can send or hear text messages, that previously were unable due to the immense technological research needed to be put in by OEMs to provide such functionality.

On the side of the manufacturer, Android Automotive has significant advantages over the use of proprietary systems. Traditional infotainment systems require time-consuming development and cannot provide the level of usability that Android has to offer. These systems usually have a step-by-step navigation interface that doesn't allow the user to easily switch between functions of the IVI system. Android, on the other hand, being a mobile operating system, allows for multi-tasking, friendly user interface and has a great variety of features and applications that users can leverage during their journey.

Furthermore, cockpits are switching from the traditional knobs and dials to a digital environment which means that the production of the IVI system is even more complex now. Being able to cooperate with a software giant such as Google means that OEMs can now focus their resources on the development of the actual car, minimizing their cost and increasing user satisfiability. This though doesn't mean that the manufacturer won't have the ability to customize the IVI system to meet their specification and requirements. Moreover, OEMs can leverage the advanced development process that open-source Android has to offer and design applications that meet Google's requirements using the design guidelines given. These guidelines not only consolidate the safety of the system but allow for a great user experience since they have been formed by a company that focuses on devices and products that the most important review factor is the user experience through the UI.

Additionally, the use of Android Automotive will allow for easier implementation of the over-the-air updates. Until now software updating to cars had to be done in a professional environment such as a mechanic shop of the OEM's shop. This means that the car's software may have not been updated at all in many cases. With over-the-air updates, a manufacturer can deploy a new version for fixing a bug, improving the user's experience, and even adding new functionality to the whole car. Since Android

Automotive is an extension to Android, it already has all the needed functionality for delivering and executing the necessary updates, whether these are security or bug fixes for the operating system which are done through the operating system itself or whether these are application updates which are delivered through the Google Play Store. This is not just an advantage for the manufacturer but also for automotive software developers who wish to roll out new applications or updates to the platform. Lastly, like the over-the-air updates, since it is an extension of Android it allows for easier implementation of the future connected car because it already provides implemented and tested connection protocols and the Google services.

Cars nowadays feature complex networks to interconnect many different modules together, including the IVI system. This allows the systems to send and receive information. For example, if there is a problem with an electronically controlled actuator, this can show up on the car's dashboard but also a notification in the car's infotainment system, prompting the driver to stop in order to stop to give further information. Another example lies in many modern EV cars, which can show the battery status, charge level and mode of operation on the infotainment system. Because the actual implementation is on the OEMs' own intelligence to decide, it may differ between models of the same manufacturer or even different iterations of the model and of course, it differs between different brands. This creates an inherent problem for the use of a universal operating system since the developers cannot predict the chosen network and protocol implementation. Android Automotive features a hardware abstraction layer (HAL) which is an interface between the platform and the physical network. This allows manufacturers to create their implementation of Android Automotive. In order to achieve communication between the car and the platform, they can create a HAL module that is responsible to communicate between the HAL of Android Automotive and their own network to retrieve and send information. For example, a function that requires the exchange of information between the platform and the car is the HVAC system that may be connected to the car's network. The infotainment system will receive information about the status of the system, e.g., running, mode, and send commands that the user has issued meaning turning a knob on the screen that corresponds to increasing the desired temperature. Google gives examples of typical implementations that might be developed to deal with the HAL. One of them is using a microcontroller with its own operating

system that can access the network and the processor running Android Automotive via a link and act as the middleman.

In Figure 1, the outline of the system architecture is shown. The vehicle HAL provided sits at the bottom level and is the interface between the car and the system, containing the property metadata. The metadata contained depict the vehicle's properties which as explained in Google's documentation may be for example if a property is an integer and the allowed modes.

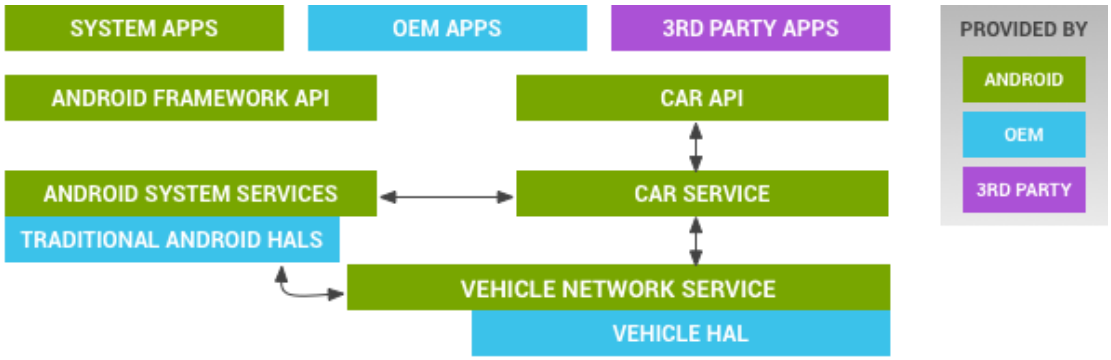


Figure 2-2 Android Automotive Architecture [6]

Security is a huge issue with vehicle data since if a malicious application retrieves access to them might be able to create vehicle malfunctions or share sensitive data which may be location information and potentially put the travelers in danger. To combat such behavior, the vehicle HAL provides security measures that divide the data into three categories. In the first category lie data that only the system can access. Since information might be needed for applications to function correctly, the second category provides information accessible through permissions like the traditional Android system we know on phones. Lastly, the last category contains accessible data without requiring permission from anyone. It is important to note that information is accessed through the Car Service.

Figure 2-3 Android Automotive on the included emulator of Android Studio

2.1.4 Android Auto

Android Auto[4] might not be an infotainment system, but still is a very useful feature that manufacturers added to their cars in order to improve the experience and enable the use of smartphone features in the cockpit. Google developed Android Auto as a feature that can be implemented in modern IVI systems in order to allow for mirroring of a mobile phone's screen to the car's infotainment screen. This allows for selected applications to function using the car's screen, enabling the driver a safer way to interact with the phone and control features like navigation, media applications, and message applications.

What Android Auto enables developers to do is the creation of a completely normal application for use on the phone and down the road add support for Android Auto in order to enable your app to be used with car dashboards that implement the feature.

All the processing and rendering of the application is done on the smartphone and then sent to the IVI system for projection. This is incredibly important especially with older car IVI systems which tend to not get updated often if ever. This enables the applications that can be used on Android Auto to be updated regularly like the rest of the smartphone applications. The system also supports routing audio to certain parts of the car, enabling separation amongst types of notifications. One example can be the routing of navigation instructions to the driver while the music is played through a media app to be played through all the speakers. While the audio focus is negotiated between the phone and the car, the latter has the authority of selecting the most appropriate audio focus – sound to be played since the car has real-time specific notifications it has to show or play in order to ensure driver safety.

2.1.5 CAN bus

CAN stands for Controller Area Network and is one of the main network architectures used in our vehicles today and is the basis of communication between ECUs. Its development started in 1983 by Robert Bosch GmbH and was introduced at the Society of Automotive Engineers (SAE) congress. During development, both Intel and Mercedes-Benz helped during the specification phase. Later came the CAN bus 2.0 and in 1993 the CAN bus was standardized in ISO 11898. In 2012 an important development was made with the introduction of CAN FD which brought flexible data rates and was too standardized in ISO 11898-2. CAN bus is based on a non-destructive arbitration algorithm and has several error detection mechanisms. The first controllers for the network were made by Intel and Philips each one using a different specification. The former used the FullCAN specification that minimized CPU load and the former used the BasicCAN specification. While back then, this separation existed, today a mixture of both is implemented. When the Can in Automation (CiA) group was founded which is a group between users and manufacturers for CAN, a recommendation for using transceivers that comply with ISO 11989 was made. This group also standardized the CAN application layer (CAL).

The introduction of ISO 11989 standardized the physical layer of the network to support bitrates of up to 1Mbit/s. Later the standard was extended in order to support a

29-bit identifier for the messages whereas the non-extended standard supported only an 11-bit identifier.

This vehicle network defines the data link and physical layers of the Open Systems Interconnection (OSI) model. The microcontrollers and devices connected to the network communicate using their application without a host computer. It is a message-based communication network meaning that the invoking program sends a message to a process which in turn will select and run the relevant code. It supports communication within threads of a process, between different processes on the same host, and between different hosts. Frames sent by a host are broadcasted, meaning that every device on the network will receive them. The receiving nodes on the network will decide if the message is important to them otherwise, they will filter it out. This allows for easy modifications to the network and non-transmitting nodes can be added without any modification at all. Using the data arbitration mechanism, only the highest priority device transmits a message, and the frame is transmitted sequentially.

The network is comprised of two wires. In order to function, it uses differential wired-AND signals. CAN high (CANH) and CAN low (CANL) can be driven to a dominant state where CANH > CANL or they can be in a recessive state where CANH<=CANL. The 0 bit encodes the dominant state and the 1 bit encodes the recessive state, meaning that nodes with lower ID values will be prioritised. Nodes in the network must have a CPU that will process the logic of the node, a CAN controller, and a transceiver. The role of the controller is to convert the application data to a CAN message frame. This results in a reduced CPU load and may sometimes be integrated into the CPU itself, reducing the I/O and further increasing performance. The transceiver end is defined by the ISO 11989 – 2/3 and its job is to drive data to the bus and sense the data coming from it. In order to do that, it has to convert the single-ended logic of the controller to a differential logic. The single-ended logic carries the electrical signal through a wire with varying voltage and has a separate wire which is ground used for reference. On the other hand, the CAN bus uses differential logic where there is no ground and the measurement to detect a 0 or 1 is taken between the difference of the two wires. In addition to the main functionality of the transceiver, it may also perform noise filtering.

Figure 2-4 CAN Frame Structure [11]

As we can see from the figure above, a CAN frame begins with the Start of frame bit and is always at dominant level. Then the arbitration field begins which consists of the identifier bits, protocol bits showing the length of the CAN-ID, and reserved bits for future use. Then comes the control field which provides control bits and indication about which of the two protocols is used, either Classical CAN or CAN FD from which we can understand the length of the data field. The data field contains the payload, either 8 bytes for Classical CAN or 64 bytes for CAN FD. The CRC field contains the cyclic redundancy checksum and in the case of CAN FD, it contains a stuff-bit counter. Following the CRC field is the ACK field that contains bits to show that the message was received correctly. The last 2 fields are the EOF and IMF, the former denoting the end of frame with 7 bits and the latter separating the frame from the next one.

Because the network uses a multi-master protocol resulting in any device having access to the bus at any time a bus arbitration system is in place. This is done to ensure a functional network even when multiple devices want to use the bus at the same time. The message that is of the highest priority gets access to transmit through the network. The identifier bits of the frame indicate the priority and are assigned by the network designer. Because the dominant level is indicated by the 0 bit, the identifier 0 is always is the highest priority. Arbitration is done while the identifiers are transmitted and is a bit-wise comparison. Nodes that transmit at the same time check the levels on the bus and if a node is in a recessive state, but the line is on a dominant state, the node loses the arbitration and automatically goes into listening mode.

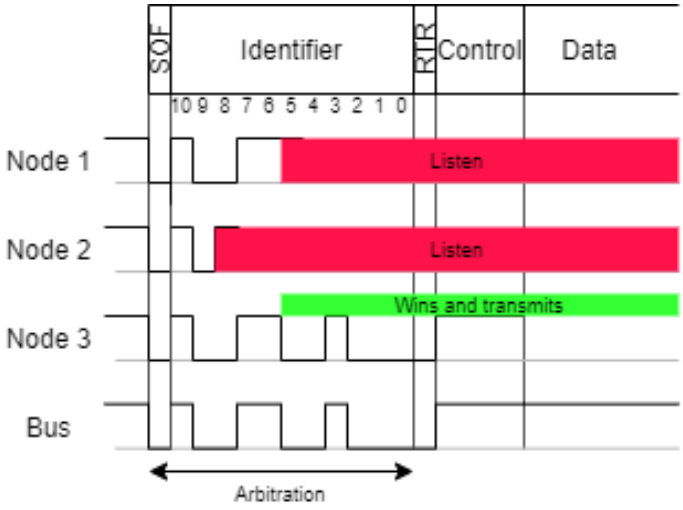


Figure 2-5 Bus Arbitration

2.1.6 OBD2

OBD2 stands for On-board Diagnostics for vehicles and it originates from California where in 1991 it was required in order to perform emissions control. Later in 2001, it was required in all gasoline cars in the EU and in 2003 in all diesel-powered cars. Through the OBD port of a car, one can get lots of information from the CAN bus and even send commands to issue changes in operation. OBD features a 16-pin connector and has five different signaling protocols. The most relevant is defined in ISO 15765 which is used for the CAN bus since is the standard for modern cars. The communication protocol used by OBD is also standardized in ISO 15031.

