POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master of Science Thesis in Computer Engineering

A vehicle Human-Machine Interface implementation based on Google Android Automotive OS

Supervisors

Prof. Gianpiero Cabodi

Prof. Danilo Vendraminetto

Candidate

Davide Cometa

List of Contents

List of Figures

List of graphs

Thesis summary and structure

Our experience in moving around in a car has changed so much in recent years thanks to constant technology innovations. Modern vehicles provide services that were almost unthinkable just 10 years ago. Continuous driving assistance, GPS-based navigation, in-car phone calls are just some of the possible examples and many other features are undergoing intense studies. The automotive industry is currently facing many challenges to integrate new features in the In-Vehicle Infotainment while respecting limitations concerning driver safety and system usability, but the absence of widely accepted standards adds complexity to this scenario.

This thesis work aims to present an Infotainment prototype implementation developed in *Magneti Marelli*, based on the new-coming Android Automotive OS. The target is to provide an easily extendible, customizable and multi-tasking system that meets safety requirements and takes advantage from the adoption of an open-source environment like Android, in contrast to the proprietary infotainment systems being released today.

Throughout this paper will be provided a description of initial system requirements, its hardware architecture elements and a more detailed depiction about the software architecture, design choices and implementation. Emphasis will be placed in the development of software components such as the service handling the messaging protocol to interface the Info-Cluster and a physical Rotary controller with the Android Automotive OS.

In particular this paper is structured as follows:

Chapter 1: This introductory chapter deals with the evolution of infotainment systems from their origins to modern times pointing out the growing importance of software in vehicles. In addition, it analyzes modern trends in Infotainment development evidencing the lack of standard approaches to

guarantee driver safety and the limits in extendibility that actual systems on the market offer.

Chapter 2: It aims in reviewing the Android Automotive environment, the operating system adopted for implementation of the Infotainment, describing the Android software stack and the advantages that its adoption for Infotainment development provides.

Chapter 3: This chapter explains the system high-level hardware architecture, a critical area that contributes to create a solid, reliable and safe to use system. Each of its components is described both from the technical and functional point of view.

Chapter 4: It outlines system requirements and focuses on the description of the software architecture and the implemented Model-View-ViewModel architectural pattern. As preparation for the subsequent chapter, a high view of the development process is also provided to clarify the tools and techniques that have been used throughout implementation phase to fulfill expected requirements in terms of features, usability, reliability and safety.

Chapter 5: The fifth chapter describes the design and implementation of new applications and customizations made to native ones.

Chapter 6: It analyses the performance of the implemented Infotainment in order to evaluate how much it can be further improved even in terms of adding new features that can require a higher amount of resources. This analysis will test system reactiveness and the reliability of the Info-Cluster, one of the critical components in a vehicle because it provides car related data useful also in guaranteeing users safety.

Chapter 7: Final chapter highlights strength and weakness points, deriving conclusive considerations on the results obtained. Finally it outlines future work ideas in order to further improve the system.

Chapter 1

Introduction

1.1 The origins of In-Vehicle Infotainment

A modern trend is to equip almost any technological device with a graphical user interface that allows humans and machines to interact in a clever and easy way by simplifying the interpretation of outputs and the input for commands. These are usually called Human-Machine Interfaces (HMI) and are essentially the natural evolution of the old PLCs (Programmable Logical Unit). The substantial divergence with respect to PLC, is not just in functionalities and possibilities that they provide, but also in the attention placed to User Experience and design while developing it, coming from years of researches in the field of ergonomics and human behavior.

In recent years, the design and development of Human-Machine Interfaces has become a fundamental process to win over competitors and gain a larger market share by attracting consumers.

Even in vehicles, HMIs deployment has been successful obtaining growing importance. For automakers is a way to make the various models on the list increasingly attractive and able to intrigue the younger generation too in search of a fully connected and technological car, that can nearly resemble the trendsetting smartphones experience.

The history of what today has gained the name of "In-Vehicle Infotainment" is not so recent and has initiated when software did not even exist in cars.

In 1930s the automotive industry begun to understand the importance of enhancing not just the driving experience but the living experience in cars by providing information, entertainment and safety to drivers and passengers.

At that time, car infotainment was just an AM Radio and nothing more. It evolved in the 1950s into record players with vinyl. They had the advantage to let the driver play whatever song he wanted to hear, but did not last long because of road bumps that affected audio quality. In 1960s, they were replaced by modern stereo using two audio streams instead of one and being able to play music from 8-track cassette tapes introduced in vehicles thanks to Ford and Motorola.

In the 1981, the very first in-car navigation system was introduced in Japan with the Toyota Celica. Meanwhile, the automotive industry witnessed the first deployment of software in cars to control the engine and, in particular, the ignition. The first software-based solutions were strictly local, functionally and technically isolated, and did not relate to one another. These independent and unconnected pieces of software (usually written in C or machine code) used to run on single dedicated controllers called Electronic Control Units (ECUs) that typically ran a few kilobytes code. Formerly, only a minimum of abstraction was applied and the focus was mainly on minimum resource consumption. [1]

In the early 2000s, the Bluetooth connectivity was introduced in cars, allowing hands-free calls and music reproduction from mobile devices. Even the introduction of touch screen systems integrated with GPS navigation started gaining popularity, but it was still far from today's concept of In-Vehicle Infotainment (IVI):

"The combination of vehicle systems that uses audio/video interfaces, touchscreens, keypads and other types of devices, as well as vehicle voice commands and other types of interactive audio or video." [2]

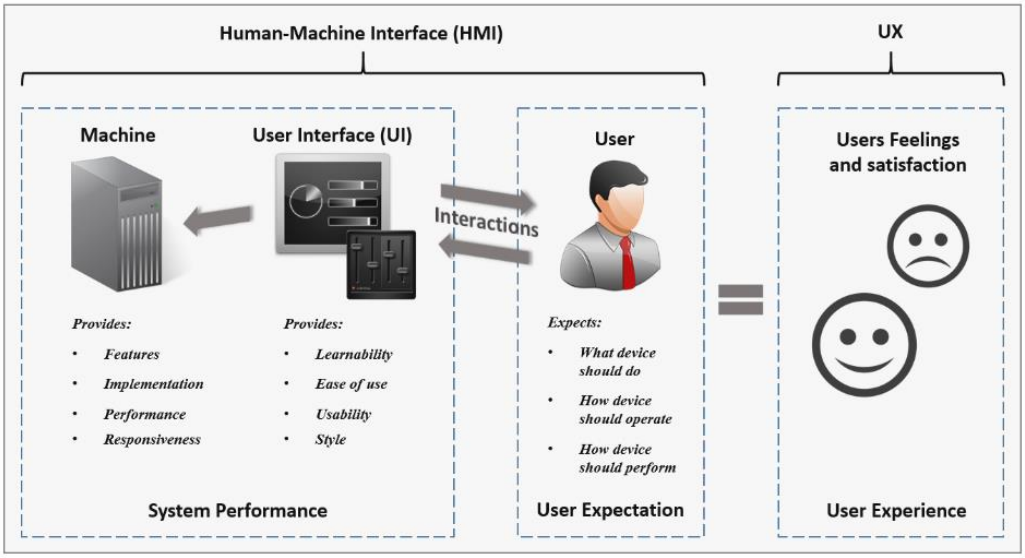


Figure 1. Diagram of UI, HMI and UX relationship

1.2 The growing importance of software in the Automotive Industry

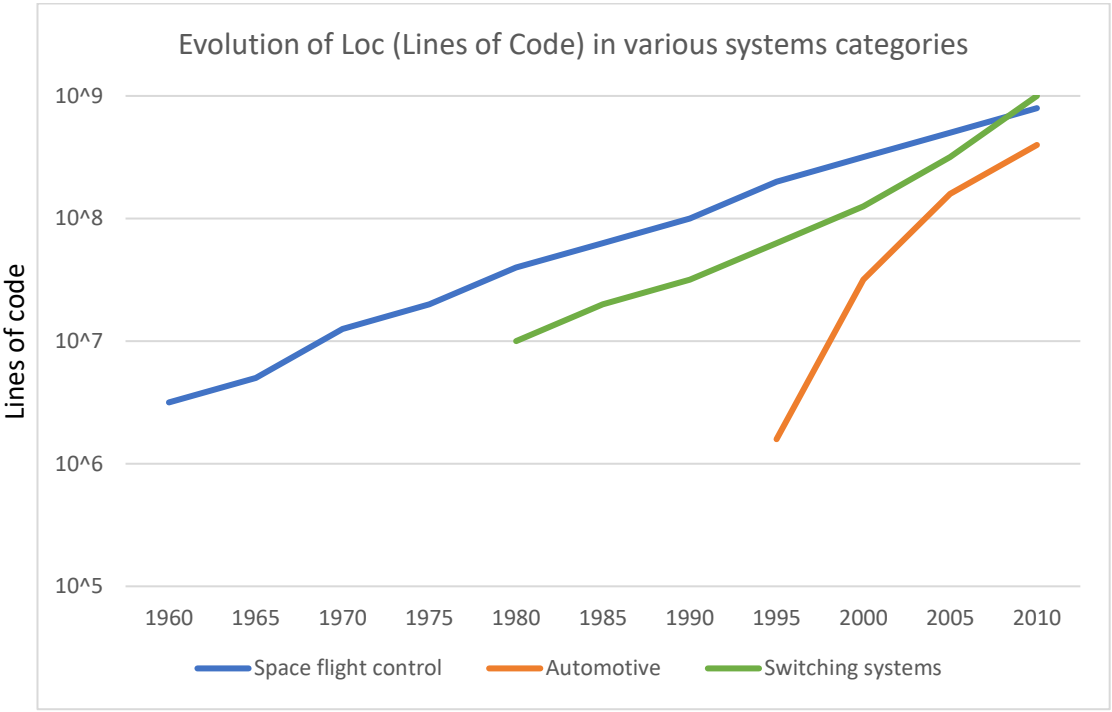
As time passed, software kept gaining importance in vehicle design.

Undoubtedly, without the help of software innovations, not just the infotainment as we know would not have been possible, but automakers could not easily meet tightening emissions standards, fuel-efficiency and most of the safety systems in cars would not exist. The estimation confirms that almost 90% of new cars functionalities introduced in recent years uses software solutions.

New high-end cars are among the most sophisticated machines on the planet, with 100 million or more lines of code, more than the Large Hadron Collider (about 50 million LOC) or even Facebook (about 60 million LOC). [3] In automotive industry software's size is growing faster than any other

system in modern years moving from hardware to software-defined vehicles.

A concrete example that shows the importance of software are the effort put in autonomous cars. Software actually represents 10 percent of overall vehicle content today for a D-segment, or large car (approximately $1,220), and the average share of software is expected to grow at a compound annual rate of 11 percent, to reach 30 percent of overall vehicle content (around $5,200) in 2030. [4]



Graph 1. Comparison of the evolution of LoC (lines of code) for various systems categories in past years.

This graph explains how, along with hardware and semiconductor evolutions, software is considered crucial for innovations in cars.

Contrary to this direction, software innovations in last decade did not focus on Infotainment, but mainly on safety systems like Anti-block Brakes (ABS), Traction Control or Electronic Stability Control (ESC).

IVI design has gained importance just in the recent years according to increased consumers demand for in-car technology:

> "Automotive HMI design has become a focal point and battleground for brand differentiation for all automotive segments. It is not only the arrival of the connected cars but it is perhaps more important the arrival of information technology and software that enables next generation multimodal, multi-touch and multi-zone HMI design. Safety remains the key cornerstone in HMI design. Minimizing driver distraction whilst optimizing the driver and passenger experience and at the same time exploiting the potential of new interaction schemes is a real opportunity for intelligent design concepts". [5]

Moreover, there are other several factors that are moving the demand; one of these are the governmental restrictions on using mobile phones while driving, which pushes consumers to search for hands-free ways of communication.

Definitely, software has become one of the backbones of the automotive industry. It powers applications from infotainment to advanced driver assistance systems and autonomous driving.

1.3 In-Vehicle Infotainment design paradigms for safety and distraction avoidance

The entertaining functionality of an In-Vehicle Infotainment does not fit the concept of safety that is one of the key aspects in vehicle design process.

Infotainment software design differentiates from other car systems modeling because it is the only human-machine interface in a modern car on which the user has full control (differently from all the other fully or partially automated systems such as the engine cooling system). For this reason, IVI design must take into consideration the user experience, easiness and safety of use along with the bundle of functionalities that it should offer through OEM or third- party applications.

Some studies revealed that even IVI could be origin of distractions if not well designed. A source of information to date on the causes of crashes comes from a 100-car study conducted by the Virginia Technology Transportation Institute. This study monitored 100 cars for 13 months using in-vehicle video cameras and extensive vehicle instrumentation. It recorded over 42,000 hours of driving, 761 near-crashes, and 72 crashes. Nearly 80% of all crashes involved driver distraction in the three seconds prior to the incident. Mobile phones and other in-vehicle driver controls were associated with the highest frequency of distraction-related crashes and near-crashes. [6]

According to a 2015 survey conducted by AT&T [7] with a sample of over 2,000 US respondents, "7-in-10 people engage in smartphone activities while driving".

Smartphone activities included:

- Text (61%)

- Email (33%)
- Surf the net (28%)
- Facebook (27%)
- Snap a selfie/photo (17%)
- Twitter (14%)
- Instagram (14%)
- Shoot a video (12%)
- Snapchat (11%)
- Video chat (10%)

Use of smartphones while driving has become one of the major issues for safety; this is why the automotive industry tried to integrate those devices in a safe manner. Most recent car infotainments allow smartphones and laptops to connect to the vehicle for hands-free passengers use, also implementing security features as preventing drivers from using any video services or other distracting system elements while vehicle is in movement.

When designing In-Vehicle Controls, visual and cognitive or even manual distractions should be prevented using appropriate design architectures and user interaction schemes. [8][9]

Today, there is no standard architecture or reference implementations for IVI to issue these and other problems. Different actors are trying to develop a scalable architecture that could merge together usability, safety and innovative features, increasing end-user satisfaction, reducing costs and defining some kind of paradigms to be followed.

One of these actors is the GENIVI Alliance, a community built in 2009 between automotive experts and industry leaders (even content providers or mobility companies) that are collaborating to produce adoptable standards and open source code.

In Europe, ESoP (European Statement of Principles) has established recommendations [10] for HMI design goals to meet some safety standards:

- The system supports the driver and does not give rise to potentially hazardous behavior by the driver or other road users
- The allocation of driver attention to the system displays or controls remains compatible with the attentional demands of the driving situation
- The system does not distract or visually entertain the driver (Visual entertainment may occur by visually displaying images which are attractive because of their form or content. It is of particular relevance in the driving context because of the importance of vision for safe driving).
- The system does not present information to the driver that could result in potentially hazardous behavior by the driver or other road users.
- Interfaces and interaction with them is intended to be used in combination by the driver while the vehicle is in motion are consistent and compatible.

In absence of legitimate standards, some common approaches can be found coming from different vendors and IVI development companies that aim to user safety while driving. They can be grouped in to the following three areas:

- *Avoid distraction through clever architectures and screens positioning*
- *Avoid distraction through clean UI design and interaction schemes*
- *Avoid distraction through fast input mechanisms*

1.3.1 Avoid distraction through clever architectures and screens positioning

For distraction avoidance, even hardware architecture can do the difference. Positioning displays and hardware buttons in a clever way can maintain cognitive attention and focus on the road while reducing IVI interaction times to what is strictly necessary.

In past years, infotainment consoles were located at the center of the dashboard. Displaying information just in the center console needed constant attention of the driver away from the on-road viewing area. This raised a concern regarding driver distraction due to adopted screens location.

Actually, the prevailing approach for IVI hardware architecture prescribes the usage of multiple digital screens: a cluster and a head-display.

Clusters are usually set in front of the driver behind the steering wheel ensuring reliable information delivering about vehicle data (such as speedometer, tachometer, temperatures, doors or lights status and various vehicle data usually taken directly from CAN bus), in addition to infotainment data summary that is a replication of main display information coming from radio, media or navigation.

They were born as analog tachometers but evolved as part of In-Vehicle Infotainment when small LCD screens have been placed as “add-on display”, along with electromechanical needles-based gauges, showing additional digital data.

In recent years, high-end cars adopted completely digital clusters that replace mechanical gauges with 2D-3D computer graphics and entirely managed by software. Current challenge is to give the user the same feel and experience of traditional clusters showing safety critical information that gets onto displays such as belt indicator, tire pressure level, temperatures or rear camera view along with miscellaneous information such as navigation, media been played or ongoing calls.