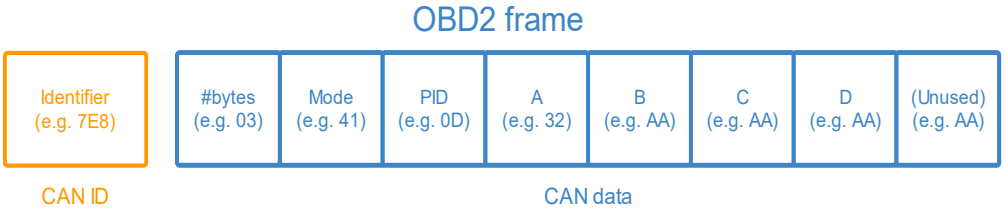


Figure 2-6 OBD2 Frame Structure [14]

OBD has different modes of operation. For example, a mode is used to display freeze frame data while a different mode retrieves vehicle information which can be the VIN number of the car. Since this protocol is standardized, many standard PIDs exist that

can be used to retrieve different information about the car. In the frame structure, the first part is the CAN ID, then the length of the following data. Next comes the mode which indicates whether the message is a request or response. Following the mode is the PID and lastly the data bytes.

OBD has many applications and is available to users since many devices exist, each with their own functionalities. OBD2 devices can be CAN loggers, which simply listen and log every message sent through the network for further examination. A different device is an OBD2 scanner that is used to reading and clearing Diagnostic Trouble Codes (DTCs) that indicate a fault on a vehicle's system. Lastly, OBD2 interfaces are devices that can be used during development and real-time advanced diagnostics.

2.2 Automotive Software certification

We have established that modern cars use complex software systems that provide useful functionality to the user. These systems are incredibly complex and the correctness of them is a critical point in order for the manufacturer to ensure the safety of the vehicle. If a software system experiences a sudden failure, this might have a catastrophic impact on the vehicle and most importantly it may even cost human lives.

Certification is one of the most important aspects for the safety of products. It provides a way to assess a product using well-defined specifications in order to ensure that it complies with laws and perhaps most importantly ensures the safety of the user by minimising risk.

Automotive certification ensures that a car complies with laws designed to ensure the safety on roads for everyone that is using them. These certifications contain standards and specifications that have to be tested against the corresponding car components and the car as a whole system. This means that different parts of a car have to be tested to separate certificates in order for the vehicle to pass the certification.

Unfortunately, in the automotive world, software certification has not yet been developed to a point where there is a concrete method of certifying the software running in a car in contrast to other types of certifications that are used for vehicles [1] [3]. On the other hand, safety-critical domains such as aviation and medical have standards such as the DO-178B for the former and IEC 62304 for the latter. A step in the right direction has been made in the automotive industry with the development of the ISO 26262, but the need for a certification method is still live.

2.2.1 ISO 26262 – "Road vehicles – Functional safety"

ISO 26262[12] is a standard for functional safety for electronic/electrical components used in road vehicles covering both hardware and software aspects. It is based on the standard IEC 61508. The standard covers the whole lifecycle of these systems starting at the conceptual phase all the way to decommissioning. The purpose is to reduce the possibility of a failure resulting in a hazardous environment. In order to achieve this goal, OEMs must provide the necessary evidence supporting that the safety goals have been achieved [2]. This is done through the preparation of safety cases that prove the standard has been followed and the risk of the component is lowered. On the software side, a software assurance case has to be prepared to support the safety case of the component.

This standard specifies an automotive-specific risk measure named Automotive Safety Integrity Level (ASIL). ASIL scale has 4 different levels ranging from A to D and is determined by combining three different factors. The first factor is the severity of the damage when a system failure happens and ranges from S0 (No Injuries occurring) to S3 (Life-Threatening). The second factor is Exposure which is the probability of a failure happening and causing hazards ranging from E0 (Incredibly low) to E4 (High Probability). Lastly, the Controllability factor determines whether the driver can control the car in the event of a failure ranging from C0 (Controllable in general) to C3 (Difficult to control / Uncontrollable).

Severity	Exposure	Controllability		
		C1 (Simple)	C2 (Normal)	C3 (Difficult, Uncontrollable)
S1 LIGHT AND MODERATE INJURIES	E1 (Very low)	QM	QM	QM
	E2 (Low)	QM	QM	QM
	E3 (Medium)	QM	QM	A
	E4 (High)	QM	A	B
S2 SEVERE AND LIFE THREATENING INJURIES – SURVIVAL PROBABLE	E1 (Very low)	QM	QM	QM
	E2 (Low)	QM	QM	A
	E3 (Medium)	QM	A	B
	E4 (High)	A	B	C
S3 LIFE THREATENING INJURIES, FATAL INJURIES	E1 (Very low)	QM	QM	A
	E2 (Low)	QM	A	B
	E3 (Medium)	A	B	C
	E4 (High)	B	C	D

Figure 2-7 ASIL levels of ISO 26262 [7]

ASIL is a critical part of the standard as it is used to determine which of the protocols defined in the standard must be followed to avoid risk. The different ASIL levels require different procedures to prove that the software has been tested properly and can respond sufficiently in a hazardous event. The standard uses the V development process with critical processes fixed and provides recommendations for the technologies that can be used in order to reach an acceptable risk level. At the software level, the requirements are functional requirements that have been extracted through a system hazard analysis and dependability requirements.

ISO 26262 provides the automotive lifecycle, the functional safety aspects of the development process, ASILs and leverages them to define the safety requirements to achieve an acceptable risk level and the requirements for validation to ensure the safety goal has been met.

The first step for the software side is to specify the software safety requirements including hardware to software interface. The second stage then takes place which is defining the software architecture. Here the standard provides recommendations on the notation to be used based on the assigned ASIL and defines critical properties of the design that have to be met to ensure safety. When the software design is finished, the

implementation phase begins where the standard applies code specific requirements such as verifiability. Here the design created in the second stage has to be strictly followed as well as code specific properties such as the prohibition of recursions since an embedded system stack might get stuck. Additionally, software verification takes place in this stage with many techniques such as data flow analysis. In the next stage, unit testing begins to ensure no unintended behaviour and ensure code correctness. Then comes integration testing and for the last step, the software has to be verified against the defined safety requirements to guarantee coverage.

A really important aspect of the standard that has to be noted is that any test tool that will be used in the development of the component must be qualified since it is of utmost importance that any failures will be detected at this stage rather than in a fully developed and deployed product.

2.2.2 MISRA – Motor Industry Software Reliability Association

MISRA[13] is an association that started between vehicle manufactures and has grown to now include members from other industries that operate in the sector of embedded systems with a safety aspect. The goal of this collaboration is to create coding guidelines for safety-critical embedded systems. MISRA has coding guidelines for both C and C++ languages that are updated to this day. The authors of these guides have collected a huge amount of the flaws and mistakes that can be made when using these languages but have also taken into consideration their respected programming language standards that are available and how those come into play.

Rules of these guidelines include common mistakes that should be avoided in any project, not only embedded systems, and are divided into three categories. First, the mandatory rules which cannot be violated, secondly the required rules which can be rejected only in the case of adequate documentation explaining the reasons and how they will not create a risk and lastly the advisory rules which are not forced but good practice. In order to get a general idea, we outline some rules such as not using a pointer to FILE after the file stream is closed, always returning a value from a non-void function, and never using the value of an uninitialized variable.

But MISRA is not just about good coding behavior that every coder should adapt to make the code readable and reliable. It contains rules that are more specific for embedded critical systems that must be forced to avoid unpredictable behavior. Such rules might sound weird to programmers that do not have experience with these types of systems because rules like not using heap memory exist. But these make perfect sense in safety-critical environments since using dynamically allocated memory might lead to memory leaks, non-deterministic behavior, and data inconsistency.

Application of MISRA guidelines should start early in the development process since if the project already has thousands of lines of code, it would be extremely hard to change the current code to be MISRA compliant. A programmer doesn't have to constantly check the code to make sure that it is compliant since many static code analyzers exist that implement the rules of MISRA and can pinpoint places in the code that have violations and can even produce a review of the code.

MISRA offers many advantages and is not just a burden for the programmers. The standards not only help minimize the possibilities of bugs in the code but also improve the re-usability, reliability and take into consideration the latest security exploits of the languages supported to help shield against them.

2.2.3 AUTOSAR – AUTomotive Open System Architecture

AUTOSAR[9] is a partnership between manufacturers in the automotive space, suppliers, and software industry on a global scale. The main goal of this collaboration is to create an open standardized software architecture for ECUs in the automotive space. As we discussed before, modern vehicles contain lots of different ECUs and all of these communicate in order to provide the functionality for the car. Developing the software separately for each ECU is a huge task and so AUTOSAR was born in order to standardize the architecture and functions among the members of the partnership to help reduce the re-development, increase security and improve the performance. Manufacturers can choose any microcontroller and use it in their car since with the AUTOSAR architecture the software is independent thanks to the hardware abstraction layer. The platform is compliant with many automotive standards and supports all the standard functionalities

of an automotive ECU such as real-time processing, connection through automotive networks and working with microcontrollers of 16 or 32 bits.

The classic architecture is divided into 3 main sections. Everything runs on a microcontroller, on top of that lies the Basic Software (BSW), then comes the Runtime Environment (RTE) and on top of all is the Application Layer. The Basic Software layer itself can be divided into 4 main layers, each one being comprised of smaller categories. On the bottom of the Basic Software, there is a Microcontroller Abstraction Layer which is responsible for containing the main drivers and communicating with the microcontroller. Then the ECU Abstraction layer is laid on top of it to provide APIs for access to devices and keep the higher layers of software independent of hardware. Another helpful layer is added which is called Complex Drivers that and provides extra functionality and drivers of devices to the Runtime Environment. Lastly, the Services Layer contains of all the operating system functionality, network communication and services to the ECU. On top of the Basic Software, the Runtime Environment is responsible for providing services and communication to the applications running.

The whole AUTOSAR architecture can be divided into many smaller categories, with libraries and deeper functionality but this is beyond the scope of this thesis. This standard provides a robust and secure architecture to manufacturers that helps strengthen the security of their software running on their ECUs, minimise the development time since the architecture can be reused in many different microcontrollers allowing for changing the application software that is running depending on the functionality needed.

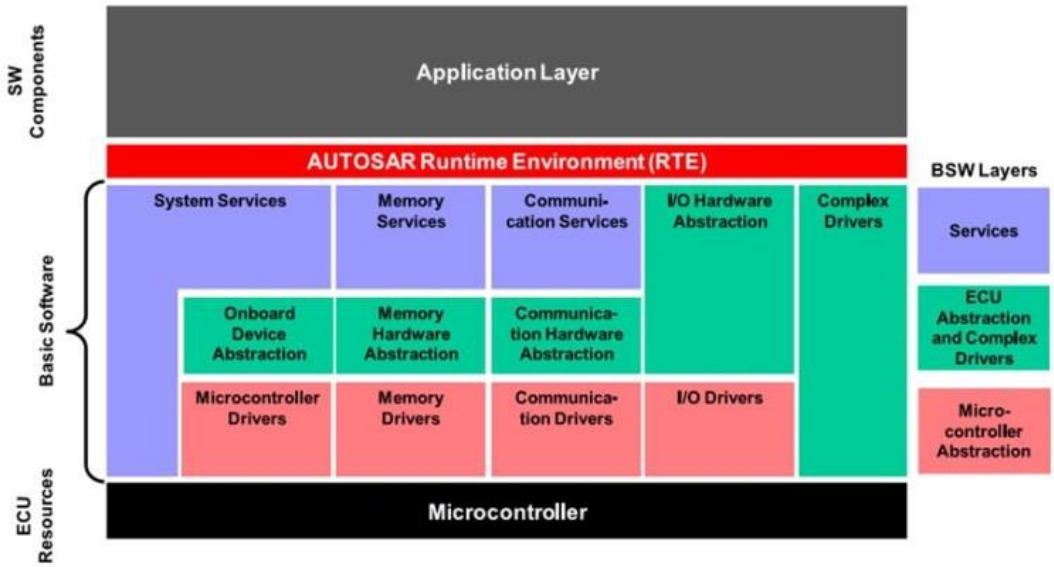


Figure 2-8 AUTOSAR architecture [10]

Chapter 3

Android Auto

3.1 Introduction

Android Auto was first introduced on June 15, 2014, at Google I/O. Later on March 19, 2015, it was released to the public. Android Auto is an application for Android smartphones that allows users to mirror their smartphone's screen to the car's infotainment system. Users connect their Android phones to their car through a USB cable and follow a pairing procedure. When this is done, the vehicle's infotainment system will display the compatible applications on the phone. Everything displayed is basically sent from the phone to the infotainment system and all the required processing is done on the phone. Android Auto interaction is achieved through the vehicle's display and also supports interaction with steering wheel buttons if the manufacturer implements the functionality. Recently Google added support for a wireless connection to the car. To connect a phone via USB cable, Android 6.0 is at least required with the Android Auto app installed. To use the wireless connection functionality, the latest version of the application is required along with an Android 11 smartphone that supports 5Ghz Wi-fi or selected smartphones with Android 10 or 9. Android Auto can also function as a standalone application without connecting to a car. When using the application this way, the smartphone's screen is used directly and can be mounted for better viewing.