Figure 2. A digital cluster from Ford Mustang 2018

Cluster operating system usually is real-time based (an example is QNX Neutrino, a real-time Unix-like POSIX-compliant OS for embedded systems) different from the core O.S. on which the infotainment system runs. This is necessary because it needs to be lighter and faster then head unit that usually has slower booting times (this is true mainly at startup

because an Android system needs in average 40 seconds to boot). While slow boot time are acceptable for a mobile device that rarely gets shut off, it becomes a bigger problem in a vehicle. Since most people immediately begin driving after turning on the car, a long IVI system boot time would result in drivers pulling up a map or a play list while the vehicle is in motion – further adding to distractions while driving.

A solution to this problem is the use of virtualization that allows running multiple OSs on the same System-On-Chip (SoC) separated from each other. Besides centralized management and cost reduction due to less hardware devices, it provides memory, CPUs and peripheral sharing. [11] In this way a hybrid architecture controlled by a hypervisor, allows developers to build IVIs without compromising functionality, security or reliability of the vehicle's operation software. Critical components such as vehicle sensors, diagnostics, and emergency services would never be impacted by third-party application, as they would be completely enclosed within their own respective operative system. [12]

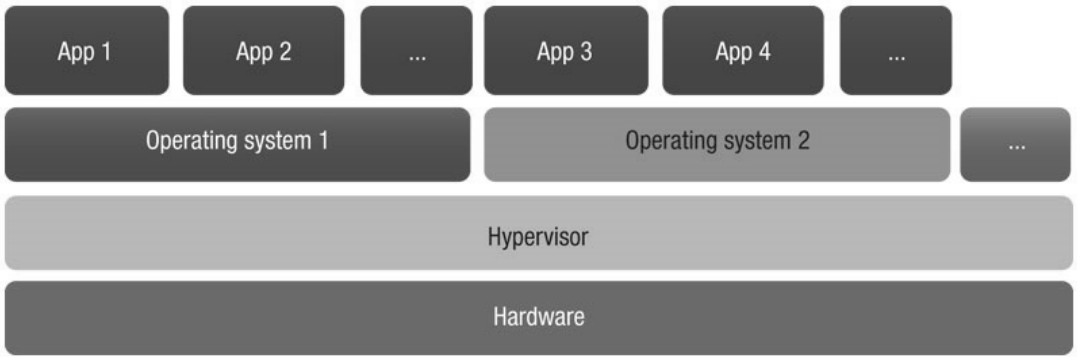


Figure 3. A type 1 Hypervisor architecture

1.3.2 Avoid distraction through design and clean interaction schemes

In any way, user interactions with the In-Vehicle Infotainment that requires sustained attention are dangerous and must be avoided because they can be source of distractions. An interface that has tiny, hard to find buttons, either physical or hidden under several layers of options or menus, is likely to get driver in a danger.

Thus, is important to design simple and clean user interfaces in terms of easiness to use and access, avoiding complex menus that could take driver's focus off the road. The main goal is to decrease user's brain load while using vehicle controls with the aim to help reaction to alarm signals quicker and understand the whole process better. [13]

Even in IVI User Experience Design, smartphones UXD is a good idea to be followed despite they see a higher number of interactions compared to IVIs; is better to model Infotainments through shortcuts for frequently used scenario reducing interaction time to few seconds.

Smartphones UI have come far away with the definition of several User Experience Design (UXD) paradigms. One of the latest and successful paradigm is the "Material Design", developed by Google. It redefines shapes, colors and element positions on the screen, based on psychology studies, in order to get a clean, easy and fast to use interface. [14]

1.3.2 Avoid distraction through fast input mechanisms

Common approach in vehicle applied user interactions prescribes the use of traditional buttons on the steering wheel or around the infotainment screen. A more recent trend is the adoption of touch sensitive screens in cars that could support multi-touch and gestures. Even though touchscreens allow faster interactions, they can bring to distractions while driving because they require visual attention of the driver.

For this reason, touch-physical controls hybrids have been implemented in all cars combining touch screens with buttons, knobs or even rotary or mouse-like controls. Physical controllers can improve the ease of scrolling through menus and, in addition to haptic feedback that recreates the sense of touch by applying forces, vibrations or motions to the user, they can drastically lower distraction possibilities. In recent years, even voice control systems have been deployed in cars thanks to improvements in technology's ability to understand human speech, which enormously helps drivers focus on the road.

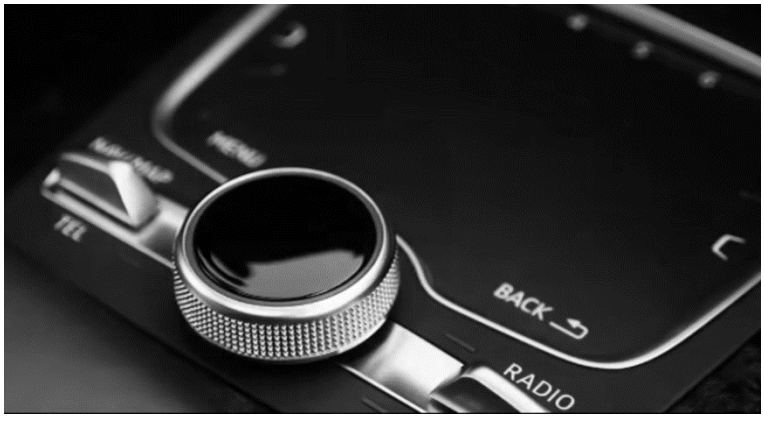


Figure 4. A rotary knob for In-Vehicle Infotainment interactions

1.4 From automakers Infotainments to software companies solutions

Since 2000s, the In-Vehicle infotainment has always been part of vehicle design for the automakers. They developed their own Infotainment systems from scratch and, observing actual market, there are several proprietary IVIs examples such us Uconnect from FCA, MyLink from GM or Sync by Ford.

The biggest drawback in having proprietary software is that it limits quicker innovations and makes the infotainment landscape fragmented. Even in Personal Computers manufactory, if each system had its own operating system would have caused that application developers needed to assure that software works with each version of them; moreover, PCs manufactures had to drastically limit the attention to hardware innovations. Instead, having a common system in many different vehicles reduces third-party application development complexity as it happens with mobile smartphones. This is what happened in recent years with In-Vehicle Infotainment market:

For most automakers, the development of a custom Infotainment OS is not worth the investment of time and resources. It requires significant researches and expertise in HMI design because it must be both intuitive and attractive. [15]

These issues initially drove the sight to demand the Infotainment development to external automotive components suppliers and then to adopt third-party technology company operating systems also in cars.

Software companies and other digital-technology players are leaving their current tier-two or tier-three positions to engage automakers as tier-one suppliers because In-Car Infotainment market is expected to garner $33.8 billion by 2022, pointing out how important this system is.

When it comes to software solutions, the biggest leaders in IT market will not stay outside.

Google and Apple, in the last years, developed their own in-car infotainment systems (respectively Android Auto© and CarPlay©) implementing smartphone projection modes via USB or Bluetooth connection. They enable mobile devices to be operated in vehicles through the dashboard head unit so that the vehicle occupants do not manipulate their devices directly, but use an interface they are familiar with, and spend more time with their eyes on the road. [16]

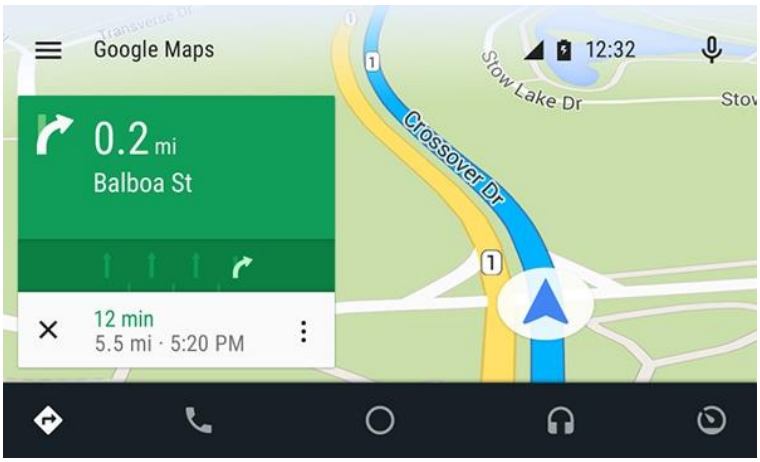


Figure 5. Google Android Auto

Figure 6. Apple CarPlay

These systems have been successful deployed on a wide range of vehicles from different automakers, taking the IVIs to a new level of functionalities and possibilities but losing the capability to retain brand identity through customization of the GUI. The lack of personalization that, in the past years, made each automaker competing for the best-looking and endearing HMI as part of the vehicle design is one of the biggest drawbacks. Moreover, they are just secondary interfaces that projects mobile phone content and for this inherently limited.

In this perspective, trying to overhaul aforementioned drawbacks, Google started to think a new open source operating system based on the mobile widespread mobile system Android. It provides extendible capabilities to allow users to install whatever application they want and developers to customize the entire system and native applications. In addition, it does not implies to have a mobile device connected to the car unless for phone calls guaranteeing less distractions while driving and increasing driver and passengers safety.

Chapter 2

Android automotive OS

2.1 Android Software Stack

Android Automotive OS, currently still a prototype, has been initially released in 2016 as a modified version of Android 7.0 N with the introduction of APIs for vehicle network interfaces and is being improved each year with new updates introducing additional features. The official Android Automotive OS development timeline in the following demonstrates how the system is becoming more and more integrated with the vehicle in order to completely the next operative system for cars:

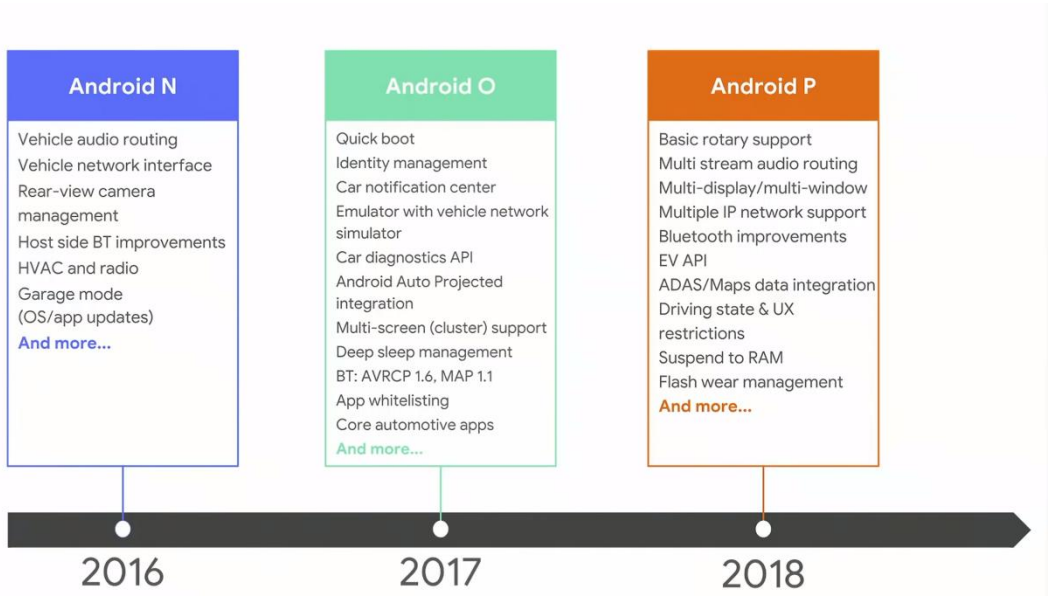


Figure 7. Android Automotive development timeline

It has been successfully installed in production infotainments as primary or even in conjunction with other operating systems such as Linux (through the hypervisor, that acts as a coordinator between guests OS on the same SoC. A further explanation about the hypervisor will be provided in the latter).

Android, from which automotive version derives, is a mobile operating system that provides mobility features along with multi-tasking, speech recognition/synthesis, connectivity and a 3D graphics engine based on the OpenGL library.

A scheme about the Android Software Stack is shown below:

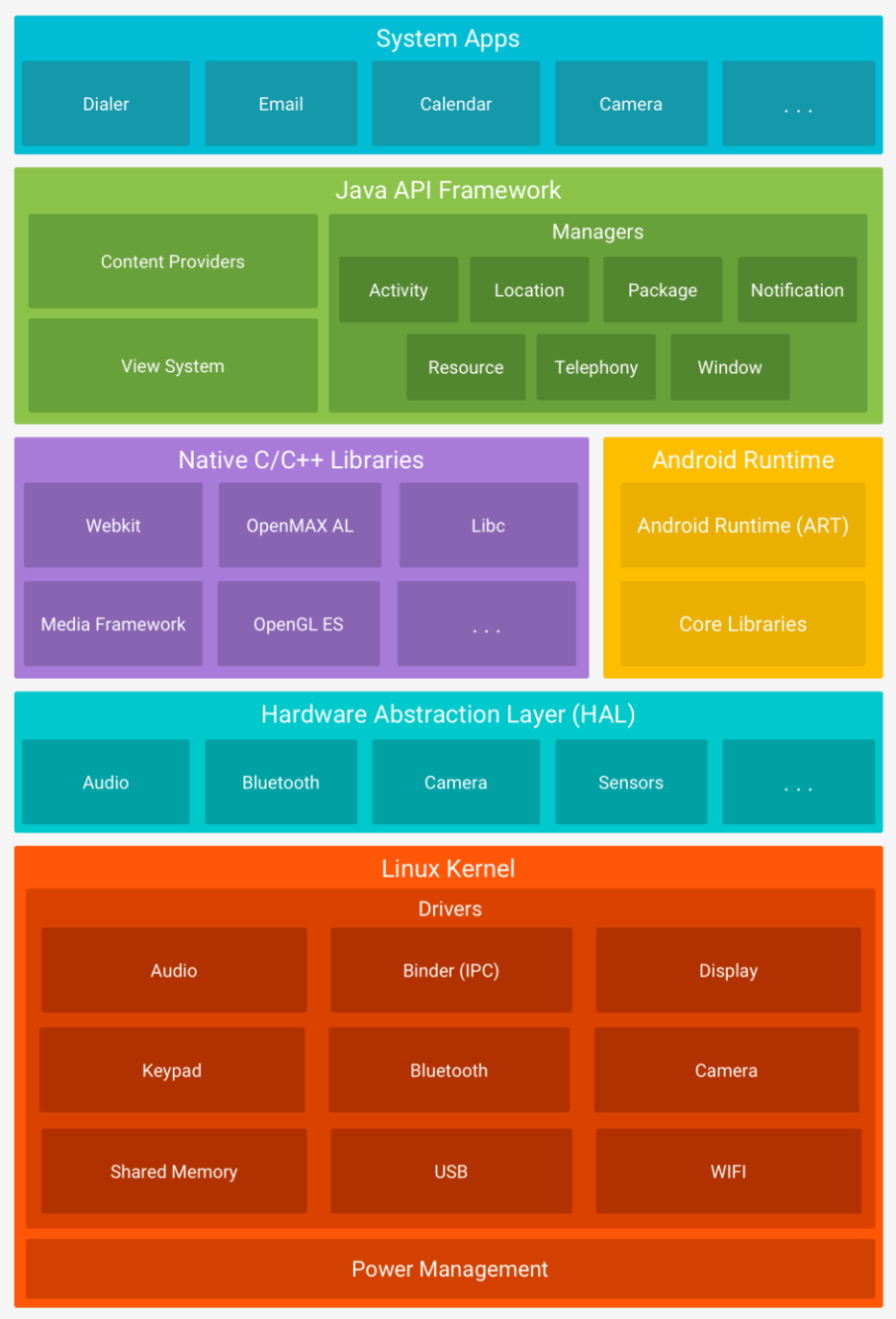


Figure 8. Android Software Stack

In the following a brief description for each layer of the Android Software Stack will be provided in order to understand which tasks they accomplish.

2.1.1 Linux Kernel

Android is based on the Linux Kernel that occupies the bottom part of the software stack; it is the core that provides the lowest level of abstraction for the hardware along with processes, memory, threading and hardware devices management.

According to Google Developers divulgement [17], the Linux Kernel has been slightly modified adding drivers, porting to ARM architecture and making low-level code changes. Some of the key changes are:

- Ashmem (Android Shared Memory), a memory sharing system based on files
- Binder, an inter-process communication system (IPC) and Remote Procedure Call (RPC)
- Logger, an optimized in-kernel logger
- Paranoid Networking, a mechanism to limit I/O on network for specific processes
- Pmem (Physical Memory), a driver for memory mapping in user-space
- Viking Killer, a substitute for OOM Killer that implements the Last Recently Used (LRU) logic in case of lacking free memory.
- Wavelocks, the Android default power manager.

2.1.2 Hardware Abstraction Layer (HAL)

The HAL is an abstraction layer for hardware communication between the Android framework and the Linux kernel libraries (drivers) regardless of the physical transport layer used. It consists in multiple library modules written in C/C++ that implements interfaces for specific hardware, such as camera, radio or GPS that usually GNU Linux device drivers does not support consistently.

Another motivation for which HAL has been implemented in Android Stack, is that Kernel device drivers have General Public License (GPL), meaning that source code for drivers must be disseminated along with the binaries. OEMs of Android devices often use proprietary hardware and related software drivers that they will not share. Adding drivers in user-space HAL relieves them from releasing their source code

In the Automotive version it has been improved by enabling the system to interconnect to the vehicle network in order to access and act on vehicle data controlling physical components and sensors.

2.1.3 Android Runtime (ART)

Each application running on an Android system is an instance of the Android Runtime (ART, previously Dalvik), a runtime engine software that provides services for application execution.

It is in charge of running multiple virtual machines by executing DEX files (a bytecode Android-specific format). It grants:

- Ahead-of-time (AOT) and just-in-time (JIT) compilation;
- Optimized garbage collection (GC) with parallel execution;
- Better debugging support;
- Sampling profiler;
- Improved diagnostic detail in exceptions and crash reports;
- Threading and synchronization mechanisms.

2.1.4 Java API Framework

Many core Android system components and services, such as ART and HAL, are built from native code that require native libraries written in C and C++; a choice motivated by the will to not limit performances while guarantee developers productivity.

The Android platform provides Java framework APIs that exposes the functionality of some of these native libraries to apps and forms the building blocks needed to create Android applications. It includes:

- View system to build application's user interfaces
- A Resource Manager that provides access to resources like strings, images and layout files
- A Notification Manager that handles custom alerts and notifications
- An Activity Manager that regulates application's lifecycle
- Content Providers that enable applications to access share application's data with other software running on the same system.

- Services, application components that can perform long-running operations in the background without providing any user interface.
- Broadcast Receivers that can catch events of interest (custom broadcast messages or system notifications) and automatically act performing some particular tasks.
- Concurrency (handlers, messages, runnables, AsyncTask)

Android's Native Development Kit (NDK) allows the implementation of apps and services using native C/C++ code. Using the NDK, even on portions of code, can help enhance performance by minimizing latency, maximizing throughput and save system resources.

Some of the Android Native C/C++ libraries are:

- System C library
- Surface Manager
- Media framework
- FreeType
- WebKit
- OpenGL ES, SGL
- SQLite
- SSL

2.1.5 System applications

Applications are what the end-user will interact with. The stock version of Android comes with some built-in applications such as Dialer, Email and internet browser but the system supports the installation of user applications creating an easily extensible environment.

2.2 Android key-features: Why is expected to be adopted in Automotive industry?

In the following will be listed some of the key-features of Android Automotive that can bring benefits to the Infotainment market justifying the expectations about its adoption:

2.2.1 Open Source

Android is an open-source platform. Open-Source software enables rapid innovation, better security and cost effectiveness. Collaborating on non-competitive pieces of technology frees up resources, enabling companies to focus more on developing new products and services.

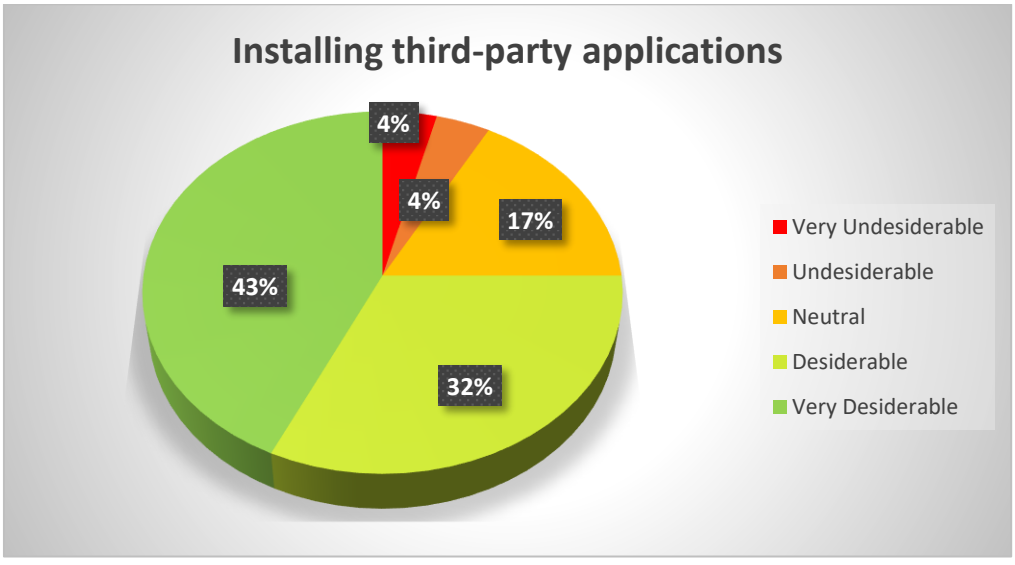
Moreover, an open source piece of software, means transparency; programmers can get full visibility into the code base while testing and sifting it to find bugs and eliminate them. Actually, the open source concept

powers about 90% of the internet and rapidly gets adopted across major enterprises.

2.2.2 Customizable and extensible features

Having an open source system enables automakers and third-party suppliers to customize the system to their own needs, not just esthetically but also functionally by modifying native applications and adding new ones.

Considering the functional aspect of Infotainments, users are familiar with smartphone-like behaviors, which have always been better, less frustrating and more satisfying than an IVI experience, not just in terms of usability but even for the multitude of different applications that modern mobile phones allow users to install and use, expanding mobile possibilities beyond simply making calls. A recent survey shows driver's desire to have external third-party applications installed into the vehicle infotainment system, in addition to OEM applications:



Graph 2. User's interest in having third-party applications in IVIs.

A system like Android Automotive lets drivers install and use their own applications just as a smartphone does, extending IVI possibilities.

2.2.3 Use of a standard and well-known environment

A standard and widespread platform such as Android gives end-users the feeling of a well-known and easy to use system they are familiar with, reducing time required to familiarize with it.

Having a standard system helps also application developers, who no more need to produce ad-hoc versions for different systems (even if different layouts of the same application could be needed, taking into consideration the various display screen possibilities in different cars).

2.2.4 Real multi-tasking environment

Actually Android Auto allows one only application window at a time, drastically limiting infotainment possibilities; for instance, the user can not monitor the vehicle status information as well as navigation to a destination at the same time.

This obliges the driver to constantly interact with the infotainment each time it needs a different application, increasing distraction possibilities.

Android Automotive, being an autonomous operating system, allows multi-tasking by enabling the possibility to run different applications at the same time, exploiting what a common Android OS does on mobile devices.

Multi-tasking, and in particular Android native software components like Broadcast Receivers and Content Providers, lets IVIs makers to produce applications that can reuse other running applications information by collecting and showing them. For instance, it is possible to have a "homepage" in the system that shows an overview about car status, media actually being played, navigation information and so on.

2.2.5 Built-in Vehicle APIs

One of the missing features in Android Auto was the possibility to retrieve data from car and act on the vehicle systems from the infotainment itself.

Commonly, many car subsystems interconnect with each other and the in-vehicle infotainment (IVI) system via various bus topologies. The exact bus type and protocols vary widely between manufacturers (and even between different vehicle models of the same brand); some examples include Controller Area Network (CAN) bus, Local Interconnect Network (LIN) bus, Media Oriented Systems Transport (MOST), as well as automotive-grade Ethernet and TCP/IP networks such as BroadR-Reach.

This vehicle HAL is the interface for developing Android Automotive implementations.

It is based on accessing (read, write, subscribe) “properties”, which are abstractions for specific hardware functions. System integrators can implement a vehicle HAL module by connecting function-specific platform HAL interfaces (e.g. HVAC) with technology-specific network interfaces (e.g. CAN bus). Typical implementations may include a dedicated Microcontroller Unit (MCU) running a proprietary real-time operating system (RTOS) for CAN bus access or similar, which may be connected via a serial link to the CPU running Android Automotive. Instead of a dedicated MCU, it may also be possible to implement the bus access as a virtualized CPU. It is up to each partner to choose the architecture suitable for the hardware as long as the implementation fulfills the interface requirements for the vehicle HAL.

2.2.6 Updatability and connectivity

Having a system as Android integrated into the vehicle opens to the possibility to have a “connected car” with an internet connection, whether it be through the smartphone or other way. More consumers are demanding for connectivity inside the vehicle and Android, being mobile oriented operating system, natively provides it supporting 3G/4G modules for internet access.

In this way, cars not only can offer entertainment and navigation assistance (such as real-time traffic or weather information), but they can also update their software or offer mechanics diagnosis, through IVI itself. This is something already introduced by newcomer automakers like Tesla, which has been able to extend the range of its cars through simple software updates

aimed to improve powertrain capabilities, vehicle dynamics and provide new onboard services.

This possibility is important as it keeps the system well updated and enables with newer features to provide longer life to the In-Vehicle Infotainment.

Chapter 3

Android automotive IVI:

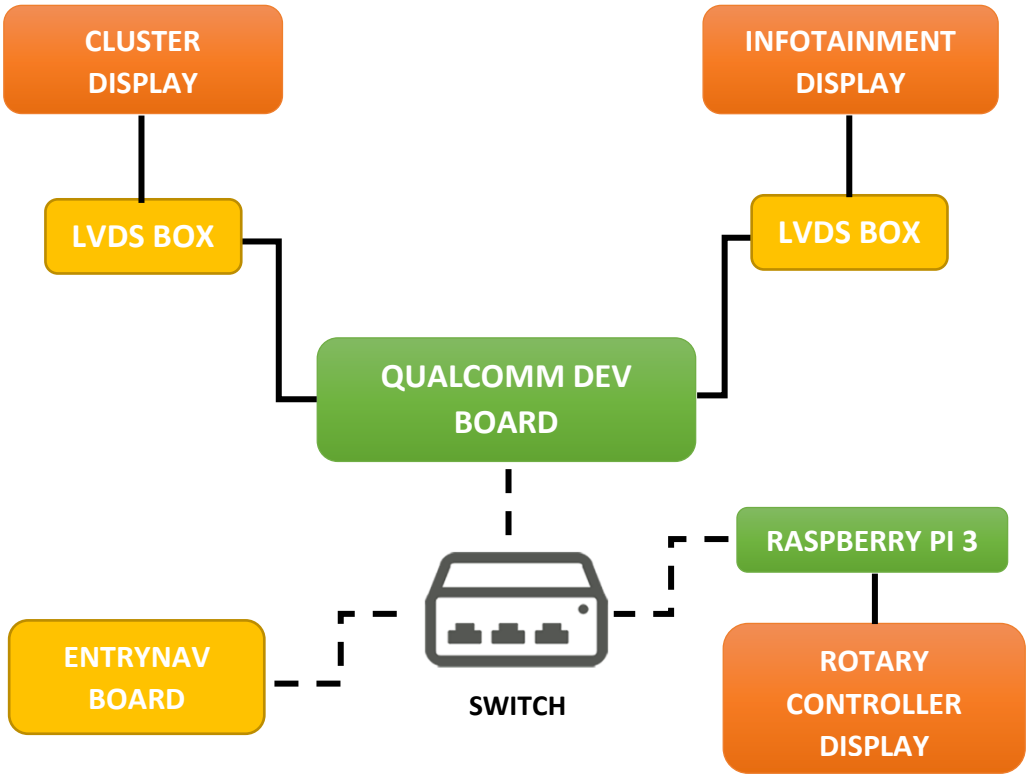
Hardware architecture design and implementation

3.1 Hardware architecture and components

As discussed in the introduction chapter, while developing an In-Vehicle Infotainment there are numerous requirements to be addressed such as usability and safety. Hardware architecture contributes to create a solid, reliable and safe to use system and for this reason must be carefully designed.

In this chapter will be explained the proposed hardware architecture, design choices and implementation of the Android Automotive In-Vehicle Infotainment developed in Magneti Marelli.

A scheme of the system's hardware architecture is shown below. For each of the meaningful elements a detailed description about the technology and design choice reason will be further provided along this chapter.



3.2 Chipset and processor

Given the set of requirements discussed in previous chapters, ranging from connectivity to information processing in a multitasking environment, the choice of a reliable and feature-full chipset is important.

The chipset chosen to develop the Infotainment system is a second-generation Snapdragon™ Automotive Development Platform (ADP) based on the Qualcomm® Snapdragon™ S820Am processor from Qualcomm® Technologies, Inc. (QTI).

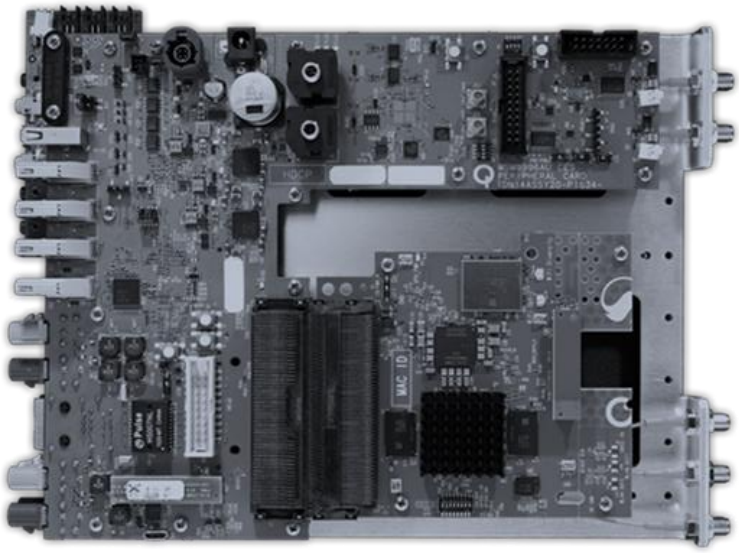


Figure 9. Qualcomm Snapdragon S820Am

The S820Am Snapdragon processor includes four Kryo™ CPUs (with clock speed up to 2.1 GHz), a Qualcomm® Adreno™ 530 GPU and Hexagon™ 680 DSP (for image processing and computer vision). It provides four GB LPDDR4 DRAM, 64GB eMMC 5.1 Flash Memory and supports expandable memory by using SD cards or USB storage devices.

Such memory is enough for partitioning it in two parts both suitable for the Android environment and QNX Neutrino O.S. with cluster logic.

Given that connectivity is one of the features required for this system, this ADP features rich connectivity. It provides X12 LTE modem (supporting 4G LTE up to 600 Mbps in download and 150 Mbps in upload), and 802.11a/b/g/n/ac WI-FI, Bluetooth 4.1 and GNSS RF receiver for time device location data using multi-satellite frequency bands (Glonass, BDS, Galileo) with Ethernet AVB and CAN support.

It even supports multiple camera sensors that can be useful not just for parking sensors or rear camera view, but also for applications in the field of computer vision and autonomous driving.

It provides a video output up to 4K resolution at 60 fps through four HDMI 2.0 connections supporting multiple touchscreen displays.

ADP brings an optimized application development environment for rapid deployment of high performance and power efficient connected automotive infotainment offerings. [18]

3.3 In-car displays

This infotainment architecture prescribes the use of three different screens in order to provide user with all the necessary information in a clever and comfortable way. Screens placement follows the standard approach to displace them around the driver, in a way that makes them easily reachable and visible.

Cluster display: placed in front of the driver, past the driving wheel. It is a 12.3" (1920x720 pixels) Amoled display without any interaction possibilities. It just delivers visual information about speed, torque and other car data to the driver, like any other cluster does with the only difference that our implementation does not impose any physical gauges, following actual trend to have a full digital cluster made in 3D computer graphics that exploits the GPU potential of the ADP.

Infotainment-display: placed in the center of the cockpit. It is the same 12.3" (1920x720 pixels) Amoled display of the info-cluster but coupled with a touch screen by Amtel. The Android interface resides on it and, along with the "Magic Rotary", it is the main interaction interface between the system and the user.

For both Info-Cluster and Head-Display, the choice of an Amoled screen is the trade-off between costs, high visual quality and energy efficiency. Amoled, that stands for *Active matrix organic light emitting diode*, is a display technology based on organic light-emitting diode that produces electroluminescence in response to an electric current and where each pixel has its own transistor and capacitor to actively maintain the pixel state (this is what creates so called active matrix) without requiring any screen backlighting.

An Amoled screen is more energy efficient than a common LED or LCD, has higher contrast and deeper blacks (because a black pixel corresponds to an off pixel) but is difficult to be viewed under direct sunlight unless the substrates are closer each other, requiring higher production costs.

The vertical positioning of Cluster and Infotainment displays makes sunlight harder to reach directly the screens. This is not true for *Rotary* screen that instead is placed horizontally near the gear shift, because it is thought to be used while driving and needs to be easily accessible to interact with the In-Vehicle Infotainment.