Displayed applications and items with Android Auto use a very simple and intuitive user interface that is non-distractive. This minimizes the amount of time needed to scroll and search through items on the screen in order to accomplish your goal. It is very useful because it allows users to interact with their phone while in the car without putting their selves, passengers, or other users of the road in danger.

Notifications from the phone are routed to the car and are displayed in a non-distractive way. In addition, messages and notifications can take advantage of voice assistance and further ensure safety by minimizing the interaction with a screen and keeping the driver focused on the road. Calls are also supported and can be made using either voice commands or dialing the number on the screen. Furthermore, voice commands using the Google Assistant can be used to take actions other than just calls while driving and further minimizing screen interactions. These include setting reminders, checking agendas, setting the music, or getting the news. The built-in navigation with Google Maps is especially useful since the user can interact with other applications and still get directions for the route either by voice or using the navigation bar.

3.2 Architecture

As we explained before, Android Auto projects your phone's screen on the vehicle's screen and uses your phone in order to execute all the necessary code. All the processing is done on the phone and the car's screen is just used to show the UI.

Android Auto running on the vehicle's infotainment system contains the vehicle abstraction layer. This layer is independent of the underlying hardware of the car for audio, display, inputs, and sensor. It is up to the manufacturer to implement these parts and create the interface between them and the abstraction layer. When the phone connects to the car, it will create a channel for each part of the car that it needs to send and receive data. There is one channel for each part in the abstraction layer. On the phone side, the android auto service will run which is responsible for the applications and the communication.

To render applications it uses XML to define windows and to render them across different apps. As explained by Google, there are multiple windows that are actual TextureViews whose job is to display content from streams. These windows include the navigation window, main content area for the search, and notification areas that will be rendered to a common surface for use on the car screen. In order to render these windows, the Android Auto service will take the TextureViews, convert them to SurfaceTexture objects and wrap them in surface objects. The surface objects will then be passed to the client service where a private virtual display is created that will contain the application's UI. The XML layout will then be rendered using the hardware video encoder from the phone and the image will be sent to the head unit. According to Google, there are two advantages to using this method. The first is that the standard Android Layout system is being utilized. Secondly, the security is enhanced since applications are rendered in their own process which improves isolation in the system.

3.3 Developing for Android Auto

As we explained previously, Android Auto only allows developers to create media and messaging applications. Only recently, support has been added for the development of navigation applications.

During this thesis, we tested both media and messaging applications to understand the process of development for each application to add support for Android Auto, find their differences and what features they enable within Android Auto.

3.3.1 Messaging Applications

Messaging applications can leverage the power of Android Auto enabling their users to continue receiving notifications, read and reply to messages while in their vehicle in a safe way.

When receiving a message, the application needs to create a notification using the class NotificationCompat.Messaging Style. In addition, the notification created must have

a reply and mark-as read action. When the notification is generated Android Auto will consume it and will find the included actions. Based on what it received it will create a notification just for Android Auto and it will be projected to the vehicle's display. Now the user will have 2 options, mark as read or reply to the message, and both will be shown on the notification. If the user interacts with the car's display and just touches the message, the mark-as read action is triggered, and the application has to process this in the background. Otherwise, if the user selects to reply to the message, which is done using voice transcription, Android Auto will include it in the action and the application again must process it.

To add support to a messaging application for Android Auto first support has to be declared in the manifest using the meta-data tag referencing an XML file that describes the capabilities that your application has within Android Auto. The next important point in implementation is handling the actions. The recommended way of handling actions is through an IntentService and with specified strings that describe the action for the Intent. When the service receives an intent, it will check for the action linked to it and trigger the appropriate action. To receive the Intent for the messaging service a RemoteInput object is needed that will allow remote applications to provide the response with the intent for the reply action. The intents for reply actions and the action itself have specific requirements that must be followed and are well documented in Google's guide. In order to enable Android Auto voice read of the message, a MessageStyle must be created that will hold the information for the conversation. Lastly, the notification must be constructed using the NotificationCompat.Builder.

We created a simple messaging application for use with Android Auto as an example. There are plenty of open-source implementations online and some are more complex including all the necessary classes and services to be complete with everything described before. In our case, because our objective was not to create a messaging application, the sample application only creates the notifications and has the necessary support for Android Auto to display them. Our sample application consists of a very basic UI that enables the user to write a message and send it in order to create a notification.

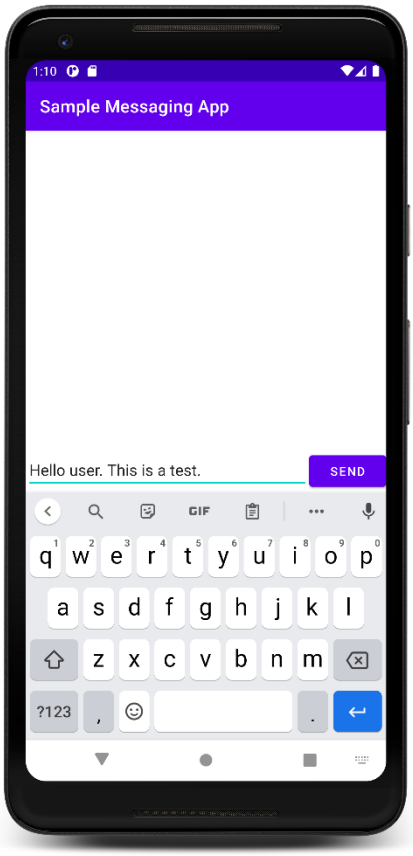


Figure 3-1 UI for Sample Messaging application

Figure 3.1 shows the UI for the sample messaging application we created. It allows the user to write a message in a text box and when pressing the send button, the application will build and send a notification to the user. If the phone is not connected with Android Auto, the notification will show up like any other regular message notification, with the only difference being that it does not have available actions associated. When the application first starts, it will create a notification channel to allow sending the notifications that will be built. Pressing the send button will trigger an action that will first read the message and call a class that acts as a notification manager we created, passing the message, a fake conversation subject, contact name, and conversation id.

In our notification manager class, we begin building a simple notification. First, we create a NotificationCompat.Builder object with the notification channel. Then we start adding the content title and message and create our two intents. Each intent is associated with one action. From the two intents, we will create PendingIntents. In our

case, because we only wanted to create a sample application to demonstrate how Android Auto works with messaging applications and how the notifications will be displayed, we created a fake Person object which we used to create our MessagingStyle. Finally using the pending intents, we create the two actions which were appended on our NotificationCompat.Builder object along with the MessagingStyle. This way whenever we type a message on the text box, it is transferred to this manager class where this procedure will take place for every message we type and the resulting notification object from the builder is used with the method notify of a NotificationManagerCompat object. Our class follows the basic structure of the guide for creating a messaging application and only omits the creation of actual conversations and their handling of the actions.

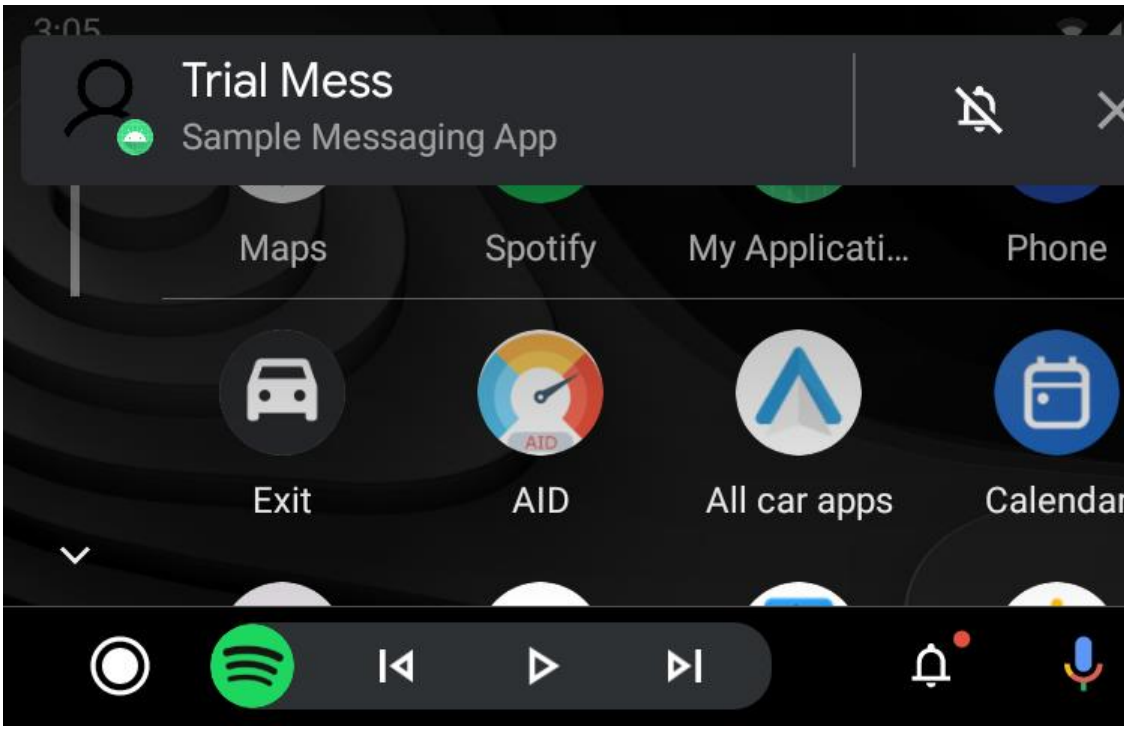


Figure 3-2 Receiving message notification on Android Auto

When not connected on Android Auto, using the application to send a message will result in a normal notification for a message and will be shown on the phone's screen along with buttons for the reply and marking the message as read. Figure 3.2 shows the result of using the application to send a simple notification when connected with Android Auto. The screen of the infotainment system displays the notification with the icon of the user that sent the message. The bright letters show the title of the conversation that is set in the MessagingStyle and the greyed-out letters below show the name of the application

used. Pressing on the notification will open the Google Assistant which will begin reading the message as well as information about what application is used and who is the sender and asking if you want to reply to the message.

3.3.2 Media applications

Media applications with support for Android Auto allow users to use their favorite applications for music while in their car. Any media application can get support for Android Auto using by implementing simple code.

In order for Android Auto to communicate with a media application, a MediaBrowserService must be present that is declared in the application's manifest. Unlike messaging applications, media applications must also declare an icon in manifest that's used by Android Auto. This is because media applications can be shown on the vehicle's screen, where messaging applications just project their notifications on the infotainment system. When launching a media application from Android Auto, the AA service will call the onCreate method of the MediaBrowserService in order to connect with, and then it will call the onGetRoot method to get the root of all the media items that are present in your media application. This then allows the service to call the onLoadChildren method which will give all the children of the media root. Whenever a user wants a browsable item from your application, onLoadChildren is called to fetch the children contained in that item. The implementation of the onGetRoot method must return a MediaBrowserServiceCompat.BrowserRoot object which will be used when calling onLoadChildren to identify the level of the tree and use that as the root to load its children.

Every node of the tree, content hierarchy, is a MediaBrowserCompat.MediaItem object. Because of this, the onLoadChildren method must return a list whose items are of this type. One additional feature is that there is support for selecting through parameters for the style of the content. These parameters are added in Bundle and returned with the root when the onGetRoot method is called. Furthermore, item specific styles can be set, a grouping of related items and searching on the items. Lastly, in order to be able to play the media items and support other actions like pausing, a MediaSessionCompat object must be created in the service.

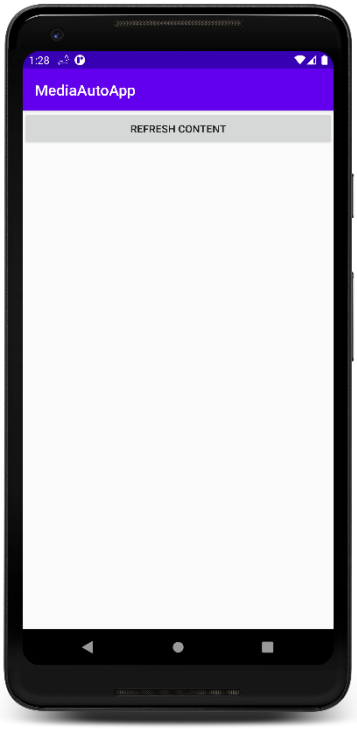


Figure 3-3 Sample Media application for Android Auto

In order to better understand the functionality, features and abilities of media applications for Android Auto, we created a sample application that projects fake media items on the car's screen. In our case, rather than showing actual music, we created media items that displayed information for the car. In other words, rather than displaying a song, our fake media item contained a property and value of the car, which could be for example the oil temperature. Figure 3.3 shows the UI of the application on the phone. It is very simple because our goal is not to create a good media application for the phone but to project items on Android Auto using a media application. Once the application starts, it will create five artificial car properties along with their values and they will be stored in a table of a database. When the "Refresh Content" button is clicked, it will randomly select a property, update its value in the database, and notify the MediaBrowserService using an Intent. This service starts automatically with the application and contains the onGetRoot implementation and the onLoadChildren. The root returned from the onGetRoot is simple the top node in our content hierarchy. When onLoadChildren is called, an instance of a class called MusicLibrary is created. This class is responsible for retrieving the properties from the database, creating and returning the media items from

those properties to the onLoadChildren method. To create the media items, the class MusicLibrary contains a method that connects to the database, retrieves all the properties and their values. Then it loops through every property and creates a MediaMetaDataCompat.Builder object that as title contains the name of the property and as an artist the value. It also adds a picture depending on what the property is. Then this object is built using the corresponding method resulting in a MediaMetaDataCompat object and stored in a TreeMap that corresponds to the content hierarchy. When we want to retrieve the items, we create and return a list from these objects transforming them to MediaBrowserCompat.MediaItem objects which is what is required by the MediaBrowserService.

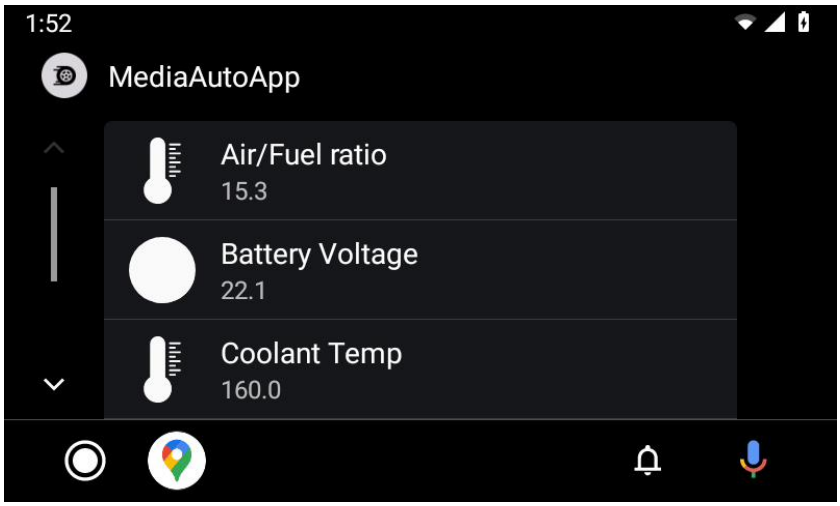


Figure 3-4 Sample media application on Android Auto

In Figure 3.4 we can see the sample application when used from Android Auto. It directly opens to this list were the five fake items are displayed. If the “Refresh Content” button is pressed only the value refreshes on the screen and the user does not notice a change.

3.4 Market Applications

Android Auto integrates seamlessly with the Google Play store. Applications that want to support the feature, do not have to create two different APKs. The single APK is delivered and if the user wants to use the feature, it is available. In the Play Store, a user

can search for specific applications that have the feature implemented in order to use them in a supported vehicle.

The most accomplished and famous messaging and media applications have support for Android Auto. Currently, there are about 238 available applications with support for this feature. In the category of media applications, there are multiple music streaming applications, radio stations, audiobook applications podcast applications. In the category for messaging applications, multiple instant messaging applications exist with support for it.

Chapter 4

Auto Insights Dashboard

4.1 Motivation

Looking at the market for available applications for Android Auto, we can easily see that no applications are available that can help the user connect to his car in a more informative way. All applications available stick to the main allowed structure of the applications supported by the service which, as we explained is either media or messaging applications. None of the available applications offer anything more or different from the main functionality. The users of the service do not have the option to opt for an application that can help them understand their car behaviour and keep an eye on important aspects of it. One of the most efficient ways a user can get more information about his car and gain a deeper understanding is through the OBD2 port. While plenty of OBD2 reading applications exist for Android smartphones, none of them support the Android Auto service. On the one hand, this is understandable since it is not officially supported by the service, but on the other hand, we have seen applications innovate in creative ways to get around limitations and offer their users a complete experience.

4.2 Description

Our application is an extension to an open-source application that connects to an OBD2 device and projects the data to the user. The job of our extension is to encapsulate all the data and project it to the user using one of the technologies we have introduced before, Android Auto, using the car's infotainment system.

The goal of our application is to allow the user to view his data on his car's infotainment screen. We want to provide a seamless experience both during connection and during the use of the application. The user must be able to view his car's properties, select the period between updates on the infotainment screen, and be able to select his preferred data items and show them in a different list. Furthermore, it is important that the data shown on the vehicle's screen are not presented in a distractive way to ensure passenger safety and avoiding driver distraction.

4.3 Main Screen

When we launch our application, we will be presented with the main screen shown in Figure 4.1 and Figure 4.2.

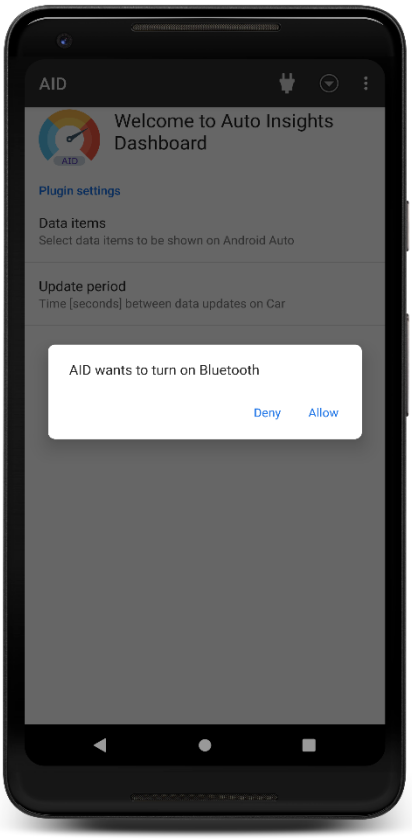


Figure 4-1 Launching the application

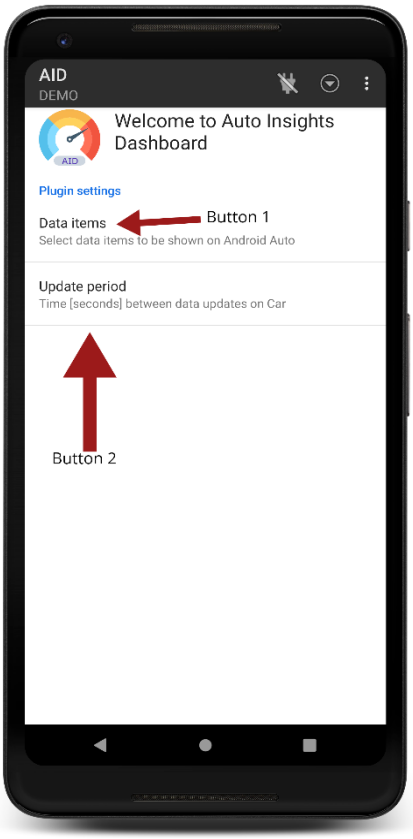


Figure 4-2 Main Screen

As we can see from Figure 5.1, we are presented with an option to turn on Bluetooth. This is the default option for connecting to an OBD2 device, but the application also supports connecting through Wi-Fi or USB. After proceeding with the selection, we are presented to the main screen as shown in Figure 5.2. We now have two main options that can affect the Android Auto presentation.

The first option is edited using Button 1 and is called Data Items. Using this option, we will get a drop-down menu as seen in Figure 4.3 that will allow us to select multiple data items that are properties of our car. Selecting those will result in a new list in Android Auto that will only show our selected items for a cleaner look since the main list will contain over 100 distinct data items.

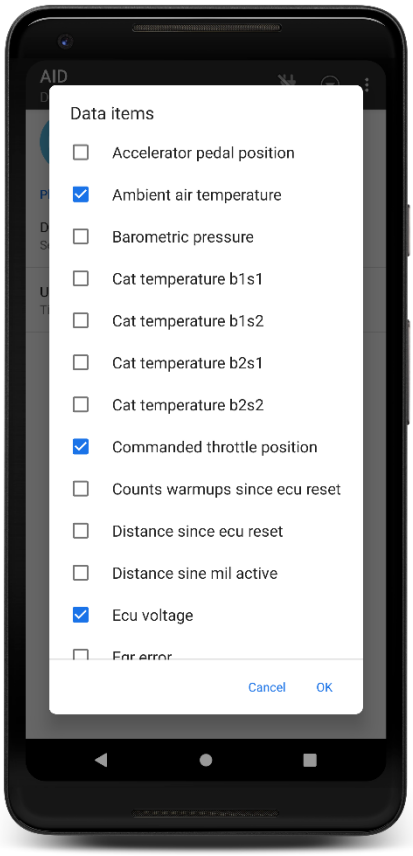


Figure 4-3 Data Items Selection

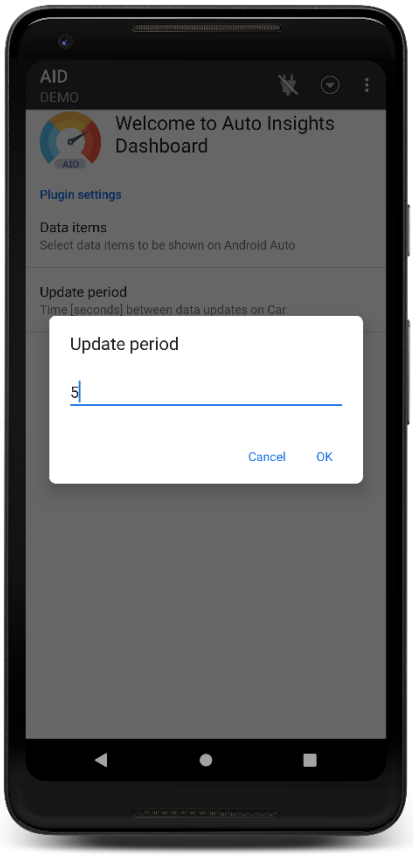


Figure 4-4 Update Period setting

The second option is changed using Button 2 and is named Update Period. This defines the interval between updates on the values of data items presented on the infotainment system. It is an integer number in seconds and selecting the option will bring up the box for entering the desired value as shown in Figure 4.4.

We can also see that on the top section of the application we have more buttons. These reveal application-specific settings which we are not going to discuss since our focus in this thesis is the car side of the application. A brief overview of these settings is that the rightmost button will reveal settings for the OBD device which include the connection method and also common settings such as units of measure of night mode. The arrow button will reveal a drop-down listing different screens of the application such as viewing the data on the phone screen or the vehicle information. Lastly next to the arrow button we have a button that connects or disconnects to the OBD device.

4.4 Android Auto presentation

We are now going to analyse the Android Auto side of the application. Once connected to the OBD device on the smartphone, we have to launch the application set out parameters, and go to the data screen to start receiving data from the OBD device. After completing these steps, we can connect our phone to the car and launch our application on the car. This will result in the screen shown in Figure 4.5 where we have 2 options that are browsable.

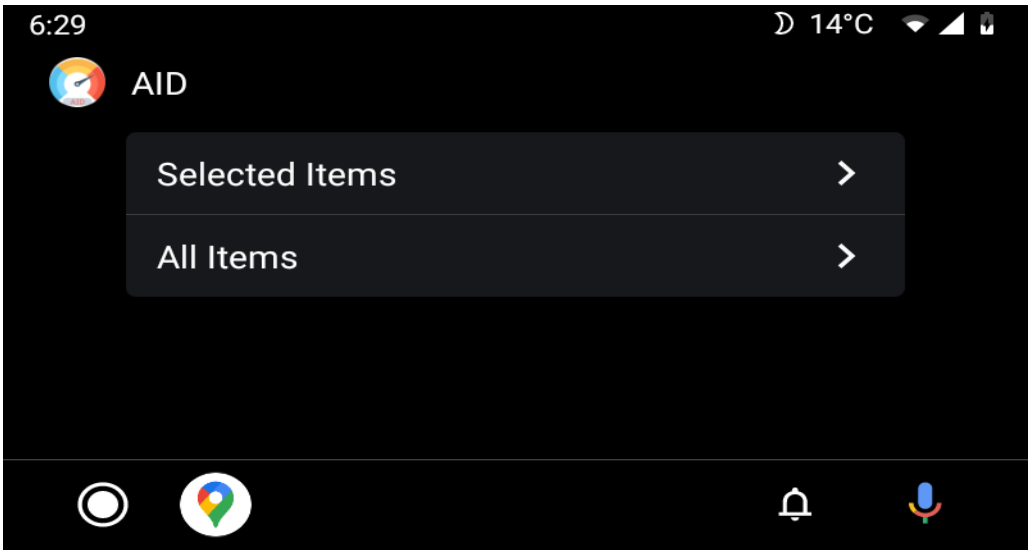


Figure 4-5 Main Screen on infotainment system

We can now select either of the 2 categories and we will be transferred to the related screen containing the appropriate data items. The “Selected Items” option will get us to the screen showing only our desired car properties for easy monitoring. Figure 4.6 shows this screen.