For this reason, it has been integrated a 6.5" TFT (Thin-Film Transistor – variant of LCD) display which offers good visibility even under direct light. It supports touch and force-touch interactions and is equipped with a rotating wheel in contact with the display used to scroll through menus and, by means of rotary pushes, to select menus items. It provides also haptic feedback in order to let the user know that the system has recognized required action.

3.4 Raspberry PI and Rotary controller

The software that enables Rotary controller logic runs on a Raspberry PI. It is a single-board computer developed to host Linux Kernel or other RISC OS and provides a Broadcom based SoC that incorporates an ARM processor, a VideoCore IV GPU and up to 1 GB of RAM memory. It does not use any Hard Drive, but a SD card for boot and non-volatile memory.

In particular, for this IVI implementation, a Raspberry PI 3 Model B has been used to run a QML software for magic rotary functionality in the Raspbian OS environment.

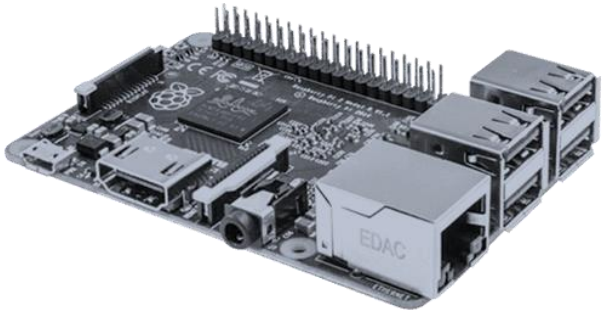


Figure 10. Raspberry PI 3 Model B

Its video output is directly connected to the Rotary screen through an HDMI connection while touch control hardware (both capacitive and force-sensitive) communicates with raspberry through common USB connections.

The rotary controller is essentially a 7 inches display (800x480 resolution) with a physical knob bonded over it. It furnishes haptic feedback generated through a solenoid on X axis and can be perceived during:

- Display touch;
- Rotary knob operations;
- Force touch interaction (over three bottom buttons).

3.5 EntryNAV system as gateway for CAN/V-MOST Bus

In modern vehicles, most of the electronic components and systems communicates each other or with Control Units (there can be more than 30 CUs in a modern car) through an enormous amount of wires and busses. One of the most important vehicle network is the CAN (Controller Area Network) introduced by BMW in 1986 that allowed to reduced vehicle wiring by almost 2 Kilometers and a communication speed up to 1 Mbps [19].

Another important vehicle bus is the MOST (Media Oriented Systems Transport), a high-speed multimedia network technology based on daisy-chain or ring topology to transport audio, video, voice and data signals via plastic optical fiber or electrical conductor.

The EntryNAV system is the original IVI placed in test car that directly communicates with ECU and other in car systems through the vehicle CAN (Controller Area Network) bus and MOST to retrieve car data such as speed, torque, temperatures, gears and so on. In our implementation, it is used as gateway for CAN/V-MOST to our Android Infotainment Ethernet network to provide information that are consequently forwarded to the Info-Cluster or shown in Android *"MyCar"* application, a custom Android application that will be described in the *chapter 5*.

The choice to use the EntryNAV for car data is a way to simplify the information retrieval without affecting too much the original test car architecture. In future development, car data retrieval will be done independently from the EntryNAV, by implementing a vehicle HAL to let the In-Car Systems communicate directly with the Android OS.

3.6 Peripherals and other devices

The communication between each part of the implemented system is Ethernet based and each data flow passes through a LAN Switch placed in between of Qualcomm devKit, Rasperry PI and EntryNAV.

An USB Hub allows the Head-Display touch panel to interface with the Android Qualcomm board and exposes USB port for peripherals connection (such as USB flash drives) in order to reproduce user's media.

Other in-car components used for the implementation are:

- **RADAMES:** HW Audio/tuner development board by Magneti Marelli;

- **Original car speakers**: speakers from test car connected to the original EntryNav system. The audio output from the ADP is convoyed through an AUX connection to the EntryNav, as if it would be an external audio source for the original EntryNav. In-Car audio is not just used for media reproduction but even for navigation turn-by-turn voice instructions and audio feedbacks during system use. This, in combination with microphone use, avoids user distraction during drive-by use.

- **Original car Microphone:** for user phone calls and Infotainment vocal instructions.

- **LVDS boxes with FDP-Link:** low-voltage differential signaling with the FDP-Link (Fiat Panel Display Link) standard is an interface for high-speed digital video transmission from GPU to the display

that supports a wide range of screen formats, refresh rates and pixel depths. They are used in the automotive infotainment industry to provide a digital plug and play interface that minimizes number of required wires and electromagnetic emissions to connect a video source to the display device. Moreover, they eliminate any kind of image fidelity loss that can result from the conversion into analog form of the signal from the source to the destination (screen panel).

Chapter 4

Android automotive IVI:

An overview about system requirements, proposed architecture and development process

Chapter overview

This chapter will depict requirements that the system must provide and will focus on the description of the proposed high-level architecture, design patterns and choices in order to develop it.

In the following, a high view of the development process will be provided to clarify the tools and techniques that have been used throughout implementation phase to fulfill expected requirements in terms of features, usability, reliability and safety.

It is important to notice that the system is a prototype and some features are still in development, while many others will be implemented later on (as described in the last chapter).

4.1 Current stage requirements definition

According to prototype requirements, at current stage, the Infotainment system was expected to at least:

- Provide an *Overview* application that collects infotainment data into a single layout, including Navigator data;
- Furnish a Multimedia player which is able to reproduce data from:
    - USB,
    - Bluetooth Streaming (through a connected mobile device),
    - WebRadio player (through internet connection via Wi-Fi);
- Implement a Radio player with AM/FM tuner and automatic/manual seek;

- Provide connectivity features for Hands-Free calls;
- Implement a navigation application;
- Furnish a *MyCar* application which collects vehicle data from CAN and allow the user to control in-vehicle systems and sensors;
- Enable the possibility to customize System UI colors combination and stream them to the Cluster and Rotary components;
- Integrate a Rotary controller with Haptic Feedback, force touch and a physical knob;
- Integrate a 3D full digital Cluster showing real vehicle data;
- Provide Hypervisor functionality in order to protect the Cluster from Android OS possible crashes (increasing Cluster reliability).

In order to implement these functionalities, a choice between two possible approaches was necessary:

- Implement a monolithic HMI application on top of the Android OS that provides required functionalities.
- Customize and extend the Native Android Automotive OS implementing new functionalities through the development of multiple applications.

By analyzing both solutions advantages and drawbacks, the choice fell on extending the Native Android Automotive OS. In the following the two approaches are explained and examined in order to justify the choice:

Monolithic HMI application for Android Automotive OS:

Figure 11. Android HMI Monolithic architecture

This solution is tailored for legacy projects reuse; an example can be the migration to Android of old non-Android systems by implementing their logics inside a single application. This approach enables the possibility to use different development technologies (such as HTML5 or QML) but requires higher effort while integrating 3rd party applications because of HMI coherence issues.

Native Android Automotive OS customization:

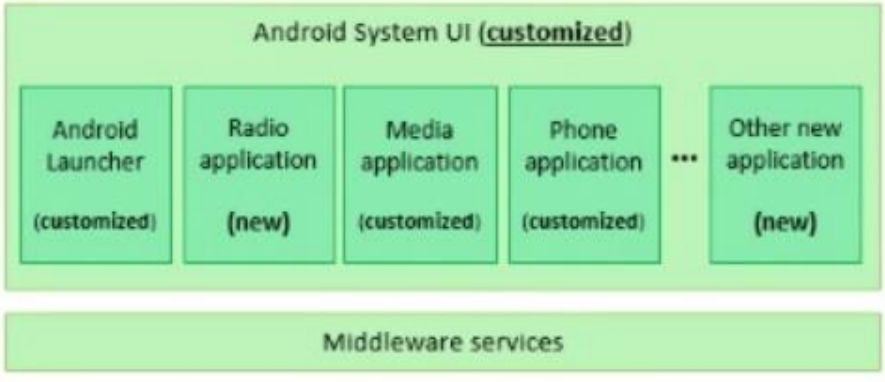


Figure 12. Native Android Automotive OS customization

Instead, this solution is Android Automotive compliant and provides a seamless 3rd party applications HMI integration (exploiting Android extendibility features). Moreover, each application runs in its own "sand-boxed" environment increasing system safety and is a cost-effective solution due to availability of Android built-in standard service and applications (some of required functionalities were already present in the native Android Automotive OS and required just customization). Finally, from the customer point of sight, provides a better end-user learning curve because system usage is based on mobile experience.

4.2 Development process

For the development of the infotainment system, the Android Studio SDK has been exploited, including a completely emulated android environment provided by the multi-device android emulator. By this approach, customizations and developed functionalities have been previewed first on PC, then on the Target (the Qualcomm Development Board).

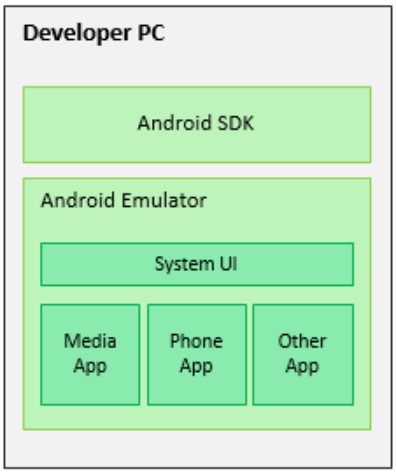


Figure 13. Development environment

An iterative development approach has been followed. It breaks down the software development of large applications in smaller chunks and iterates over design, implementation and test phase. At each iteration, design, development and test of additional features is possible until reaching a fully functional software ready to be deployed to customers. It is a key practice in Agile development methodologies.

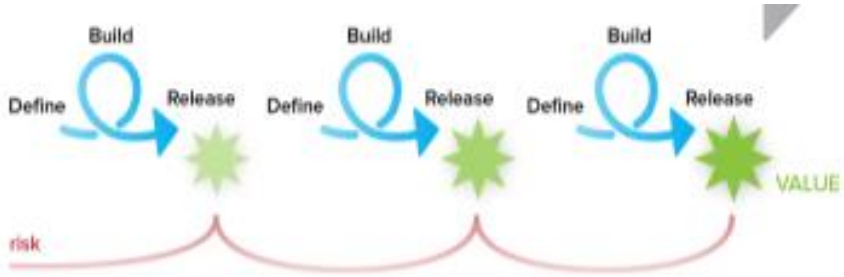


Figure 14. Iterative development model

Some of the advantages this approach provides are:

- Potential defects are spotted and dealt with early
- Functional prototypes are developed early in the project life cycle
- Less time spent on documenting and more on designing phase
- Progress is easily measured
- Changes are less costly and easier to implement
- Most risks can be identified during iteration
- Successive iterations can be managed easily as milestones
- An operational product is delivered with every iteration

In order to provide versioning, organization, ease of access and code sharing a common approach is to set up a shared server between each project

involved party. A versioning system based on *Git* has been used to manage the development mainline and work on features or proposals on different branches.

The parties involved in this project can be grouped in four categories:

- **UX designers**

  They define visual style and create mockups, animations, transitions and user interactions. Collaborate in optimizing graphical design to fit real implementation.

- **HMI designers**

  They are responsible for ergonomics and usability of the HMI. Define user interaction schemes and control/approve the look and feel of the product.

- **HMI Developers**

  The undergraduate belongs to this team. They implement visual states, graphics in chosen graphical engines, application code and business logics. They connect data to User Interface and validate design performance and applicability. Finally, they optimize system performance and execute tests.

- **Management and Validation**

  They monitors the whole co-design process and verify or approve compliance with respect to expected product features and requirements.

4.3 Proposed high level Infotainment architecture

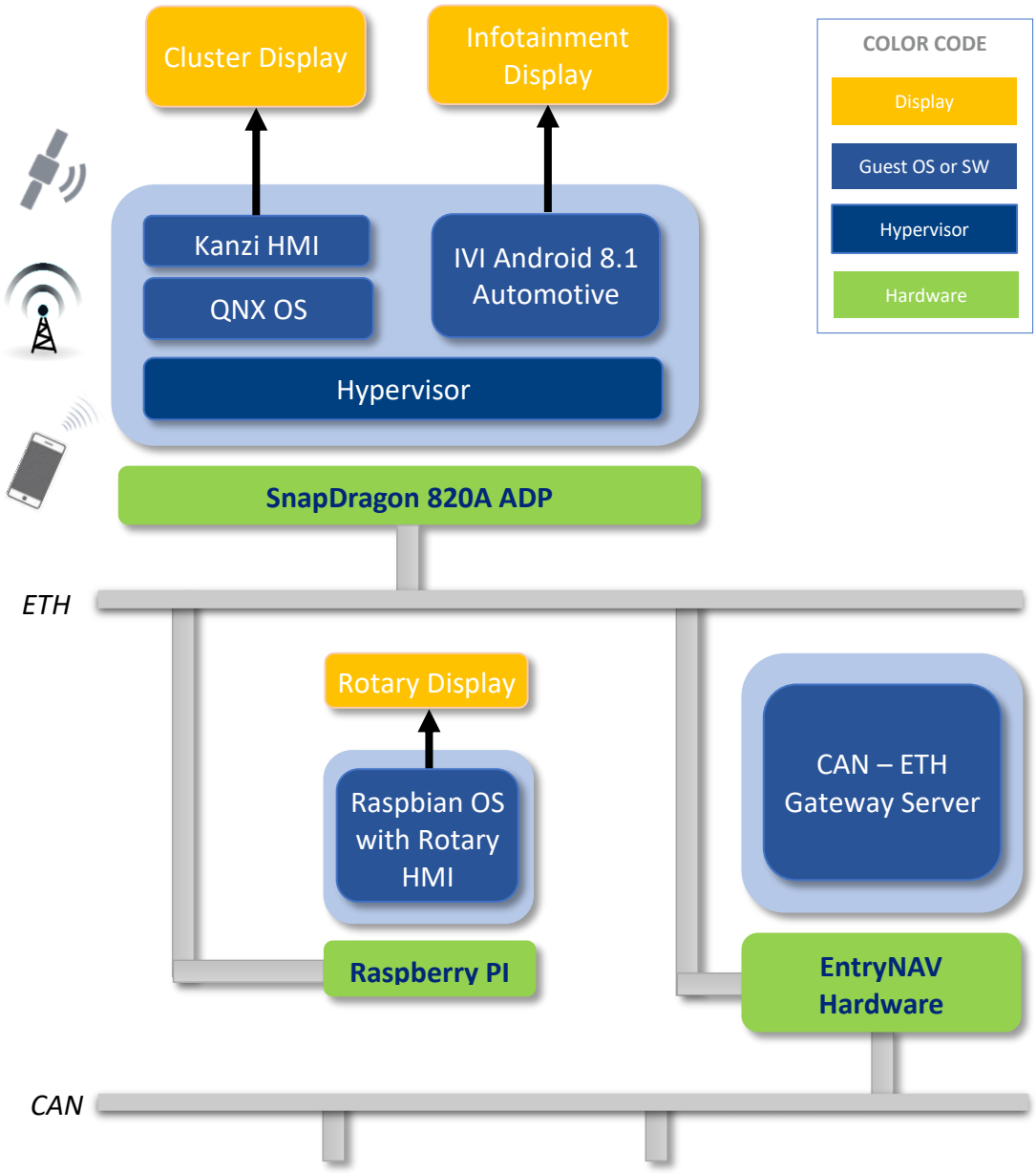


Figure 15. Infotainment High-Level architecture

The whole system is made of several software components that interact each other enabling all the required functionalities expected by this infotainment. A brief description of each component will be provided throughout this chapter.

The software architecture can be split in these main elements:

- Hypervisor running two guest OS:
    - QNX Neutrino OS for Kanzi HMI (Info-Cluster);
    - Android 8.1 Automotive OS core for Infotainment. It includes
        - Android Applications (native and custom ones);
        - WebSocket/Socket service;
- EntryNAV Gateway server.

4.3.1 QNX Hypervisor 2.0

In the first chapter has been outlined the need of a hypervisor to separate safety-critical components from non-safety critical ones in different guest operating systems. This is a common approach in automotive HMI development and, in our implementation, it enables separation of the Info-Cluster (which provides safety-critical data) from the Android Core but running on the same SoC, reducing costs by sharing resources.

To implement this logic, the *QNX® Hypervisor* 2.0 has been chosen. It is a Type 1 real-time priority-based microkernel built for managing virtual machines actually present in many safety-critical areas such as air traffic control systems, medical devices and nuclear power plants as long as in infotainment systems. Some of its main features are:

- Virtual CPU model (vCPU model): the hypervisor uses a portion of the physical CPU cycle and allocates it to a vCPU assigned to a VM. The vCPU should be considered as a share of the time in the processor's core.
- Share cores and resources among virtual machines based on priority
- 64-bit/32-bit guests: QNX Neutrino, Linux, Android, RTOS
- Shared memory
- Failure detection and restart of guests
- TCP/UDP networking between virtual machines to let them work cooperatively
- Virtual machines can render graphical output to shared or separate displays by sharing GPU and graphics.

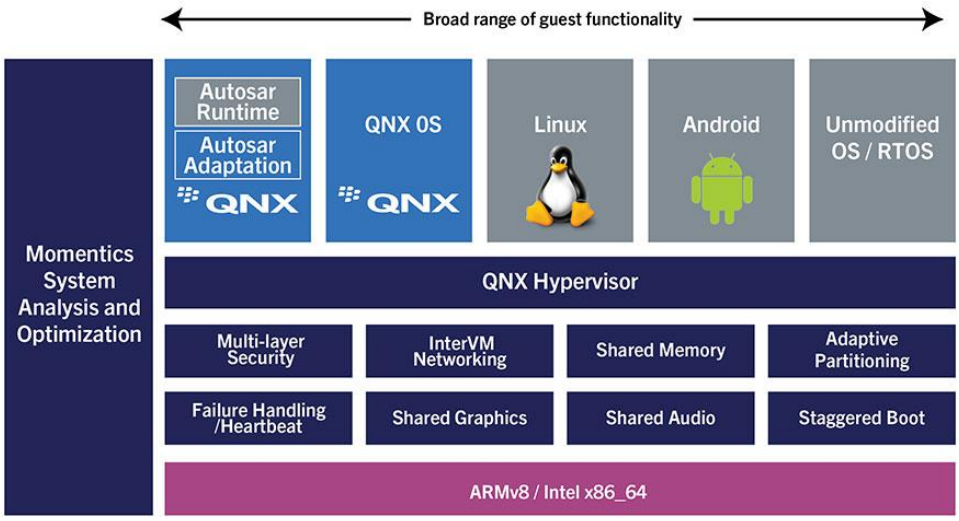


Figure 16. Hypervisor architecture model

Our infotainment implementation used two guest operating systems above the hypervisor: Android OS as core Infotainment system and QNX Neutrino OS to run the software for the Info-Cluster logic that reads messages from the Ethernet channel, interprets them and updates the cluster UI.

4.3.2 QNX Neutrino OS for Kanzi HMI (Info-Cluster)

QNX Neutrino is the operating system running as guest OS over the hypervisor, along with Android OS. It executes the Info-Cluster software written in C++ that powers a 3D user interface designed in Kanzi Studio, to provide data such as speed, RPM, external/internal temperatures, engine temperature, gears and fuel level.

Its adoption as OS for the Info-Cluster is justified by its fast boot times and the highly customizable environment it provides; in fact, the microkernel can be up and running even in 250 milliseconds. This is due to the large flexibility it offers because based on a modular architecture where each component is extremely independent from the others allowing developers to activate just the ones needed, making easy to rearrange the system startup sequence to suit specific design needs. This approach also allows the system to meet faster audio/video startup or accessing hardware or in vehicle networks (such as CAN) in smaller times. [20]

4.3.3 EntryNAV Gateway server

Android Automotive OS provides a built-in Hardware Abstraction Layer that, as already mentioned, permits OS to be agnostic about lower-level driver implementations and enables a software interface for communication with the hardware.

At current prototype status of our infotainment system, it does not interact with vehicle hardware, thus no HAL was implemented. Anyway, the system

is able to read vehicle data from the CAN bus through the original EntryNAV system of the demo car. In order to enable this logic, the EntryNAV has been modified implementing a server constantly sending out data coming from ECUs and other control systems, solely acting as a gateway for CAN bus.

Our android service application created to implement a communication protocol (previously adverted and later on explained in detail), is in charge of instantiating a client that connects to the EntryNAV server and retrieves data to be forwarded Info-Cluster side.

4.3.4 Android Automotive OS, Android Applications and WebSocket/Socket service

In our Infotainment implementation, the main core is Google Android Automotive OS. The implemented version is 8.1 Oreo and, because of partial support at current Android stage of cluster mirroring and rotary interactions, we decided to do not adopt a Full-Android infotainment software architecture and opted for a solution using two other different systems (as already presented, QNX Neutrino OS and Raspbian OS) for managing Info-Cluster and Rotary logics.

In order to let the Rotary control the infotainment and implement In-Vehicle data repetitions, which makes the system more usable and reduces driver distraction possibilities, a client-server messaging protocol has been implemented. An Android service application (in the following named *SocketService*) takes the role of server, collecting data from bound

applications and forwarding them through a Socket and a WebSocket. Its clients can be of three types:

- *Infotainment applications*: They furnish their data binding to the service and sending messages through the interface provided by the service itself, once the application is bound.
- *CarClient*: It is a client thread started by the service itself, that listens for data from the EntryNAV server (which provides vehicle data coming from CAN bus) and forwards it to the *SocketService*.

- *Info-Cluster* and *Rotary controller*: They connect respectively to the Socket and WebSocket on which data is emitted by the *SocketService*. In particular, through the WebSocket which exposes a HTTP-based communication, the Rotary controller can request data (by means of GET requests) and forwards control messages to the infotainment (by POST requests).

A further explanation about the *SocketService* messaging protocol is provided in the *Chapter 5 – Section 4.4* and in *Appendix A*.

An additional layer to the Infotainment software architecture is provided by its applications (described in the next chapter).

Android Automotive OS comes with some native applications such as the Radio player, Media player and Phone. For them business logics were provided by Android Automotive and have been partially inherited from it, extended or eventually customized. Other applications required completely new business logics implementation and user interface design in order to accomplish system feature requirements.

4.4 The Android software architecture

The main target of this section is to provide information about our custom Android infotainment architecture. This information will serve as basis for the detailed applications description held in *Chapter* 5.

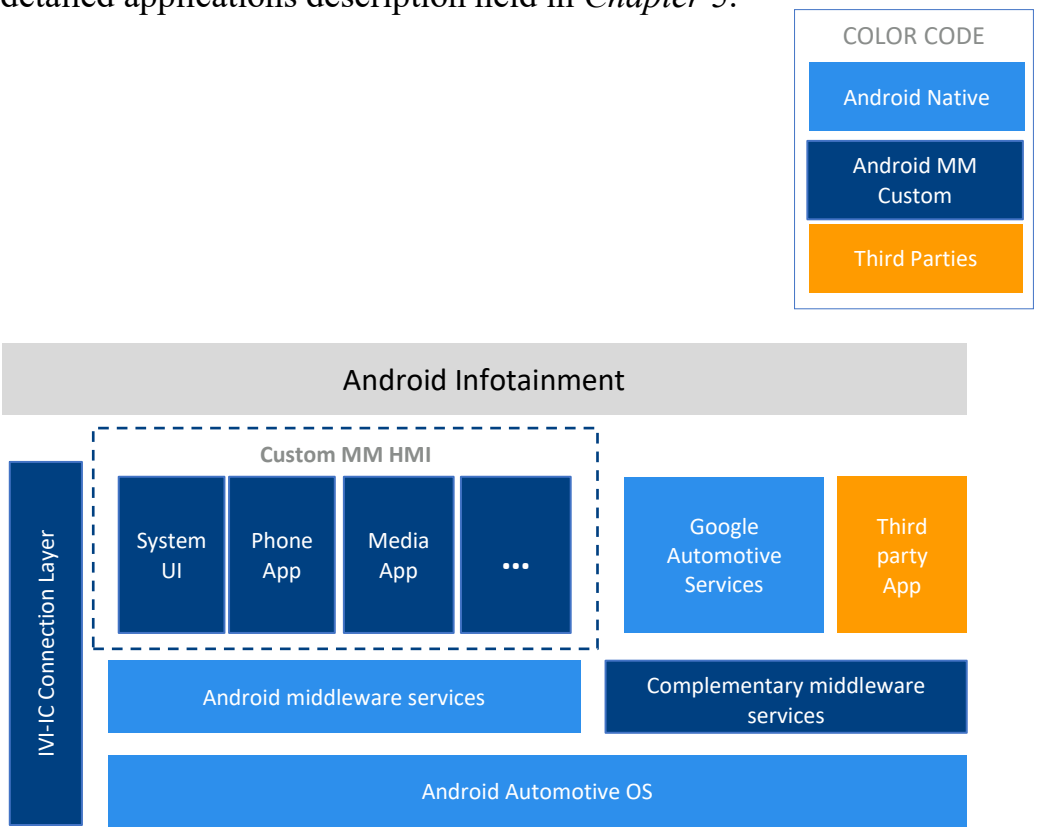


Figure 17.Android Infotainment software architecture

The Android Automotive OS is furnished with native middleware and automotive services. They are not real applications but packages of APIs very linked to the Android system itself. In Automotive OS, many services are available such as Google Play Services, Google Assistant and Google Maps but, for this implementation, complementary middleware services have also been developed (such as the *SocketService* messaging protocol).

Along with the customization and extension of built-in apps, many other applications have been implemented to bring required features.

One of the most important aspects in a system is the coherence among each rendered user interface. For an infotainment such this, that runs many different applications even from third party, it becomes a key aspect to be considered. Each user interface that the system renders has to be coherent with the others and, because one of the expected features is the possibility to customize system UI colors, a common substructure for each application is required.

With this objective, two techniques have been exploited making development easier by providing a common starting point for each application:

- Implementation of a custom user interface library providing classes and methods for managing UI and other common applications functionalities such as the Drawer menu.

- Definition of a common scheme in order to support UI theme customization of system colors among each application by exploiting the *data binding* and Android *viewModel* logic, implementing the Model-View-ViewModel architectural pattern.

4.5 Custom User Interface library: *mm_ui_lib*

Before moving on to applications, it is useful to explain for which purposes a custom library has been implemented. The library, called *mm_ui_library*, exposes some classes and methods useful in the development of almost any application implemented in the Infotainment system.

It furnish a custom activity called *BasicDrawerActivity*, an extension of the android native *AppCompatActivity,* which provides a drawer menu implementation and some methods to handle its animations, addition or deletion of menu entries (elements rendered as independent fragments added through a drawer menu adapter) and rotary control commands.

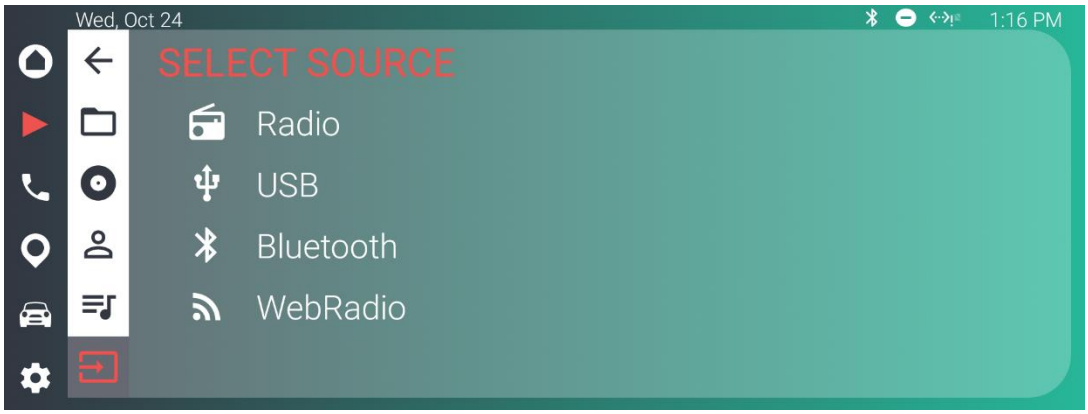


Figure 18. Drawer usage example in the Media application to enable source selection

In fact, with the aim to make the system usage faster and reduce driver distraction possibilities, the Rotary controller allows to navigate through menus items. The exchange of messages between the Android OS and Rotary (in both directions) goes through the SocketService application (in next chapter explained in detail) that implements a service managing a

messaging protocol for infotainment components communication (Cluster and Rotary).

The *BasicDrawerActivity* provides methods for handling rotary interaction messages by binding to the service. Its *onCreate()* callback executes a *bindService()* function that binds the current application (the one extending BasicDrawerActivity) to the service.

A service connection interface provides two callbacks in order to set a Messenger once the connection is established:

```
private ServiceConnection mConnection =
   newServiceConnection(){

      public void onServiceConnected
      (ComponentName className, IBinder service){
         mService = new Messenger(service);
         mBound = true;
         notifyConnection();
      }

      public void onServiceDisconnected (ComponentName
      className){
         mService = null;
         mBound = false;
      }
};
```

The Messenger, by using a messages handler, interprets rotary commands and eventually acts on the implemented drawer menu to open, close or select items by calling drawer methods such as *openDrawer()* and *closeDrawer()*.

In addition, the Rotary controller must be aware of the current drawer state when it gets opened or closed, by interacting with the main display clicking on the drawer menu icon instead of using the Rotary. This explains the call to the method *sendDrawerStateToService()* when drawer open or close gets invoked*:*

```
protected void openDrawer()
{
    if (drawerOpened) { return; }
    if (drawerStateChangeListener != null) {
        drawerStateChangeListener.OnChangeState(true);
    }
    drawerOpened = true;
    sendDrawerStateToService(drawerOpened);

    binding.setDrawerOpened(true);
    applyDrawerAnimation(0);
}



protected void closeDrawer()
{
    if (!drawerOpened) { return; }
    if (drawerStateChangeListener != null) {
        drawerStateChangeListener.OnChangeState(false);
    }
    drawerOpened = false;
    sendDrawerStateToService(drawerOpened);

    binding.setDrawerOpened(false);
    applyDrawerAnimation(-elemSizeParams.width);
    setBasicLayoutVisibility(View.VISIBLE);
    currentSelectedItem = null;
}
```

The library also provides classes (such as the *ColorsHelper* interface) and methods to interact with a content provider for system UI colors in order to

inform each application of the colors combination chosen by the user (settable through the *Preferences* application, further explained in details) making the system appearance consistent in any application the user interacts with.

Apart from various utility classes, it offers an additional extension of the *BasicDrawerActivity* class called *BasicDrawerMediaActivity*. This activity adds methods for loading into the drawer menu, items corresponding to the four audio source possibilities, setting for each of them the related intent to be called in order to start the right application (among USB, Bluetooth, Radio or WebRadio).

4.6 Model-View-ViewModel (MVVM) architectural pattern for system UI colors customization

As aforementioned, one of the features that the infotainment system provides is the possibility to customize system UI colors through the *Preferences* application.

From the coding perspective, a fast and reliable logic to let each application's UI elements update with new colors combination had to be implemented. In order to accomplish these needs, two Android native support libraries have been exploited:

- The **Data Binding Library**: Allows developers to bind UI components in layouts to data sources in the application by using an XML declarative format rather than programmatically. Binding components in the layout file makes possible to remove many UI

framework calls in activities, making them simpler and easier to maintain. This approach also improves applications performance and helps prevent memory leaks and null pointer exceptions. [22]

- **ViewModel Library**: this class is designed to store and manage UI-related data in a lifecycle conscious way. It allows declaring "observable" objects capable of notifying *observers* about changes in their data.

With this solution, it has been possible to implement the Android architecture pattern named *MVVM – Model-View-ViewModel*. In this scheme, the ViewModel component exposes a stream of states to which the View can bind to, in order to get notified when changes in ViewModel's data happen. This means that the View keeps references of the ViewModel but not vice versa.

This mechanism of synchronization between the ViewModel and View is kept alive by the data binding through a declarative syntax in the View itself. This implies that modifications to bound data in the ViewModel are automatically reflected into the Views without particular burden from the developers. Doing so, we ensure that the View always displays current state of data in the ViewModel.

By default, a binding class is generated based on the name of the *View* layout file, named in *CamelCase*, removing underscores and suffixing "Binding". An object of this class gets instantiated when *DataBindingUtil.inflate()* method is called to tell the targeted activity to bind to a given layout. In order to enable the automatic generation of the binding class at compile time, the XML android layout file must have a specific format:

- It must enclose in the <*layout*> tag both layout elements and a <*data*> tag containing one or more <*variable*> children.
- Each <*variable*> tag defines an object to be bound to the layout by specifying a variable name and its class path. The attributes from the bound object are accessed by using the syntax *@{VariableName.attribute}*.

In our implementation, each layout file that needs to adapt to system colors has the following format:

```
<layout

xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <data>

     <variable
       name="colorsViewModel"
       type="com.magnetimarelli.mm_car_ui_lib.
             theme.ColorsViewModel"/>

  </data>

       <!-- Layout elements here. They will use the
            @{colorsViewModel.atributeName} syntax
            in order to reference data in the viewModel.
            For instance, a button will have a background
            color defined as:
            android:backgroundTint =
                        @{colorsViewModel.accentColor};
       -->

</layout>
```

where *ColorsViewModel* is the entire ViewModel binded to the *View.* It exposes four Observable attributes:

- *backgroundGradientColors*: is an array of two elements representing start and end color for the background;
- *accentColor*: for highlighting buttons, icons or texts;
- *primaryColor*: mainly used to color large areas;
- *textColor*: standard text color (usually white).