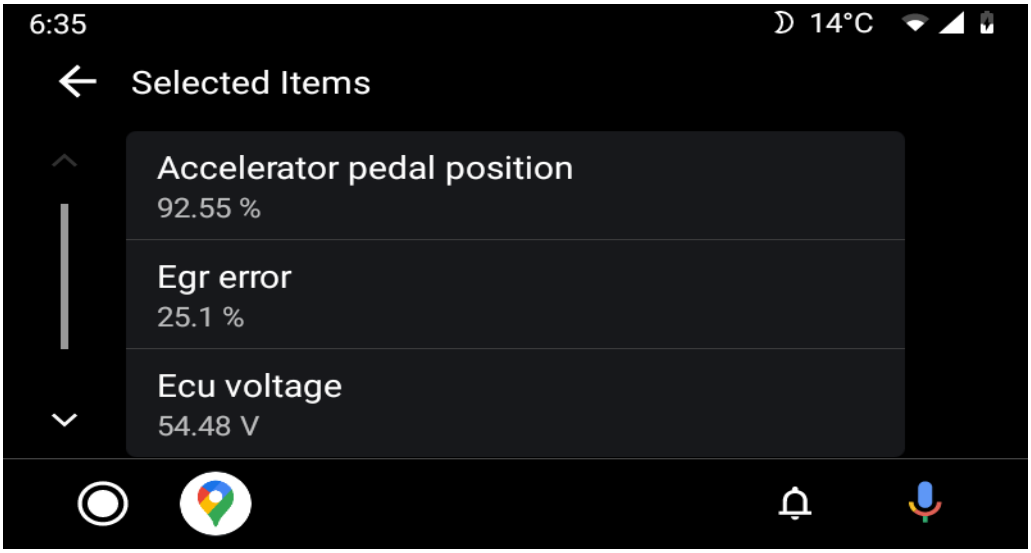


Figure 4-6 Selected Items screen for Android Auto

Lastly, the “All items” screen shows a list that contains all the properties that are read by the OBD device connected. Figure 4.7 shows a screenshot of that page.

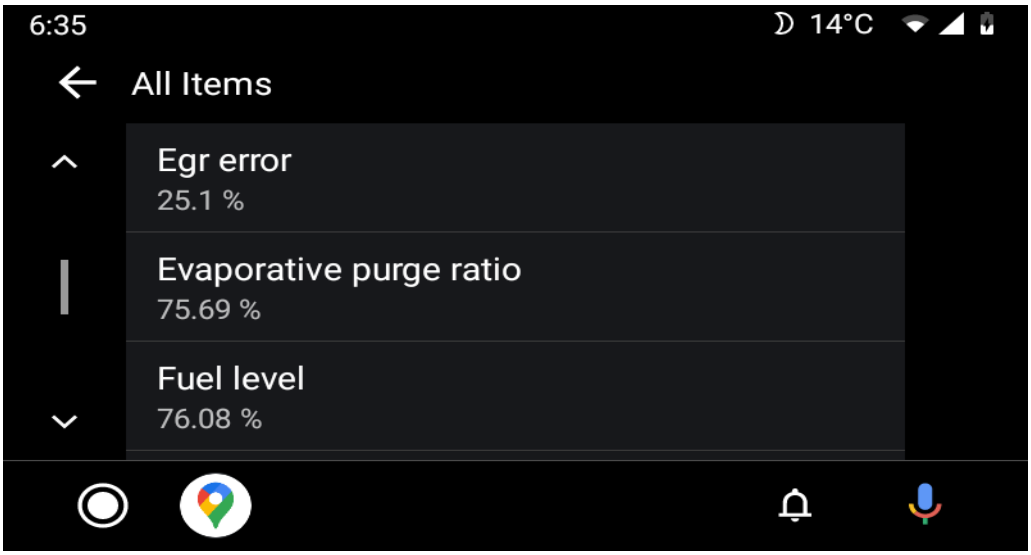


Figure 4-7 All Items Screen on Android Auto

4.5 AID Website

To promote our application to potential users we created a website that shows all the features of AID. Our website contains a description of the application and a video showing how to use our application and its features. In addition, we allow our users to directly download the APK file to install our application directly from the website. A link to our Github repository is also available on the website allowing users to view the code and evaluate the application themselves.

This website plays an integral part in the distribution of AID since it contains out Terms of Service, the Privacy Policy of the application and the Rules for using it. Any user who wishes to use this application must read these documents before proceeding to download, install and use our application. Users who would like further information about the application or have recommendations about it can contact us through the contact form included on the website.

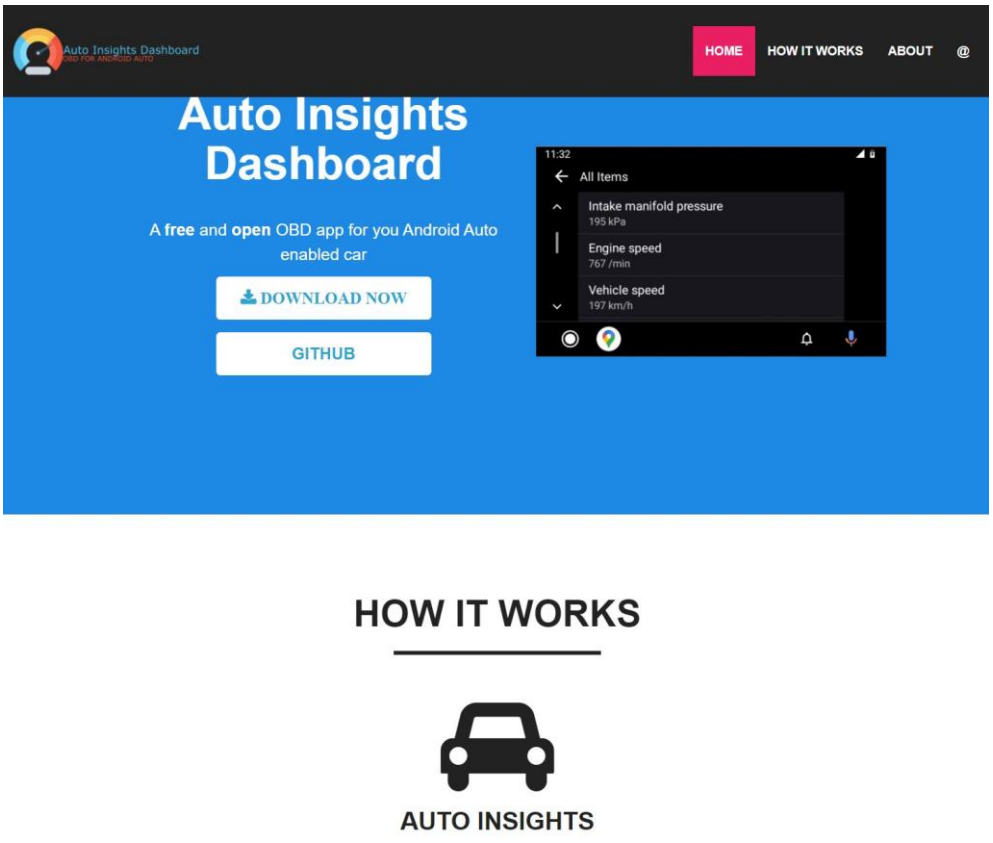


Figure 4-8 The website for AID application

Chapter 5

Architecture of Auto Insights Dashboard

5.1 AID Architecture

In this chapter, we are going to analyse the architecture of our application. The goal with AID was to learn more about the new technologies that are used in our cars, so we selected to develop a simple application that would allow the user to project his vehicle data to the infotainment system in real-time. We believe that many people will find this application useful since monitoring your vehicle's data in our days is extremely hard and the provided information from the manufacturers is limited.

As we can see from the diagram, the architecture is simple. We selected an OBD reader application that suited our needs, and we developed a set of classes along with a simple database in order to get the data and project them to Android Auto. The OBD reader had to be able to connect to OBD devices with Bluetooth and Wi-Fi and most importantly it had to be opensource.

Our application can be divided into three distinct layers. The first layer is the hardware layer, and it is where our data originates, processed and also where data will be displayed. The next layer is the Data layer that describes the properties of the car. Data in the application begin with one form and transition to different forms until they are presented. Lastly, the Functional layer where the Data and Hardware layers tie in together to give the user a simple UI that in the background handles multiple processes.

Android Auto does not allow for these types of applications to be developed and used and as we discussed before the only allowed applications for development are media applications and messaging applications.

After experimenting with both types of applications, we settled for the Media application. To be clear both types of applications could be used. The reason we chose a media type application is because it allows us to show all the data at once on the infotainment system in a more organized way rather than showing notifications containing data in them. Another advantage is that we can organize our data in different categories with a media application.

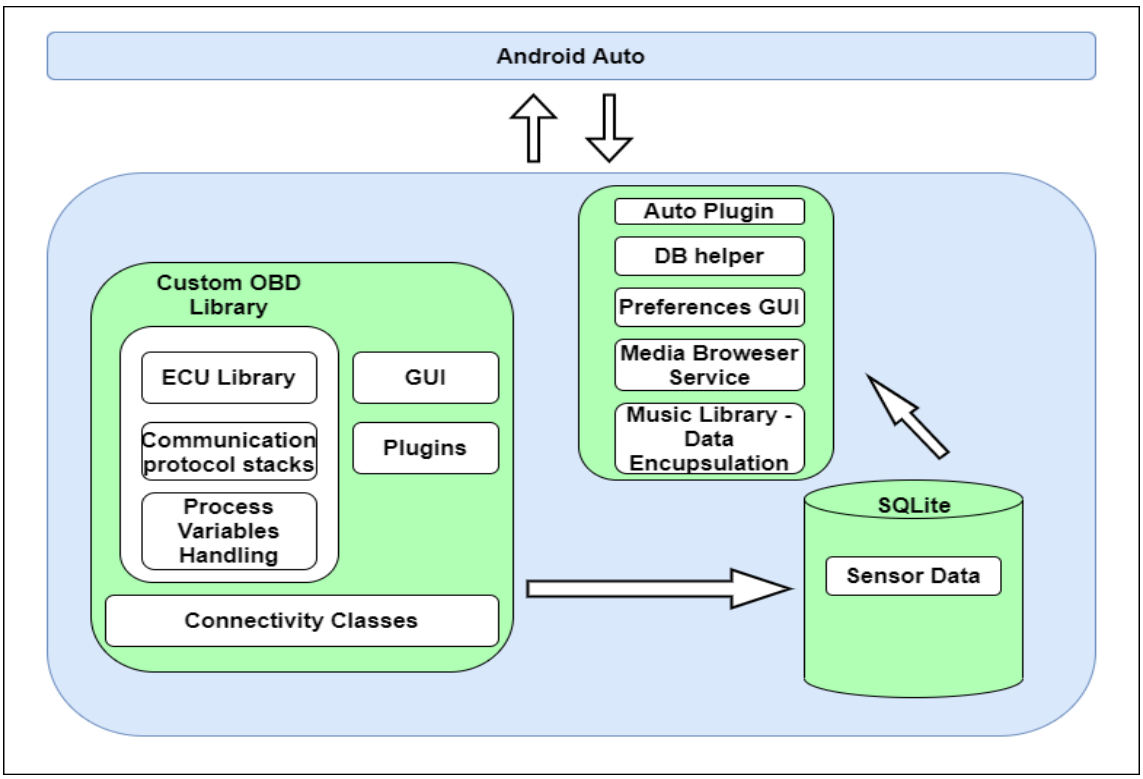


Figure 5-1 AID Architecture

5.2 Hardware Layer

The heart of our application lies in the hardware layer. This is where everything begins. The car for our application is a piece of hardware. Here is where all the data are produced. The next hardware piece is the OBD2 reader itself. This device is what allows a connection between the car and the smartphone running the application. This will handle the transmission of instructions from the phone to the CAN bus of the car in order to get the desired data properties. Then through the CAN bus, the OBD2 reader will receive the response and transfer it back to the phone where it will be further processed. Last but not least, the smartphone as a hardware piece. The smartphone must be able to process the data but also work with Android Auto and this is the reason why it must be able to run Android 6.0 and higher.

5.3 Data Layer

We can say that the data is the main point of the application. The data layer is what the application processes and shows to the user. This layer of the application is comprised of the database. Data are received from the OBD2 device and then are stored in our database. From the database, when the time for an update comes, the data will be retrieved and transformed into a new form in order for the Android Auto service to receive and display them. The database must be able to handle the large amount of data that are both stored and requested in short periods of time.

5.3.1 SQLite

For our application, we used the supported database for Android, SQLite. This is a relational database that provides ACID (Atomicity, Consistency, Isolation, Durability) properties. The application uses this database to store the received data from the OBD device on the car and to read the latest values for each item to be projected for the car's screen.

The application receives data items from the OBD device, which have certain properties that include the name of the ECU data property and the value. An event listener waits for an update and receives an object containing all the data for an ECU property. For example, this can be the current vehicle speed or coolant temperature. After this object is handled by the application in order to be able to show it on the phone's screen, it is stored in the database and when the timer for updating the vehicle's screen expires, the items are read from the database to be further processed and be projected.

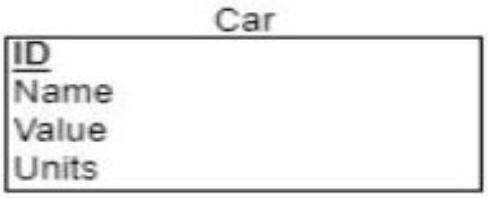
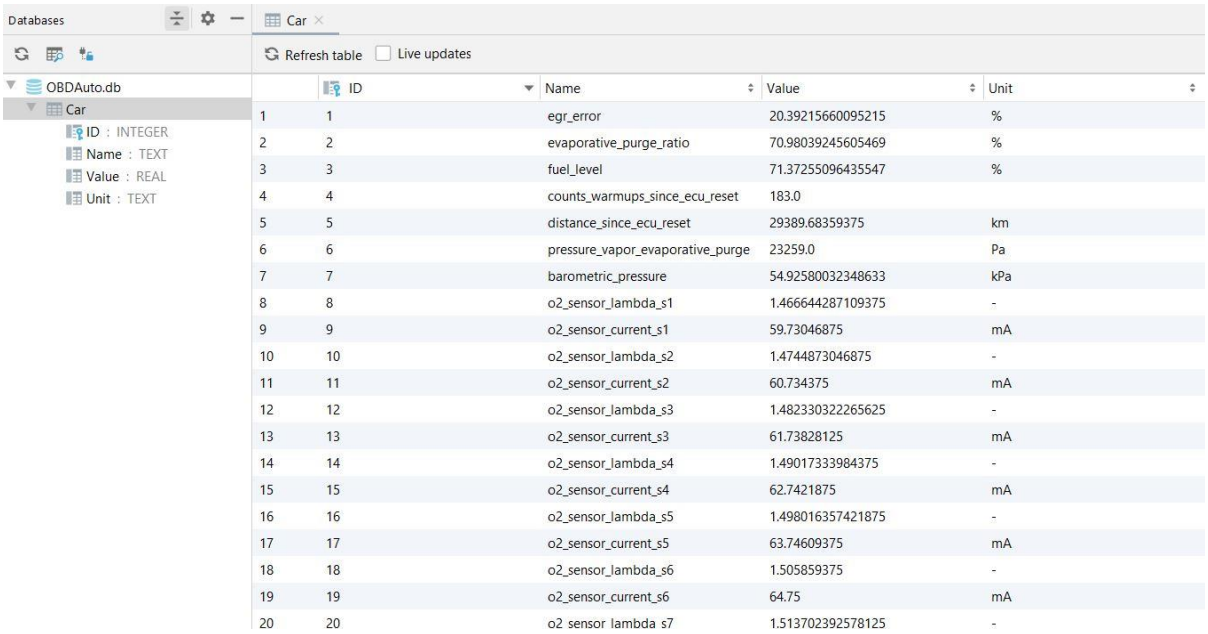


Figure 5-2 Database table structure

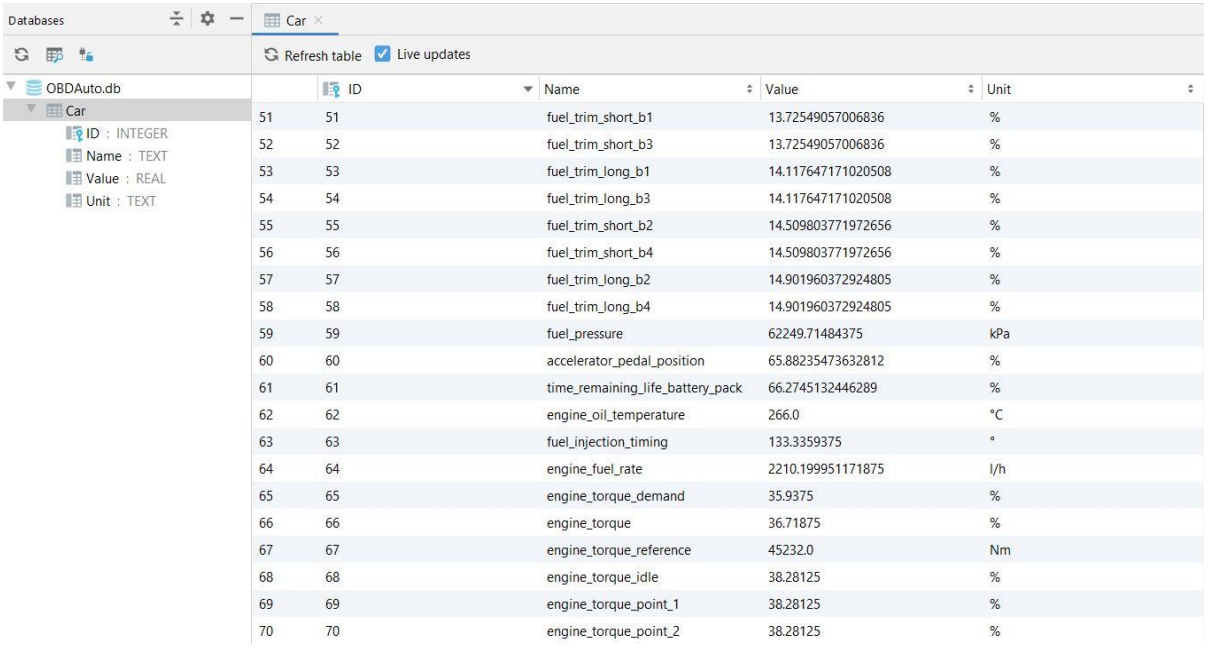
The above figure shows our database's schema which contains only a simple table. The table keeps a unique ID for each item, the name as text, the value as a float and the unit of measure as text. At first initialization, the application will create the database and when an item is encountered that is not present in the table entries, a new entry will be created with a unique ID and the current value. We chose to keep a very simple database schema without creating multiple tables, which for example could be a table for each data item that could keep the history of values for each session. This is done to limit the amount of space the application's data would take from the phone. If we kept a history for every item separately, a small session of using the application would result in a huge amount of storage space used because the application can track over 120 unique properties of the car and since data updates are received in real-time, the storage requirement could quickly get out of control.

On the other hand, the design of our application can be adopted easily in order to accommodate a more complex database that could keep a brief history of the values for future use or output to a file for further examination.



	ID	Name	Value	Unit
1	1	egr_error	20.39215660095215	%
2	2	evaporative_purge_ratio	70.98039245605469	%
3	3	fuel_level	71.37255096435547	%
4	4	counts_warmups_since_ecu_reset	183.0	
5	5	distance_since_ecu_reset	29389.68359375	km
6	6	pressure_vapor_evaporative_purge	23259.0	Pa
7	7	barometric_pressure	54.92580032348633	kPa
8	8	o2_sensor_lambda_s1	1.466644287109375	-
9	9	o2_sensor_current_s1	59.73046875	mA
10	10	o2_sensor_lambda_s2	1.4744873046875	-
11	11	o2_sensor_current_s2	60.734375	mA
12	12	o2_sensor_lambda_s3	1.482330322265625	-
13	13	o2_sensor_current_s3	61.73828125	mA
14	14	o2_sensor_lambda_s4	1.49017333984375	-
15	15	o2_sensor_current_s4	62.7421875	mA
16	16	o2_sensor_lambda_s5	1.498016357421875	-
17	17	o2_sensor_current_s5	63.74609375	mA
18	18	o2_sensor_lambda_s6	1.505859375	-
19	19	o2_sensor_current_s6	64.75	mA
20	20	o2 sensor lambda s7	1.513702392578125	-

Figure 5-3 Database instance



	ID	Name	Value	Unit
51	51	fuel_trim_short_b1	13.72549057006836	%
52	52	fuel_trim_short_b3	13.72549057006836	%
53	53	fuel_trim_long_b1	14.117647171020508	%
54	54	fuel_trim_long_b3	14.117647171020508	%
55	55	fuel_trim_short_b2	14.509803771972656	%
56	56	fuel_trim_short_b4	14.509803771972656	%
57	57	fuel_trim_long_b2	14.901960372924805	%
58	58	fuel_trim_long_b4	14.901960372924805	%
59	59	fuel_pressure	62249.71484375	kPa
60	60	accelerator_pedal_position	65.88235473632812	%
61	61	time_remaining_life_battery_pack	66.2745132446289	%
62	62	engine_oil_temperature	266.0	°C
63	63	fuel_injection_timing	133.3359375	°
64	64	engine_fuel_rate	2210.199951171875	l/h
65	65	engine_torque_demand	35.9375	%
66	66	engine_torque	36.71875	%
67	67	engine_torque_reference	45232.0	Nm
68	68	engine_torque_idle	38.28125	%
69	69	engine_torque_point_1	38.28125	%
70	70	engine_torque_point_2	38.28125	%

Figure 5-4 Database Instance 2

5.4 Functional Layer

The functional layer of the application is responsible for all the processing required in order to retrieve data from the OBD2 device and transform them into a user-friendly form to be projected on the vehicle's infotainment system. This layer will retrieve everything from the data layer and encapsulate the data into the form required from the Android Auto service. It is important that this is done in the most efficient way and this was our main consideration when designing the parts which this layer consists of. These are the MusicLibrary class where the content hierarchy is built, the MediaBrowserService that communicates with the Android Auto service, and our AutoPlugin class which takes the user preferences.

5.4.1 Music Library – Vehicle Data

In order to accomplish the projection of data to the vehicle's infotainment system, we had to encapsulate all of our data in Media Items. This was done by creating a class that would act as a Music Library and that would be responsible for updating the list of media items sent to our service.

The class contains a hashmap where the key is a string indicating the level, which can be for example the root that contains everything and in it we can find the keys for the selected items from the user or the key that holds all the data items. The value of every key is a Linked List that holds the actual media items.

To create our media items, we must create an object that describes each item. We use a MediaMetadataCompat object to which we will add a media id key and its value will be the name of the data item we want to show. We use the key for the artist to add the value of the data item and assign an artificial duration. Lastly, for the title of the media item we again use the name of the data item we want to project. This configuration will result in a description of a media item that holds the information received with the OBD device from the car. An advantage of the configuration is that when using the app on the infotainment system, the songs shown from the app which are our data items will have as

name the name of the property of the car and bellow that the value and as a result will be easily readable.

On initialization, this class will create one linked list that holds media items. In it will be created and stored two browsable media items meaning that those act as albums that can hold music in them. The first item is used to store all the data items received by the OBD device on the car and the second item stores the data items selected by the user to be shown in a different list for convenience. Lastly, this list is added to the hashmap and acts as our root. The media IDs of these two media items will be used to indicate the ID of the list selected by the user to find it in the hashmap.

When it is time to add data items to our lists or update them, we will connect to the database and receive all items with their respected values. We have a choice to either update the “all items” list or update the “selected items” list. This is determined by the parent key passed to the method which shows which list the user selected to view.

5.4.2 Media Browser Service

For our application to be recognized and function as a media application it must implement a MediaBrowserService. This service will determine the way our media items will be displayed on the infotainment system and will handle instructing the updates and sending the lists of media items for display.

We implemented the two necessary methods of the service as we explained before in Chapter 3. For the onGetRoot() method, we need to verify that the request is coming from a verified source meaning and return the root id along with a bundle containing information on how the browsable and playable items will be displayed. Our browsable items are the All Items and Selected Items. Playable items are all of our data items converted to media items.

Android Auto will then request this root from OnLoadChildren to get the hierarchy of content and display it. This method will be called by Android Auto every time a different level of the hierarchy is requested which means every time the user

requests a different sub-menu. When this method is called for our application, we will call our library's update method in order to refresh the items with the new items and return the result. It is important to note that we implemented an extra way of updating these items with the use of a timer. The user can alter the interval between updates on the infotainment screen. When the timer expires an intent is sent to our service that activates the method notifyChildrenChanged which will cause the service to fetch the content with the specified ID.