Once specified data to which the *View* must be bound, it has been necessary to implement a mechanism to keep updated colors in the *ViewModel*, represented as the *ColorsViewModel* class, with user settings.

Our choice has been to let the *ViewModel* observe changes into a *ColorsHelperImpl* (deeply depicted in the section about *Preference*s application and theme logic description) class that handles interactions with the implemented Colors content provider, a unique system repository for the chosen color combination from which any application can read actual theme data, by referencing upcalled helper. The *ColorsViewModel* class attaches the following observer to an instance of the *ColorsHelperImpl* class (that implements the Observable interface):

```
private final Observer colorsObserver = new Observer() {
    public void update(Observable observable, Object arg)
    {
        ColorsViewModel.this.loadColors();
    }
};
```

Whenever a change is notified from the *ColorsHelperImpl,* this observer calls the *loadColors()* method in the ViewModel that queries (through the helper) the content provider in order to update ViewModel colors attributes.

The *ColorsHelperImpl* registers the following *ContentObserver* that notifies when changes in the provider data happen (after user has set a new theme):

```
private final ContentObserver contentObserver = new
ContentObserver(new Handler()) {
    public void onChange(boolean selfChange) {
              super.onChange(selfChange);
              ColorsHelperImpl.this.colors =
                     ColorsHelperImpl.this.queryColors();
              ColorsHelperImpl.this.setChanged();
              ColorsHelperImpl.this.notifyObservers();
    }
};
```

Any of the applications that will be described in following sections will implement the MVVM pattern by getting an instance of the *ColorsViewModel* from the ViewModelProvider and binding its View to the ViewModel by executing the following lines in the *MainActivity* creation:

```
ColorsViewModel theme = ViewModelProviders.of(this)
                          .get(ColorsViewModel.class);
MainActivityBinding  binding =
DataBindingUtil.inflate(inflater,
R.layout.main_activity_layout, container, false);
binding.setColorsViewModel(theme);
```

The framework automatically provides, in the generated binding class, *getters* and *setters* for the binding variables declared in the XML layout. As we bind the whole *ViewModel*, a *setColorsViewModel()* method is available in the binding class, accepting a *ColorsViewModel* object.

Chapter 5

Android automotive IVI:

Infotainment applications design and implementation

Chapter overview

This chapter will depict design and implementation of Android software applications that enable all the functionalities offered by the Infotainment system describing also user interfaces design to develop an easy and fast to use system contributing to meet safety requirements. To understand better the logic behind the applications, portions of significant code will be introduced and explained.

In addition, this part will cover two services that were needed to implement required functionalities:

- The logic behind the *Stream* service, that allows other software components to provide their data to the *Overview* application in charge of summarizing infotainment data to support driver with useful information while minimizing distraction possibilities.

- The service that enables communication between Android system and infotainment "appendices" (Info-Cluster and Rotary controller) not just to provide In-Vehicle Infotainment repetitions guaranteeing data consistency among these systems and reduces driver distraction possibilities, but even for transmitting HMI control commands from the Rotary to the Android Infotainment and vice versa.

5.1 *Overview* application

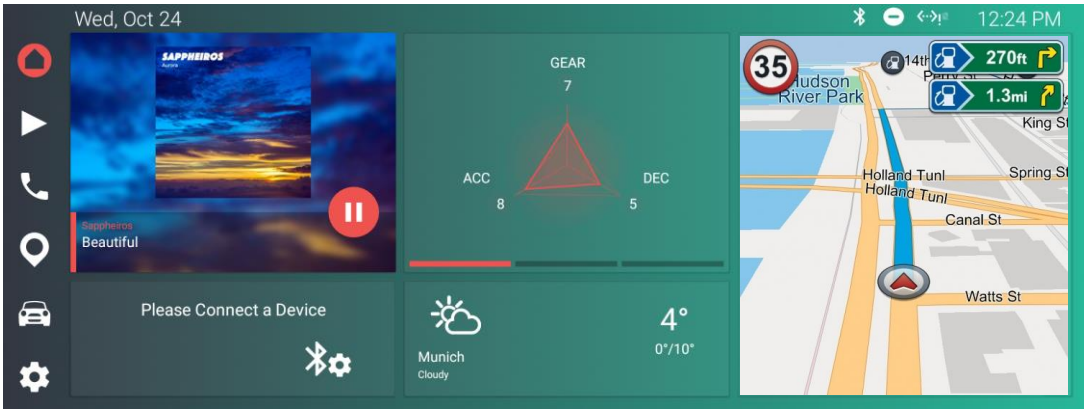


Figure 19. Overview application layout

The *Overview* application is thought to bring a summary of all the possible information that the infotainment can provide to the user in a clean and clever layout. It has the target to be as much condensed as possible, making easy for the driver to get enough information from a single layout without necessarily requiring interactions with the Infotainment. With this approach, we accomplish cognitive distraction avoidance.

It exploits *CardViews*, design elements introduced with Google material design guidelines, to show five different blocks of information. The absence of complex menus is obtained by enabling the possibility to click on each of the cardViews, taking the user to the related application with just one interaction; in addition, some cards support more than just simple click interactions making possible to operate on the application itself without opening it.

Two of the cardViews are created by the service application Stream (further explained), which collects data coming from the following applications:

- Media/Radio/WebRadio: providing data about current media track or radio station. It displays a cover image (or a default placeholder) and allows to play/pause the current audio source that is being played;
- Dialer: phone and eventually data about on-going or recent calls;

At Overview startup, the *bindStreamService()* method operates the binding to the Stream service by setting the appropriate component intent and executing the following line of code

```
bindService(intent, mConnection, BIND_AUTO_CREATE);
```

passing a mConnection object that is an instance of *StreamServiceConnection* in order to register two callbacks:

- *onServiceConnected()*: retrieves an *mService* interface exposed by the *onBind()* method of the Stream service.
- *onServiceDisconnected()*: retries the connection to the service in case it is lost.

The *mService* object provides methods to access the *StreamCards* generated by the Stream service and, once connected, Overview starts fetching available cards loading corresponding data into related *cardViews*.

Vehicle data cardView is filled with car information (this data in future implementation will be taken from the CAN bus, but actually is just simulated). It exploits a *tabLayout* with *ViewPager*, a native Android layout element that supports switching horizontally through multiple pages (usually rendered as independent fragments). The weather *cardView* takes data from an internet service (in presence of an internet connection).

Phone cardView offers a recycleView dinamically loaded with recent calls contacts taken from a connected mobile phone (via bluetooth). In particular, this card gets its data from the Stream service and is enabled just when a phone is actually connected showing a profile picture related to phone user and the device name (bluetooth name). If no phone is connected, a placeholder text is shown and, after a cardView tap, the *ConnectivityFragment* from the *Preferences* application is opened (refer to section *Preferences - ConnectivityFragment*).

The right most card contains just a map overlay to show current navigation data if any, otherwise a placeholder is presented to do not add useless overhead to actual system memory usage by starting an application that is not actually necessary for the driver.

As any other infotainment application, it exploits the MVVM architectural pattern to be always consistent with system colors.

5.2 *Preferences* application

Preferences application exposes some settings to personalize user interface aspect in terms of colors and to manage system connectivity and volumes. From the user interface point of view, it is made of a *tabLayout* composed of three different tabs rendered as independent fragments:

- *Themes Fragment*: enables the possibility to customize system colors;
- *Connectivity Fragment*: exposes settings for Bluetooth and WiFi connectivity;
- *Volumes Fragment*: provides sliders for system volumes settings.

5.2.1 *Themes* fragment

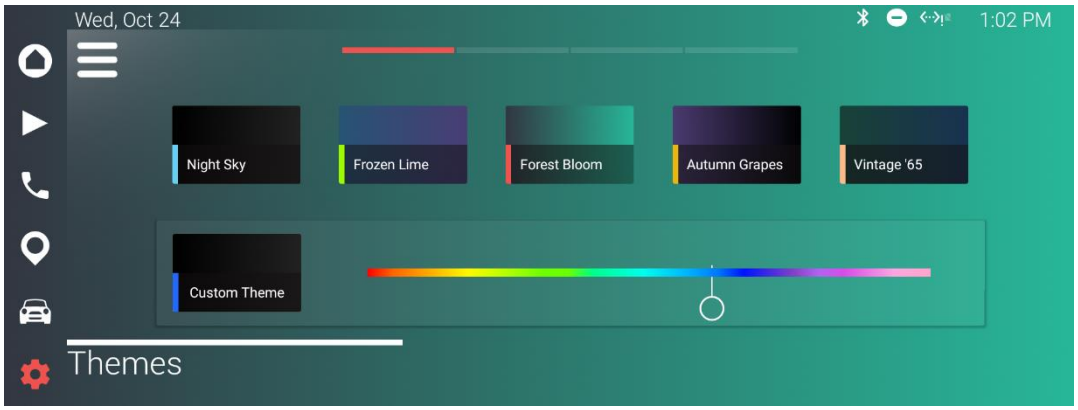


Figure 20. Preferences – Themes Fragment layout

The themes fragment renders five elements representing standard themes plus a customizable one where background gradient is on a black scale and

accent color is settable by using a seekBar, a native Android sliding bar design element working as color selector.

Enabling a personalization of colors is not just a matter of tastes but it can make the system usage more comfortable: in nighttime darker backgrounds are better while, under sunlight, brighter colors are preferable.

Of particular interest is the logic behind system colors setting and how this information gets forwarded to any other application. In fact, the chosen theme must be known to each application in order to set the right colors combination when rendering their layout. In order to have a common element from whom access to data a content provider has been implemented. Access to its data is provided through the methods of *ColorsHelperImpl* class, implementing an interface defined in the custom *mm_ui_lib* library.

A content provider manages access to a central repository of data. A provider is part of an Android application, which often exposes its data (stored in some manner that can range from a Json file to persistent data storage as databases or shared preferences) to other applications by using a provider client object. Together, providers and provider clients offer a consistent, standard interface to data that also handles inter-process communication and secure data access. [21]

The preferences application implements its own content provider and, by the use of up called custom library, any other application can use a provider client to access data it exposes.

The content provider stores data in one or more tables similar to the ones found in a relational database.

Preferences application implements its own content provider by the *ColorsProvider* class. The data inside this content provider is identified by

a content URI, which is the union of provider symbolic name (called authority, unique android-internal identifier) and the path to the table or file. An optional id part points to an individual row in a table. Every method to access content provider data requires a URI.

The authority for the *ColorsProvider* is specified in the library itself through the *ColorContract* class which contains, along with other attributes, these two lines:

```
public static final String AUTHORITY =
              "it.zirak.automotive.colorpicker.provider";

public static final Uri AUTHORITY_URI = Uri.parse(
                           "content://" + AUTHORITY);
```

The content provider exploits the *UriMatcher* convenience class to map URI paths to integer values and evaluate the corresponding action to be taken when it receives a query request (retrieve data from the provider returning a Cursor object):

```
@Nullable
@Override
public Cursor query(@NonNull Uri, @Nullable String[]
                     projection, @Nullable String
                     selection, @Nullable String[]
                     selectionArgs, @Nullable String
                     sortOrder)
{

    MatrixCursor returnValue;

    switch (uriMatcher.match(uri))
    {
       case MATCHED_COLOR:
          returnValue = new MatrixCursor(
                              new String[]{
```

```
                ColorsContract.Color.COLUMN_KEY,
                 ColorsContract.Color.COLUMN_VALUE
            });

            addRowIfValueExists(returnValue,
                  ColorsContract.Color.KEY_ACCENT);
            addRowIfValueExists(returnValue,
                   ColorsContract.Color.
                     KEY_BACKGROUND_GRADIENT_BEGIN);
            addRowIfValueExists(returnValue,
                   ColorsContract.Color.
                     KEY_BACKGROUND_GRADIENT_END);
            addRowIfValueExists(returnValue,
                  ColorsContract.Color.KEY_PRIMARY);
            addRowIfValueExists(returnValue,
                  ColorsContract.Color.KEY_TEXT);

            break;

        default:
            throw new IllegalArgumentException(
                          "Unsupported uri " + uri);

    }

    return returnValue;

}
```

The query method returns a MatrixCursor loaded with provider's stored colors taken from shared preferences by calling the method *addRowIfValueExists()*, passing the MatrixCursor reference and a color key.

Another important method overridden in our *ColorsProvider* implementation is the *insert()* which provides the insertion of colors values

for accent, text and backgrounds gradient colors in our provider shared preferences:

```
@Nullable
@Override
public Uri insert(@NonNull Uri uri, @Nullable
ContentValues values) {
    if (values == null) {
        return null;
    }

    switch (uriMatcher.match(uri)) {
        case MATCHED_COLOR:

            Context context = getContext();
            assert context != null;
            SharedPreferences.Editor editor =
              sharedPreferences.edit();

            String key =
                values.getAsString(ColorsContract.
                                    Color.COLUMN_KEY);
            int value =
                values.getAsInteger(ColorsContract.
                                      Color.COLUMN_VALUE);
            editor.putInt(key, value);
            editor.apply();
            context.getContentResolver()
                          .notifyChange(uri, null);
            broadcastAllColors();

            break;

        default:
            throw new IllegalArgumentException(
                            "Unsupported uri " + uri);
    }

    return null;

}
```

This method instantiates a shared preferences editor that allows putting or modifying values. A call to this method always corresponds to a new setting of colors for the UI; this is the reason why the method *broadcastAllColors()* is called. It essentially broadcasts new theme colors to infotainment applications and to the SocketService application used to forward data to Cluster and Rotary controller (this service application will be further explained in details).

```
private void broadcastAllColors() {

    Context context = getContext();
    Bundle colorBundle = new Bundle();
    assert context != null;
    Intent intent = new Intent();
    intent.setAction("it.zirak.automotive.colorpicker");

    addExtraIfValueExists(intent,
      ColorsContract.Color.KEY_ACCENT);

    addExtraIfValueExists(intent,
      ColorsContract.Color.KEY_BACKGROUND_GRADIENT_BEGIN);

    addExtraIfValueExists(intent,
       ColorsContract.Color.KEY_BACKGROUND_GRADIENT_END);

    addExtraIfValueExists(intent,
       ColorsContract.Color.KEY_PRIMARY);

    addExtraIfValueExists(intent,
       ColorsContract.Color.KEY_TEXT);

    context.sendBroadcast(intent);

    int secondary, primary, bg_begin, bg_end;

    secondary = intent.getIntExtra(
                ColorsContract.Color.KEY_TEXT,0);

    primary = intent.getIntExtra(
              ColorsContract.Color.KEY_ACCENT,0);
```

```
    bg_begin = intent.getIntExtra(
               ColorsContract.Color.
               KEY_BACKGROUND_GRADIENT_BEGIN,0);

    bg_end = intent.getIntExtra(
               ColorsContract.Color.
               KEY_BACKGROUND_GRADIENT_END,0);

    colorBundle.putString(MessagesHelper.COLOR_PRIMARY,
       String.format("#FF%06X", (0xFFFFFF & primary)));

    colorBundle.putString(MessagesHelper.COLOR_SECONDARY,
       String.format("#FF%06X", (0xFFFFFF & secondary)));

    colorBundle.putString(MessagesHelper.COLOR_BG_BOTTOM,
       String.format("#FF%06X", (0xFFFFFF & bg_end)));

    colorBundle.putString(MessagesHelper.COLOR_BG_TOP,
       String.format("#FF%06X", (0xFFFFFF & bg_begin)));

    sendBundleToService(colorBundle);

}
```

It is possible to notice that an intent is created by adding some *Extras* values and sent in broadcast to the whole Android system. It actually notifies the system UI (and any other application that registered an *Intent Filter* to catch it) to change its colors adapting to the user selected theme.

In the *onCreate()* method of ThemesFragment class a *colorsHelper* object of the ui_library class *ColorsHelperImpl* gets instantiated.

This class is in charge of:

- querying for colors the *ColorsProvider;*

- registering a Content Observer in order to detect changes in the provider content (new settings of colors) and notification to eventual observers;
- requesting for new colors insertion in provider database.