5.4.3 Auto Plugin

Auto Plugin is a service we created that handles specific tasks for our extension. In order to be able to get the changes that a user makes to the "Selected Items" list and the update period, we had to implement shared preferences. This service implements a listener and whenever a shared preference changes it triggers a call to the method onSharedPreferencesChanged(). Here we check what preference has changed and if it concerns this service, it will inform the MediaBrowserService through a broadcast to act and refresh the content for the infotainment system.

The service also implements a timer. To do this we created a thread that sleeps for the specified time of the update period set by the user. Whenever the time expires, the thread wakes up and sends a broadcast to the MediaBrowserService in order to update the data for the Android Auto.

Chapter 6

Experimental Methodology and Evaluation

6.1 Methodology

In order to evaluate our application's performance, we had to test both versions of the application, one without the functionality of projecting to Android Auto and the one we extended to include this functionality. Both versions of the application had to be tested in the same scenario, executing the same procedure and showing the same activities. The only difference between the two applications is that our version executes more instructions to create the necessary objects for the Android Auto projection. The extended version of the application was tested using the Desktop Head Unit (DHU) that Google provides.

To use the DHU, first we had to set up the phone. Developer Settings have to be both enabled on the phone and in Android Auto application installed. Then in the Android Auto application, we have to go in the Developer settings, enable unknown sources in order to recognize our application, and then start the head unit server. This is done because our application is not published in the Google Play Store and therefore it is considered to originate from an unknown source since it is not signed. In addition, we have to turn on the Head Unit Server on the phone through the Android Auto application which allows the phone to project on the computer rather than a real car. On the PC, all we must do is forward the port for the Android Debug Bridge to 5277 and then start the DHU. When

the DHU starts, it will show all applications that are installed on the smartphone and have the functionality for Android Auto.

To find the differences in performance between the two applications, we used the Android Profiler which is included with Android Studio. This utility allows us to monitor CPU usage, Memory, Network, and Power Consumption. It also allows us to trace the methods executed by our application and shows us exactly when a thread gets execution time and what it calls.

6.2 Performance

First of all, we had to test the application without our extension to establish a baseline. This is going to determine how efficient our extension is and what is the overhead applied. Our findings show that the application by itself is very efficient since it is not utilizing much of the CPU on a modern smartphone, and only uses about 3 – 5%. In Figure 6.1, we show the graph taken while using the application in this form. The darker shade indicated the CPU utilization by the application, the lighter shade is the background usage, and the dashed line is the total thread count. We can see that the CPU usage by the application is stable throughout the test and does not exceed 5%.

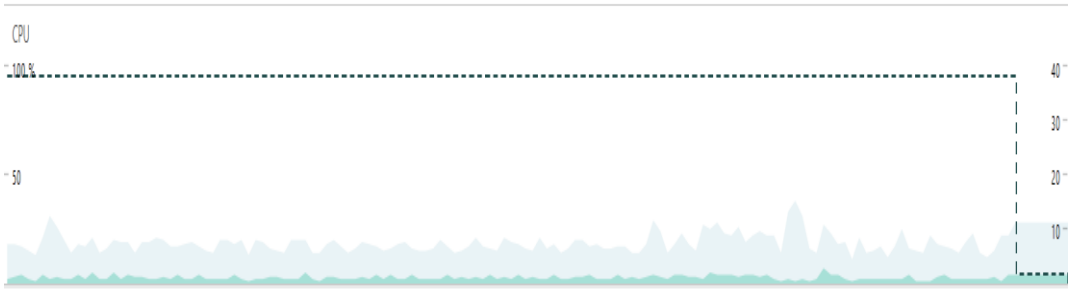


Figure 6-1 Normal CPU usage without extension

Using our version of the application and being connected to Android Auto using the DHU in order for our services to be running and projecting on a computer, we observed a spiking in CPU utilization in the order of 7-13%. These spikes are understandable since they happen whenever the application has to fetch new information from the database and create all the data items to display them to the infotainment system.

In Figure 6.2 we can see the behaviour described above with the CPU spikes up to 13% while updating the data items.

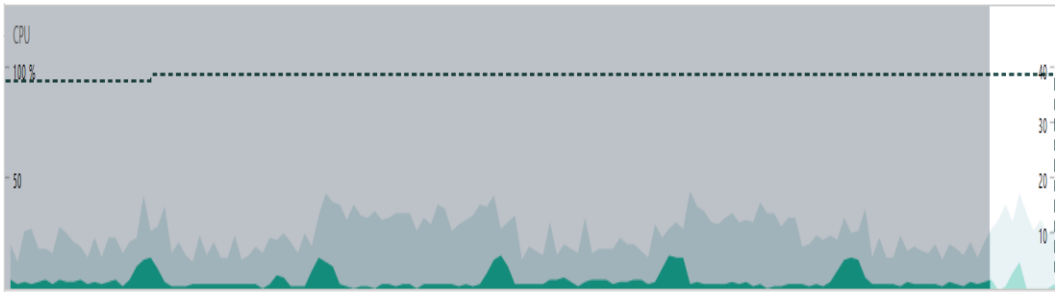


Figure 6-2 CPU behaviour, dashed line indicates thread count, green indicates AID application

When the timer expires the MediaBrowserService will get notified in order to retrieve the updated data from the database. This is when the spike of CPU usage happens. The application retrieves all the items stored in our table along with their values and then encapsulates them in MediaItem objects. Not only that, while this is happening the application continues to retrieve new information from the OBD device and updated the database. This was confirmed using the Java Method Trace Recording utility provided with the Profiler and it showed that every time that we had an update, these spikes occurred. Figure 6.3 illustrates the trace of methods called during a spike.

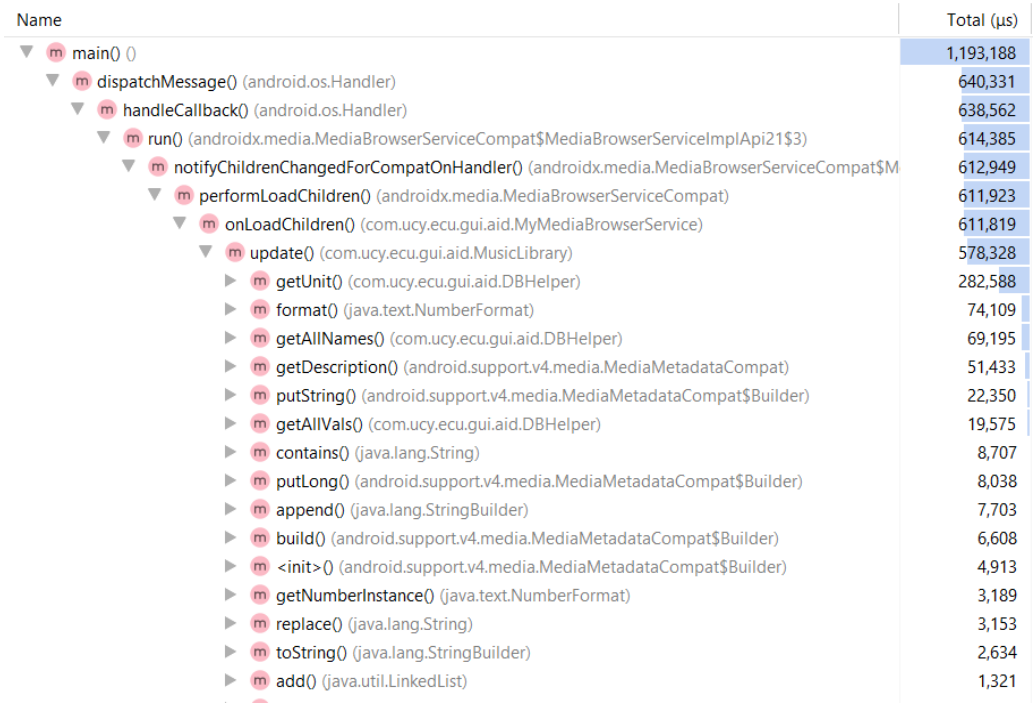


Figure 6-3 Shows the trace of methods called during a spike/update

Between updates, the CPU utilization was normal since the application was running exactly like it didn't have the extension. The only minor difference is that it stores the information in the database. It is important to note here that although a spike of utilization was occurring, nothing was noticeable on the phone nor the infotainment's screen, and the experience remained seamless. In Figures 6.1 and 6.2 we can clearly see the difference in CPU usage. In Figure 6.3 we can see that the methods of our extension are called, whenever an update occurs and this results in a short spike of CPU utilization.

In addition to the CPU utilization being higher, memory usage was also higher. This is again reasonable since we have to retrieve our data from the database and then create the MediaItem objects from them. These objects are then stored in a data structure (Linked List and then HashMap) and remain in memory so that they can be sent to the vehicle's infotainment system for projection.

6.3 Real-World Evaluation

Our application was also tested in a real-world scenario connected to a car to cross-reference our experimental findings we got by using the provided Desktop Head Unit (DHU). The test was carried out following all the traffic rules and most of the interactions were made by the passenger. The driver made minimal interactions to the application, as this is the intended use of it, and was never distracted from the road.

A smartphone was connected to an Android Auto enabled vehicle with a wired connection. Then using our application, we connected via Bluetooth to an OBD2 device that was connected to the car. After the connection procedure was completed and we made sure that we were receiving data on the smartphone, we set out parameters for selected items and update period. When this was complete, we only used the infotainment's screen to interact with the application.

Overall, the interaction was very responsive and consistent. We did not experience any lagging or crashing, and the screen was updating at the pre-set period. The vehicle's infotainment system displayed real-time data that were otherwise hidden. These data properties include O2 sensor values, intake air temperatures, and fuel trim. We made sure to test both lists of the application over an extended period of time.

We found that the "Selected Items" list was very easy to use and more desirable. The objects displayed on the screen were not distractive and easy to read allowing the driver to stay focused on the road.

On the other hand, the "All Items" screen contains too much information that results in the user having to scroll through a lot of items to reach the desired properties. This is solved using the "Selected Items" screen but, if the user selects a lot of data properties to be shown in that list, it will result in the same situation. In addition to this, when scrolling for an extended period, Android Auto assumes that the driver is not focused on the road. This results in the service pausing the projection for a small duration and showing a message, to force the driver to concentrate on the road and not on the screen. Another drawback is that even though we have an update period set, the OBD device is not

receiving all property values at once and this results in some of the data items being outdated by a small margin. The application still updates on the period that was set but the only data available are outdated.

Overall, AID offers a very good experience and accomplishes its purpose. It is not distractive, easy to configure, and use. It offers a solution to people that want to see more of what is happening behind the scenes in their car with some degree of accuracy. Through testing our application, we found it very interesting and intriguing to be able to monitor the vehicle's behaviour in many different situations. Bellow in Figure 6.4 and Figure 6.5 we show the application during testing on the vehicle.

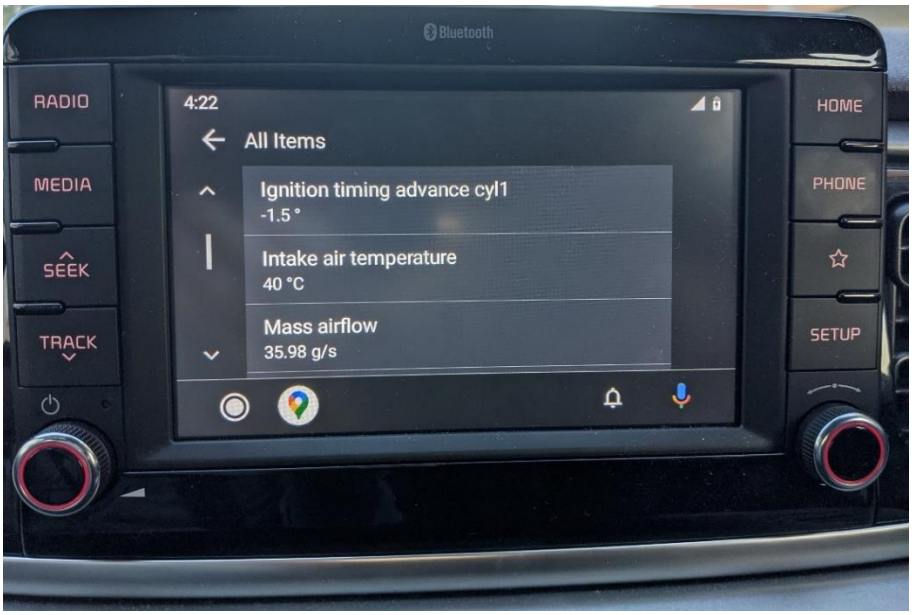


Figure 6-4 Test on a real car. All items list

Figure 6.4 illustrates the application used in the vehicle used for testing. In this figure, we can see the screen for the “All Items” list. From this figure, we can clearly see the length of the list by the scroll bar and conclude that it is too long to use while in a vehicle.