Current color combination is retrieved by the method *queryColors()* which involves content provider query() method previously described:

```
@NonNull
private Colors queryColors() {

    int accentColor = Colors.DEFAULT_ACCENT_COLOR;
    int backgroundGradientBeginColor =
       Colors.DEFAULT_BACKGROUND_GRADIENT_BEGIN_COLOR;
    int backgroundGradientEndColor =
       Colors.DEFAULT_BACKGROUND_GRADIENT_END_COLOR;
    int primaryColor = Colors.DEFAULT_PRIMARY_COLOR;
    int textColor = Colors.DEFAULT_TEXT_COLOR;

    Cursor cursor = application.getContentResolver().
                    query(ColorsContract.Color.
                    CONTENT_URI, null, null,
                    null, null);

    if (cursor != null) {

        int columnKeyIndex =
                     cursor.getColumnIndex(
                     ColorsContract.Color.COLUMN_KEY);
        int columnValueIndex =
                     cursor.getColumnIndex(
                     ColorsContract.Color.COLUMN_VALUE);

        while (cursor.moveToNext()) {

            String key = cursor.getString(columnKeyIndex);
            int value = cursor.getInt(columnValueIndex);

            switch (key) {
```

```
                case ColorsContract.Color.KEY_ACCENT:
                     accentColor = value;
                     break;
                case ColorsContract.Color
                     .KEY_BACKGROUND_GRADIENT_BEGIN:
                     backgroundGradientBeginColor = value;
                     break;
                  case ColorsContract.Color
                       .KEY_BACKGROUND_GRADIENT_END:
                       backgroundGradientEndColor = value;
                       break;
                  case ColorsContract.Color.KEY_PRIMARY:
                       primaryColor = value;
                       break;
                  case ColorsContract.Color.KEY_TEXT:
                       textColor = value;
                       break;

             }

        }

        cursor.close();

    }

    return new Colors(accentColor,
                backgroundGradientBeginColor,
                backgroundGradientEndColor, primaryColor,
                textColor);

}
```

ThemesFragment can request new theme's colors insertion by calling the putColors method which implies the *insert()* method to be executed from the *ColorsProvider* already depicted:

```
@Override
public void putColors(@NonNull Colors colors) {

    insertColor(ColorsContract.Color.KEY_ACCENT,
                colors.accent);

    insertColor(ColorsContract.Color.
                KEY_BACKGROUND_GRADIENT_BEGIN,
                colors.backgroundGradientBegin);

    insertColor(ColorsContract.Color.
                KEY_BACKGROUND_GRADIENT_END,
                colors.backgroundGradientEnd);

    insertColor(ColorsContract.Color.
                KEY_PRIMARY, colors.primary);

    insertColor(ColorsContract.Color.
                KEY_TEXT, colors.text);

}


private void insertColor(@NonNull String key, int value) {

    ContentValues contentValues = new ContentValues();
    contentValues.put(ColorsContract.Color.
                      COLUMN_KEY, key);
    contentValues.put(ColorsContract.Color.
                      COLUMN_VALUE, value);

    application.getContentResolver().insert(
              ColorsContract.Color.
              CONTENT_URI, contentValues);

}
```

5.2.2 *Connectivity* fragment

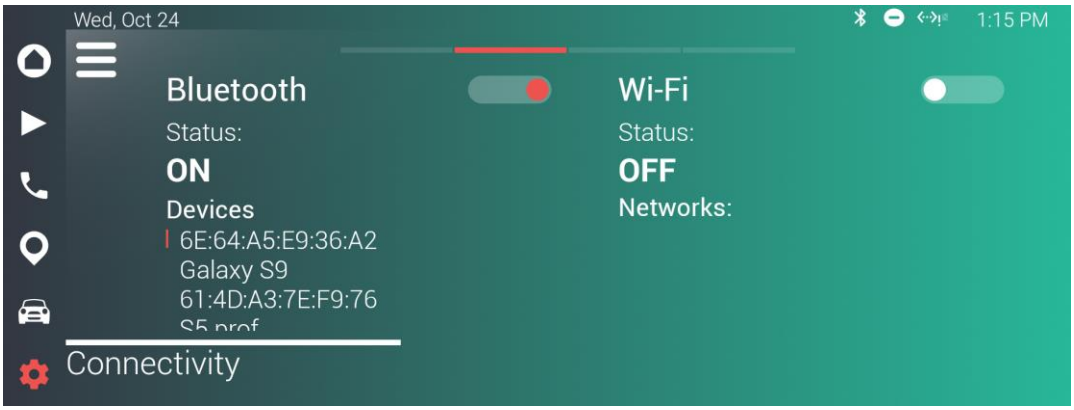


Figure 21. Preferences –Connectivity Fragment layout

The connectivity fragment operates on system connectivity management through native APIs provided by android. It lets the user enable Bluetooth or Wi-Fi connection and scan for visible devices or networks. Once fragment view is created, it instantiates a *WifiController* and a *BluetoothController*.

WifiController constantly scans for available networks by implementing *WifiTracker.WifiListener* interface, an android native element that internally registers a *BroadcastReceiver* in order to be notified from the system when new results are available.

Found networks are then loaded into a listView as *AccessPoints* objects. Once the user clicks over one of the items, an *AccessPointClickListener* calls the *onWifiClicked()* passing the selected AccessPoint object as parameter. This method, from the *WifiController* class, checks the security level of the AccessPoint to understand if selected network is public and eventually directly connecting to it. Otherwise, if it is an already known private network (whose authentication data has been previously specified)

tries the connection, else asks for credentials by showing a “credential” fragment and, after correct user input, it connects.

In the following the portion of code for the *onWifiClicked()* method:

```
@Override
public void onWifiClicked(AccessPoint accessPoint) {
   // for new open unsecured wifi network, connect to
         it
      if (accessPoint.getSecurity() ==
             AccessPoint.SECURITY_NONE &&
             !accessPoint.isActive()) {

          mCarWifiManager.connectToPublicWifi(accessPoint,
                                    mConnectionListener);
      }
      else
      {
          if (!accessPoint.isSaved() &&
                    !accessPoint.isActive())
          {
             Bundle accessPointState = new Bundle();
             accessPoint.saveWifiState(accessPointState);
             WifiCredentialFragment credentialFragment =
                   new WifiCredentialFragment();

      credentialFragment.setArguments(accessPointState);
      mFragmentManager.beginTransaction()
           .add(com.magnetimarelli.mm_car_ui_lib.R.id
                  .basic_activity_container,
                  credentialFragment,
                 "wifi_credentials_fragment").commit();
          }
          else
          {

          mWifiManager.connect(accessPoint.getConfig(),
                       mConnectionListener);
          }
      }
}
```

BluetoothController component is instead based on the use of a *LocalBluetoothManager* that provides a simplified interface on top of a subset of the Android Bluetooth API. Like the *WifiController*, it scans for available devices by registering a *BroadcastReceiver*. Each new found device is loaded into a listView as *CachedBluetoothDevice* objects. An object of this type represents a remote Bluetooth device containing attributes like address, name and RSSI.

Once the user clicks over an available device in the list, the *BluetoothClickListener* in its *onClick()* callback retrieves from the adapter the related *CachedBluetoothDevice*. Three conditions are then evaluated:

- The device is actually connected so a disconnection is executed.
- The device is a known paired one; a straight-forward connection is accomplished.
- The device is not paired; a pairing fragment is shown in order to pair the device and, on success, eventually connect.

5.2.3 *Volumes* fragment

The system volumes fragment lets the user set media, phone and notifications volumes by just interacting with three *seekBars*. Three *VolumeControllerPresenter* objects, are instantiated in the *onViewCreated()* callback of the volumes fragment, one for each *seekBar* slider, setting the relative *StreamType* that can take the following values:

- AudioManager. STREAM
- AudioManager.STREAM_VOICE_CALL

- AudioManager.STREAM_NOTIFICATION

The *VolumeControllerPresenter* controls UI interactions with the sliders by registering listeners on their progress changes, accordingly modifying system volumes by acting on an *AudioManager* object, instance of a class that provides APIs for managing system volumes and ringer profiles (silent, vibrate, loud).

At fragment startup, the *AudioManager* retrieves current and maximum volume levels, in order to set seekBars maximum values and current progress, by means of the following lines of code:

```
mSeekBar.setMax(mAudioManager.
                    getStreamMaxVolume(StreamType));


mSeekBar.setProgress(mAudioManager.
                    getStreamVolume(StreamType));
```

Listeners registered on the seekBars react to interactions by calling the *onProgressChanged()* callback that sets the current *StreamType* volume as following (notice that each of the three *VolumeControllerPresenter* has its own *StreamType* set at construction time):

```
mAudioManager.setStreamVolume(mStreamType,
                  progress, AudioManager.FLAG_PLAY_SOUND);
```

where *progress* is an integer value corresponding to the actual progress level of the touched seekBar and *AudioManager.FLAG_PLAY_SOUND* is a flag that indicates to play a sound while changing the volume as feedback for the user.

In order to create a better user interaction scheme for acting on infotainment volumes with the aim of reducing user distractions while driving, the Rotary controller allows to change volumes using its knob. The SocketService itself acts on the STREAM_MUSIC volume when a related message is received from the Rotary.

5.3 *Stream* service

In android automotive the system is natively able to generate, through the *Stream* service, StreamCards, parcelables carrying data and used for communication between various components. *StreamCards* are available to all the applications that implement the interface *IStreamConsumer* in order to bind to the service and get notified when new StreamCards are produced.

For each application that should post some of their data, a *StreamProducer* must be implemented. It is in charge of fetching data from given application's controllers or managers, and post the generated StreamCard to the Stream service in order to forward it to registered consumers.

As already said, the Overview application presents data coming from various apps and for this reason it must be notified when new data is available. Overview app is a consumer for the Stream service, implementing the IStreamConsumer interface in order to retrieve data from three

applications. In particular, in our Infotainment system, four producers have been implemented:

Radio: *RadioStreamProducer* connects to the RadioManager in order to retrieve current radio band and channel frequency.

Media: *MediaStreamProducer* which is bound to the *MediaPlaybackMonitor* service connected to a *MediaStateManager* to retrieve media updates (track playback state and metadata).

Telephone current active call: *CurrentCallStreamProducer* listens for active call events to produce a StreamCard. In particular it starts a Broadcast Receiver (an android component which allows to register for system or application events) called *CurrentCallActionReceiver*, to be notified for current call events.

```
private class CurrentCallActionReceiver extends
BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {

       String intentAction = intent.getAction();

       if(!TelecomConstants.
            INTENT_ACTION_STREAM_CALL_CONTROL
            .equals(intentAction))
            return;

        String action =
              intent.getStringExtra(
              TelecomConstants.EXTRA_STREAM_CALL_ACTION);

        switch (action) {
```

```
            case TelecomConstants.ACTION_MUTE:
                mInCallService.setMuted(true);
                break;

            case TelecomConstants.ACTION_UNMUTE:
                mInCallService.setMuted(false);
                break;

            case TelecomConstants.ACTION_ACCEPT_CALL:
                acceptCall();
                break;

            case TelecomConstants.ACTION_HANG_UP_CALL:
                disconnectCall();
                break;

            default:
        }
    }
}
```

Telephone recent calls: *RecentCallStreamProducer* loads, from the connected phone, the call log to produce a StreamCard. In particular, it creates a CursorLoader that once started queries android native CallLog content provider:

```
private CursorLoader createCallLogLoader()
{
    StringBuilder where = new StringBuilder();
    List<String> selectionArgs = new ArrayList<String>();
    String selection = where.length() > 0 ?
                            where.toString() : null;

    Uri uri = CallLog.Calls.CONTENT_URI.buildUpon()
      .appendQueryParameter(CallLog.Calls.LIMIT_PARAM_KEY,
      Integer.toString(CALL_LOG_QUERY_LIMIT))
      .build();
```

```
    CursorLoader loader = new CursorLoader(
              mContext, uri, null, selection,
              selectionArgs.toArray(EMPTY_STRING_ARRAY),
              CallLog.Calls.DEFAULT_SORT_ORDER);

    loader.registerListener(0, this
                           /*OnLoadCompleteListener*/);

    return loader;

}
```

Once CallLog has been loaded the *onLoadComplete()* callback is invoked. From the actual call, number and date are extracted and a StreamCard is created.

```
@Override
public void onLoadComplete(Loader<Cursor> loader, Cursor
cursor) {

    if (cursor == null || cursor.moveToFirst()) {
        return;
    }

    int column = cursor.getColumnIndex(
                               CallLog.Calls.NUMBER);

    String number = cursor.getString(column);
    column = cursor.getColumnIndex(CallLog.Calls.DATE);
    long callTimeMs = cursor.getLong(column);

    // Display if we have a phone number, and the call was
    // within 6 hours (to display just calls in last 6
    // hours).
    number = number.replaceAll("[^0-9]", "");
    long timestamp = System.currentTimeMillis();
    long digits = Long.parseLong(number);
```

```
   if (!TextUtils.isEmpty(number) &&
      (timestamp - callTimeMs) < RECENT_CALL_TIME_RANGE)
   {
      if (mCurrentStreamCard == null || mCurrentNumber !=
          digits)
      {
         removeCard(mCurrentStreamCard);
         mCurrentStreamCard = mConverter.createStreamCard(
                              mContext, number, timestamp);

         mCurrentNumber = digits;
         postCard(mCurrentStreamCard);
      }

   }

}
```

5.4 *SocketService* application

As already anticipated, a messaging protocol has been implemented in order to exchange data among infotainment applications and external hardware components like Rotary controller and Info-Cluster. The SocketService enables in-vehicle infotainment repetitions to maintain data consistency in information brought to the user on the various in-car displays and, in addition, provides the transmission of control commands coming from the rotary controller to the Android OS and vice versa.

An example about its utility is given if we consider the selection of a new theme for the IVI user interface from the Preferences application. When the user chooses a theme, the new color combination must be notified to the Info-Cluster and Rotary in order to update their layout and be consistent

with the user choice. This is just one of the information SocketService forwards. In the following, a terse list of data that this service handles and sends to connected clients, in order to underline its role and importance in the whole software architecture:

- **Source messages:** audio source that is currently active (one among radio, media or phone);
- **Menu control messages:** drawer menu current state (open/closed) and rotary commands to allow menus navigation;
- **Rotary volume commands:** the service interprets volume control commands coming from the rotary and sets system volume accordingly;
- **Colors messages:** current system UI color combination;
- **Media messages:** current track metadata, timing and cover (if any). This data is shown both on Cluster and Rotary;
- **Tuner messages:** current radio station data and cover (if any);
- **Phone messages:** call data, elapsed time, contact image (if any) and call state;
- **Car messages:** data about vehicle status;
- **Time messages:** current date and time taken from the Android system and forwarded to the Cluster.

If necessary, a deeper list about exchanged messages is available in the *Appendix B* section at the end of this thesis.

The service dispatches incoming messages (sent by applications using a *Messanger* interface exposed by the service itself) to:

- *SocketServer* on which the Cluster connects, transmitting raw data formatted in TLV (Type-Length-Value) encoding scheme.

- *WebSocketServer* for rotary controller connection based on HTTP communication.

They both get instantiated the first time an infotainment application binds to the service, in particular in the service's *onBind()* callback. In fact, in Android environment, a "bound service" gets started and lives until another component is connected (so bounded) to it.

This callback is also in charge of instantiating a *DataCenter* object, whose task is to keep a snapshot of current infotainment data state.

In particular, it stores:

```
private MediaData mMediaData;
private TunerData mTunerData;
private PhoneData mPhoneData;
private DataColors mDataColors;
```

The same is done for vehicle data that is kept in a different object named *CarData,* containing following values:

```
int pCarSpeed                // current speed
int pLongitudinalAcc         // longitudinal cceleration
int pTransversalAcc          // transversal acceleration
int pGearValue                      // current gear
int pDoorLockState           // current doors lock state
int pDriverDoorState         // driver door open/close
int pPassengerDoorState
int pBehindDriverDoorState
int pBehindPassengerDoorState
int pTrunkDoorState          // trunk door open/close
int pOilLevel                // oil level
int pTorque                  // current torque
int pInternalTemp            // internal temperature
```

```
int pExternalTemp           // external temperature
int pEngineTemp                    // engine temperature
int pEngineOilTemp          // engine oil temperature
// GPS data from CAN
int pCanLatitude
int pCanLongitude
int pCanGPSSpeed
int pCanGPSHeading
int pCanGPSAltitude
// data integrity check
int pCrc
```

The SocketService opens a client thread (named *MyCarClient*) to read data from the server in the EntryNAV that plays the role of gateway for the CAN bus.

Along with these initializations, the service starts a *BroadcastReceiver* for system time changes detection in order to send through the SocketServer current date and time, to be displayed on the Info-Cluster.

The service is thought to forward a new message to connected clients each time new data is available, so when an application posts a message through the *Messanger* interface returned by the service's *onBind()* method. The Messanger defines a *Handler* named *IncomingHandler* to interpret incoming messages by the *HandleMessage* method which retrieves the message type by reading the "*what"* attribute, a user-defined message code among one of those specified in a *MessageHelper* class.

Forwarding a new message just when new data is posted to the Messanger caused infotainment appendices to be not consistent with Android OS at system startup. Data like colors could not be sent until the user explicitly changed the theme selection through the Preferences application.