Figure 6-5 Test on car. Selected Items screen

Figure 6.5 shows the screen for the “Selected Items” list on the car used for testing. Looking at this figure we can conclude that this list is much easier to use while in a vehicle since it is shorter and thus easier to find an item of interest to monitor.

Chapter 7

Conclusion and Future

7.1 Conclusions

In this thesis, we set out to learn about the latest evolutions of our everyday cars that use software to provide a better user experience. We have presented the state of the art systems used in vehicle's today, the underlying network and protocols, how these software solutions are developed to ensure safety and get certified. We analysed in detail the system we chose to develop an application for. Then we presented our application that we designed and works with Android Auto, called AID. This application provides an OBD2 monitoring platform for Android Auto and currently, there is not an application on the market with its features. The goal with AID was to present a simple and useful application that shows the functionality and possibilities that Android Auto provides for developing modern applications to be used in vehicles.

It is apparent that today, more and more software is making it into our cars as they evolve. We live in an era where we can select our cars based on what software features it offers. It is of utmost importance that the code that is running in vehicles is designed, developed, and tested in a way that prioritizes the safety of the passengers. In addition to this, the certification of this software must be mandatory. Even though the ISO 26262 has been defined and is constantly updated, certification in the automotive world still needs to be more strictly defined. Applications developed for these systems from individuals also have to be tested and evaluated by the responsible people managing them.

Auto Insights Dashboard is a very useful application for people that want to view more information about their vehicle on their infotainment system. It provides an intuitive user interface and allows the user to create his own list of properties that his interested in. In addition, it shows many capabilities of the systems running in our modern cars and especially in Android Auto.

7.2 Future Work

Auto Insights Dashboard is currently in a very good position regarding the development. It provides everything that was set as a goal and was tested extensively. For the future, AID can be extended to provide even more features to the user.

One potential development could be the addition of user-specific lists. This would allow the user to create an arbitrary number of lists with his preferred items in each of them. This is a very important feature to be implemented since it clears the clutter of the "All Items" list and will show the data in a more organized way, further avoiding driver distraction.

Furthermore, a machine learning algorithm could be implemented in order to recognize abnormal behavior and inform the user of an imminent failure. This would be very useful since it would recognize a failure early on and could save time for the passengers and allow them to act faster.

Lastly, applications like AID could be developed using the platforms analysed in this Thesis. Using the platforms and features we explained, more complex and useful applications can be developed that will be running in our everyday vehicles. Some of the platforms we discussed in this Thesis offer even more functionality than Android Auto and this can be used to the user's advantage.

References

[1] C. Areias, J. C. Cunha, D. Iacono and F. Rossi, "Towards Certification of Automotive Software," 2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, 2014, pp. 491-496, doi: 10.1109/ISSREW.2014.54.

[2] A. B. Hocking, J. Knight, M. A. Aiello and S. Shiraishi, "Arguing Software Compliance with ISO 26262," 2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, 2014, pp. 226-231, doi: 10.1109/ISSREW.2014.88.

[3] H. Yu, C.-W. Lin, and B. Kim, “Automotive Software Certification: Current Status and Challenges,” SAE International Journal of Passenger Cars - Electronic and Electrical Systems, vol. 9, no. 1, pp. 74–80, 2016.

[4] Android Auto. Available at: https://www.android.com/auto/

[5] Android Automotive. Available at: https://source.android.com/devices/automotive

[6] Android Automotive Architecture. Available at: https://source.android.com/devices/images/vehicle_hal_arch.png

[7] ASIL levels of ISO 26262. Available at: https://www.aptiv.com/insights/article/what-is-asil-d

[8] Automotive Grade Linux. Available at: https://www.automotivelinux.org/

[9] AUTOSAR. Available at: https://www.autosar.org/

[10] AUTOSAR architecture. Available at: https://autonom.medium.com/autosar-fundamentals-what-is-autosar-part-1-ac6198c4b075

[11] CAN Frame Structure. Available at: https://www.can-cia.org/can-knowledge/can/can-data-link-layers/

[12] ISO 26262. Available at: https://www.iso.org/standard/68383.html

[13] MISRA. Available at: https://www.misra.org.uk/

[14] OBD2 Frame Structure. Available at: https://www.csselectronics.com/screen/page/simple-intro-obd2-explained/language/en

Appendix A

This is the code for the notification manager that we created to be used with our sample messaging application. This code generates the notifications that are going to be sent to the user and shown on Android Auto. The application first will call the function notifyUser() and the notification will be created through the subsequent call to the notificationBuilder() function.

```
public class NotificationManager {

    private static final String ACTION_REPLY = "com.example.REPLY";
    private static final String ACTION_MARK_AS_READ = "com.example.MARK_AS_READ";

    private static final String REMOTE_INPUT_RESULT_KEY = "reply_input";

    private Context mContext;
    private NotificationManagerCompat notifManCompat;
    private NotificationCompat.Builder notifBuild;

    public NotificationManager(Context context) {

        mContext = context;
        notifManCompat = NotificationManagerCompat.from(mContext);
        notifBuild = new NotificationCompat.Builder(mContext, CHANNEL_1_ID);

    }

    public void notifyUser(int conversationId, String conversationSubject,
                           String contactName, String message) {

        notifBuild = new NotificationCompat.Builder(mContext, CHANNEL_1_ID);
        Notification notification = notificationBuilder(
                conversationId,
                conversationSubject,
                contactName,
                message);

        notifManCompat.notify(0, notification);
    }
```

```
private Notification notificationBuilder(int conversationId,
        String conversationSubject, String contactName, String message) {

    //add title and message in notification
    notifBuild
            .setContentTitle(conversationSubject)
            .setContentText(message);

    //create reply intent
    Intent actreplInt = new Intent(mContext, NotificationManager.class);
    actreplInt.setAction(ACTION_REPLY);
    actreplInt.putExtra("ConvID", conversationId);

    RemoteInput rm = new RemoteInput.Builder(REMOTE_INPUT_RESULT_KEY).build();
    //mark as read intent
    Intent markRead = new Intent(mContext, NotificationManager.class);
    markRead.setAction(ACTION_MARK_AS_READ);
    markRead.putExtra("ConvID", conversationId);

    //create simple icon from resource to use on person
    IconCompat ic = IconCompat.createWithResource(mContext, R.drawable.user);
    //create pending intents for the 2 actions
    PendingIntent pi = PendingIntent.getService(mContext, 2, actreplInt,
                                                PendingIntent.FLAG_UPDATE_CURRENT);
    PendingIntent markPi = PendingIntent.getService(mContext, 1, markRead,
                                                PendingIntent.FLAG_UPDATE_CURRENT);

    //create a fake person for the conversation
    Person pb = new Person.Builder().setName("User2")
            .setIcon(ic)
            .setKey("1")
            .build();
    //get current timestamp in long
    Timestamp tm = new Timestamp(System.currentTimeMillis());
    long timestamp = tm.getTime();
    //create a messaging style and set title, message and timestamp
    NotificationCompat.MessagingStyle style = new
                                            NotificationCompat.MessagingStyle(pb);
    style.setConversationTitle("Trial Mess");
    style.setGroupConversation(true);
    style.addMessage(message, timestamp, pb);
```

```
//create the 2 actions
        NotificationCompat.Action replyAction =
                new NotificationCompat.Action.Builder(R.drawable.email,
                                                                "Reply", pi)
                 .addRemoteInput(rm)
                 .setSemanticAction(NotificationCompat.Action
                                                .SEMANTIC_ACTION_REPLY)
                 .setShowsUserInterface(false)
                 .build();

        NotificationCompat.Action markaction =
                new NotificationCompat.Action.Builder(R.drawable.book,
                                                        "Mark As Read", markPi)
                 .setSemanticAction(NotificationCompat.Action
                                                .SEMANTIC_ACTION_MARK_AS_READ)
                 .setShowsUserInterface(false)
                 .build();

        //finalize the creation of the notification
        notifBuild.setSmallIcon(R.drawable.chat)
                .setStyle(style)
                .addAction(replyAction)
                .addAction(markaction);

        //return the built notificationcompat to notify
        return notifBuild.build();
    }
}
```

Appendix B

This Appendix includes the code used for the MediaBrowserService in the sample media application we showed. It also includes the code to create the custom MusicLibrary that is responsible to create the MediaItem objects for the service.

```
public class MyMediaBrowserService extends MediaBrowserServiceCompat {


    private MediaSessionCompat mSession;
    public static final String inten="Refresh";
    /** Declares that ContentStyle is supported */
    public static final String CONTENT_STYLE_SUPPORTED =
            "android.media.browse.CONTENT_STYLE_SUPPORTED";

    /**
     * Bundle extra indicating the presentation hint for playable media items.
     */
    public static final String CONTENT_STYLE_PLAYABLE_HINT =
            "android.media.browse.CONTENT_STYLE_PLAYABLE_HINT";

    /**
     * Bundle extra indicating the presentation hint for browsable media items.
     */
    public static final String CONTENT_STYLE_BROWSABLE_HINT =
            "android.media.browse.CONTENT_STYLE_BROWSABLE_HINT";

    /**
     * Specifies the corresponding items should be presented as lists.
     */
    public static final int CONTENT_STYLE_LIST_ITEM_HINT_VALUE = 1;

    /**
     * Specifies that the corresponding items should be presented as grids.
     */
    public static final int CONTENT_STYLE_GRID_ITEM_HINT_VALUE = 2;
    BroadcastReceiver mReceiver;
```

```
public void onCreate(){
    super.onCreate();

    mSession = new MediaSessionCompat(this, "MusicService");
    MediaSessionCompat.Token sessionToken = mSession.getSessionToken();

    setSessionToken(mSession.getSessionToken());
    mReceiver=new MyReceiver();
    IntentFilter filter = new IntentFilter();
    filter.addAction("RefChild");
    registerReceiver(mReceiver, filter);

    //register Callback

}

public void onDestroy(){
    if(mSession.isActive()){
        mSession.setActive(false);
        mSession.release();
    }
}
```

```
    @Nullable
    @Override
    public BrowserRoot onGetRoot(@NonNull String clientPackageName, int clientUid,
@Nullable Bundle rootHints) {
        Bundle extras=new Bundle();
        extras.putBoolean(CONTENT_STYLE_SUPPORTED,true);
         extras.putInt(CONTENT_STYLE_BROWSABLE_HINT,
                                          CONTENT_STYLE_GRID_ITEM_HINT_VALUE);

         extras.putInt(CONTENT_STYLE_PLAYABLE_HINT,
                                          CONTENT_STYLE_LIST_ITEM_HINT_VALUE);

        return new BrowserRoot("ROOT",extras);
    }

    public void onLoadChildren(final String parentMediaId,
                               final Result<List<MediaBrowserCompat.MediaItem>>
                                                                    result) {
        result.detach();
        MusicLibrary ms = new MusicLibrary();
        ms.createMediaMetadata(getApplicationContext());

            result.sendResult(ms.getMediaItems());

    }


    // use this as an inner class like here or as a top-level class
    public class MyReceiver extends BroadcastReceiver{

        @Override
        public void onReceive(Context context, Intent intent) {
           notifyChildrenChanged("ROOT");
        }
    }
}
```

```
class MusicLibrary {
    private static final TreeMap<String, MediaMetadataCompat> music =
                                                            new TreeMap<>();

public List<MediaBrowserCompat.MediaItem> getMediaItems() {

    List<MediaBrowserCompat.MediaItem> result = new ArrayList<>();

    for (MediaMetadataCompat metadata: music.values()) {
        MediaBrowserCompat.MediaItem x=new
                        MediaBrowserCompat.MediaItem(metadata.getDescription(),
                         MediaBrowserCompat.MediaItem.FLAG_PLAYABLE);
        result.add(x);

    }
    return result;
}
public void createMediaMetadata(Context context) {
        int i=0;
        DBHelper db = new DBHelper(context);
        ArrayList <String>array_list = db.getAllNames();
        ArrayList <Double>values=db.getAllVals();
        String img="";
        for(String names : array_list){
            if(names.contains("Voltage")){
                img="voltage";
            }
            if(names.contains("Temp")){
                img="temp";
            }
            if(names.contains("RPM")){
                img="speed";
            }
            float val=values.get(i).floatValue();
            music.put(names,
                    new MediaMetadataCompat.Builder()
                            .putString(MediaMetadata.METADATA_KEY_MEDIA_ID,names)
                            .putString(MediaMetadata.METADATA_KEY_ARTIST,""+val)
                            .putLong(MediaMetadata.METADATA_KEY_DURATION,
                                                                    100 * 1000)
                            .putString(MediaMetadata.METADATA_KEY_TITLE, names)
                            .putString(MediaMetadataCompat.
                                 METADATA_KEY_DISPLAY_ICON_URI,
                                                        getAlbumArtUri(img))
                            .build());
            i++;
        }
    }
}
```