This problem was solved by using DataCenter and CarData objects. Both have some methods to retrieve and store all possible data from the Android system (such as current system color combination taken from the color content provider) in order to have a “snapshot” ready to be sent to clients. When a new client connects to the SocketServer and WebSocketServer, they send all the data available in DataCenter and CarData. In this way, at system startup, both Rotary and Cluster display data consistent with current infotainment status.

5.5 *Media* application

The system natively provides the media player application to bring just entertaining functionalities originally coming from two sources:

- from an external USB storage or local media,
- from a connected Bluetooth device.

A third source has been added in order to support web radio. The webRadio application plays audio from internet podcasts when the IVI is connected to internet via wi-fi (for instance using a mobile device tethering functionality).

Once media application is open, it starts a *MetadataService.* It is a service that we implemented for exchanging current track metadata with infotainment appendices through the *SocketService* by binding to it. This enables In-Vehicle repetitions of media data.

To make it possible to be aware of current track being played and to retrieve its attributes and information, the *MetadataService*, once started, gets an instance of the *MediaManager*, an Android class that manages which audio source the application should bind to, and registers a listener for media app changes.

The *MediaManager* instantiates a *MediaBrowser* object that operates as a client for a second native Android service performing two main tasks:

- It connects to a *MediaBrowserService*, a service able to get the root node of the content hierarchy in order to fetch available media items.
- Once connected to the *MediaBrowserService*, it creates a *MediaController* for managing UI interactions.

With this approach, the *MediaBrowser* can traverse the content hierarchy obtaining a list of *MediaBrowser.MediaItem* objects. Each of these items has a type (*BROWSABLE or PLAYABLE*) and a unique ID. When the MediaBrowser is asked to browse or play an item, the corresponding ID is used.

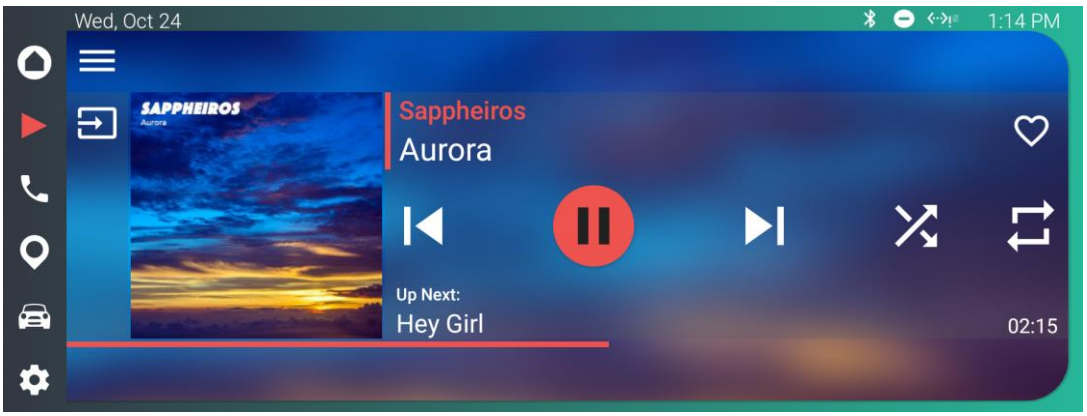


Figure 22. Media– CardView for media player

Media application renders a layout based on a cardView which welcomes current track metadata (song, album, artist name and current timing), skip backward or forward, shuffle, repeat, play and pause buttons, along with eventually the album cover (if not present a default cover placeholder is shown).

The *MediaController* handles interactions with UI elements and registers callbacks for Playback state (playing or paused) and metadata changes by sending corresponding informational messages to the *SocketService* in order to update rotary and cluster displays with consistent data.

5.6 *Radio* application

The native radio application provided by Android Automotive has been customized both in layout and functionalities.

It has been modified by extending the custom *BasicDrawerMediaActivity* to provide a drawer menu and binding to system UI colors. In its *onCreate()* method a *RadioController* object is instantiated. This class manages the display of metadata on the UI based on current radio station. Moreover, it creates two other objects:

- *RadioDisplayController* that controls the appearance state of some UI elements such as the favorite list on the bottom of the layout;
- *RadioStorage* for persistent storage of radio data such as the favorite stations list.

The *RadioDisplayController* is also in charge of registering some listeners in order to react to user interactions with UI elements. Of particular interest are the interaction possibilities offered by the favorite stations list:

- A long press over one of its buttons records the current station being played as favorite by saving its name and frequency (station name and current program/track name are taken from the RDS – Radio Data System whose support was already integrated in Android Automotive). In case pressed button was the one already being playing, a long press deletes the favorite from the list.
- A short press over its buttons (if not empty) plays selected favorite station by synchronizing the radio on its frequency.

The bottom favorite bar has been graphically implemented as a *LinearLayout*; it contains six clickable boxes always rendered as *LinearLayout* and filled with two *TextViews* representing station name and frequency.

The data about favorite stations is managed by the *RadioStorage* class that operates addition and removal of presets from a persistent storage implemented as *SQL Database* through the *RadioDatabase* class that extends a helper class *SQLiteOpenHandler* natively provided by Android.

At application startup, the favorite stations are also loaded into a listView available through the drawer menu. From the same menu, along with source selection fragment (pre-loaded through the implementation of the *BasicDrawerActivity* as already depicted), a *ManualTunerFragment* is accessible; it enables the possibility to manually tune the radio on a valid frequency by interacting with a graphical numeric pad.

RadioStorage is also in charge of loading pre-scanned channels in a list of available radio channels found during tuner seeking or by using a secondary tuner antenna (if provided by the vehicle) that constantly scans radio band.

5.7 *Dialer* application

The Dialer application comes natively with android automotive OS but it has been modified both in user interface and functionalities.

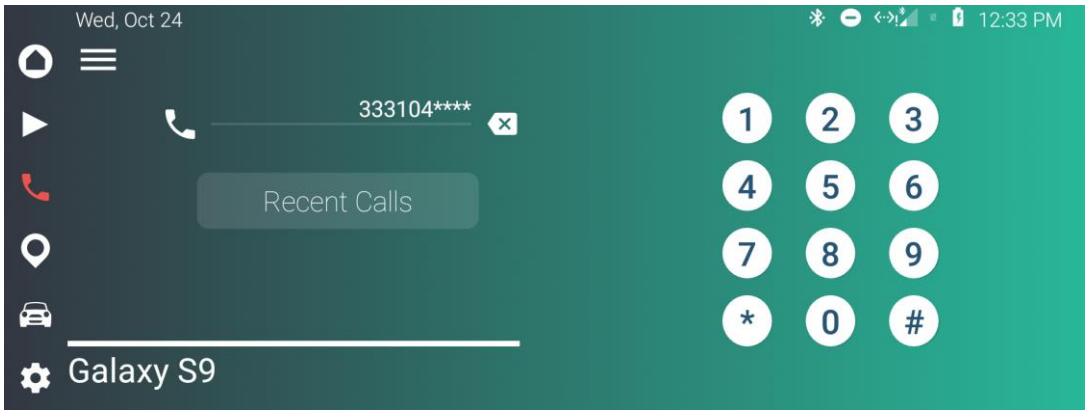


Figure 23. Dialer – Main application layout

To be functioning it requires a mobile device connected to the In-Vehicle Infotainment through a Bluetooth connection, to share its contacts and other data. In fact, it acts as a common mirror link application where all the data comes from the phone itself and the IVI offers a clean and fast to use interface to make or answer phone calls by using the provided numeric pad, search for recent contacts and manage on-going calls.

As any other infotainment application, the main activity *TelecomActivity* extends the *BasicDrawerActivity* class from our *mm_ui_lib* and loads the drawer menu with two fragments: “Recent calls” and “Lost calls”. In both cases, a listView is dynamically loaded with the recent calls made or received on the connected phone. For each of the items in the lists the contact name (if any), the number, the typology (mobile, home, office) and eventually a contact image (or the default android placeholder) are provided. A tap on one of the listViews items starts a call.

In order to retrieve contacts data from the phone, the native class *PhoneLoader* calls methods that perform asynchronous queries to a phone content provider. They are asynchronous calls in order to do not overload the main thread UI that is in charge of rendering layouts and managing user interactions.

TelecomActivity also binds to an opportune service called *PhoneDataService* that exposes callbacks for dialer state changes. It has been implemented with the purpose to periodically send updates to the *SocketService* by binding to it, in order to forward phone data to Info-Cluster.

Each callback is activated by an instance of the *UiCallManager*, a class that handles interactions with the user interface enabling telecom functionalities.

In particular the logic behind the *PhoneDataService* is:

1. When an incoming call is received the *onCallAdded()* callback is executed.

```
@Override
public void onCallAdded(UiCall call) {
    super.onCallAdded(call);
    Log.d("#### DIALER INFO" , "CALL ADDED" +
           call.getNumber());
    sendCallInfo(1,call.getNumber());
    Bitmap contactImage = TelecomUtils
              .getContactPhotoFromNumber(
              getContentResolver(),call.getNumber());

    if(contactImage!=null){
      sendContactBitmapToService(contactImage);
    }
}
```

It receives a UiCall object, an abstraction of a single call; it sends call information to the *SocketService* (type 1 stands for "ringing"), retrieves contact image from the phone's content provider and, if any, its bitmap is also sent.

2. Once an outgoing call starts or and incoming one is accepted the *onStateChanged()* is called:

```
@Override
public void onStateChanged(UiCall call) {
    super.onStateChanged(call, state);
    Log.d("#### DIALER INFO","STATE CHANGED"+state +
          "STATE TIME"+call.getConnectTimeMillis());
    sendCallInfo(2, call.getNumber());
    if(timer==null) {
        timer = new Timer();
        timer.schedule(new TimerSenderTask(
                       call.getConnectTimeMillis()),
                        0, milliStep);
    }

}
```

It receives a UiCall object, an abstraction of a single call; it sends call information to the *SocketService* (type 1 stands for "ringing"), retrieves contact image from the phone's content provider and, if any, its bitmap is also sent. It sends a new message to the *SocketService* with counterpart phone number and type 2 meaning call is "Accepted". In case a timer has not been scheduled, it instantiates a new one which each second will keep the *SocketService* updated about current call timing.

3. When a call ends, the *onCallRemoved()* is executed:

```
    Log.d("#### DIALER INFO","CALL
            REMOVED"+call.getNumber());
     sendCallInfo(3,call.getNumber());

     if(timer!=null){
         timer.cancel();
         timer=null;
     }
}
```

It sends a new call information message to the *SocketService* with type 3 that equals to "Declined" and cancels the timer.

5.8 *MyCar* application

MyCar application provides vehicle information rendered in a *tabLayout*, managed by a *viewPager* that loads three different fragments. Each fragment shows various information about vehicle such as mileage, current trip data, fuel level or tires pressure through animated gauges and graphs.

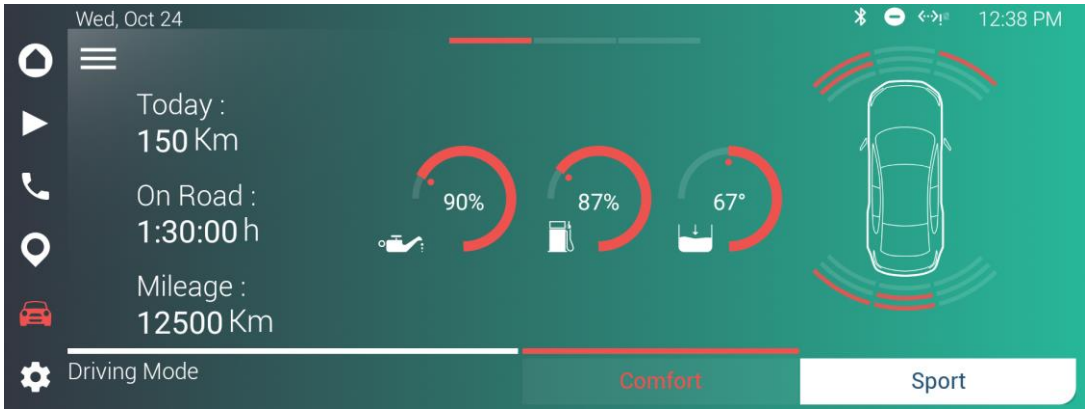


Figure 24. MyCarApp – First tabLayout fragment

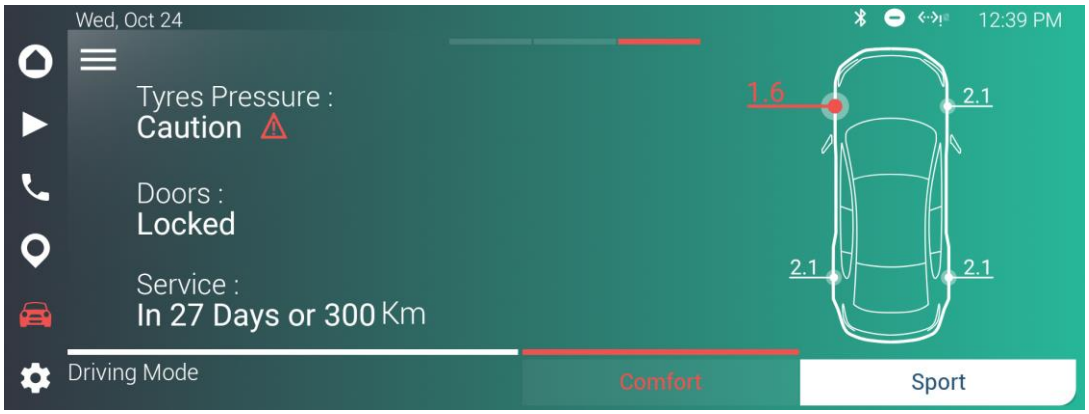


Figure 25. MyCarApp – Third tabLayout fragment

Moreover, the application supports controlling vehicle components and sensors through various settings accessible in the drawer menu:

- Lights controls
- Unit settings for distances (Km or miles), temperatures (C° or F°), pressures and fuel consumption.
- Safety related controls for speed limiter, lane departure warning and rear view or park sensor.
- Doors and Locks controls
- Cluster settings

Each of the settings pages extends the abstract class *BaseSettingPage* that binds a *RestoreFragment* to a "Reset" textView. This fragment asks for confirmation to restore default values in the current settings page when the user presses on the aforementioned textView.

In order to save and retrieve current user's settings a *SettingsSharedPreferences* class has been implemented. It exposes methods to get and put values from the *car_settings_preferences* by providing just the key and the value (in case it is a get operation, the passed value is the default one returned if the required key is absent from the storage).

At current implementation status this application just simulates vehicle data by means of a random generator but, in future development, these information will be taken from the CAN network by implementing a vehicle HAL. The vehicle HAL will be necessary also to enable controlling vehicle components like sensors, doors and lights.

5.9 *Navigation* application

Actual navigator implementation uses a proprietary application from Mireo, a company specialized in GPS navigation software solutions. It features turn-by-turn navigation and voice guidance supporting multiple languages. In order to provide the infotainment system with navigation data, the *SocketService* application after receiving GPS data generated from the vehicle built-in sensor (forwarded through the CAN bus along with other car data as already explained) sets a mock location in the Android system as follows:

```
Location lMockLocation = new
Location(LocationManager.GPS_PROVIDER);

if (lMockLocation != null) {
    long lTime = System.currentTimeMillis();
    long elTime = SystemClock.elapsedRealtimeNanos();
    lMockLocation.setLatitude(mCurrentCarData
       .getRealNavLatitude(mCurrentCarData
       .getCanLatitude()));
    lMockLocation.setLongitude(mCurrentCarData
       .getRealNavLongitude(mCurrentCarData
       .getCanLongitude()));
    lMockLocation.setAccuracy(1);
    lMockLocation.setAltitude(0);
    lMockLocation.setTime(lTime);
    lMockLocation.setElapsedRealtimeNanos(elTime);
    lMockLocation.setBearing(mCurrentCarData
       .getRealCanGPSHeading(mCurrentCarData
       .getCanGPSHeading()));
    lMockLocation.setBearingAccuracyDegrees(1);
```

```
    try {
       mLocationManager.setTestProviderLocation(
              LocationManager.GPS_PROVIDER,
              lMockLocation);
    } catch(SecurityException excp) {
       Log.i(TAG, "It seems this app is not allowed
                   to access MockLocation!");
    }
}
```

By setting a mock location, the system is tricked thinking it is in a given location.

Chapter 6

System performance evaluation

Chapter overview

This chapter will analyze the performance we obtained with our infotainment implementation, in order to define if it can be further improved even in terms of additional features that can require a higher amount of resources. This analysis will test system reactiveness and the reliability of the Info-Cluster, one of the critical components in a vehicle because it provides car related data useful also in guaranteeing users safety.

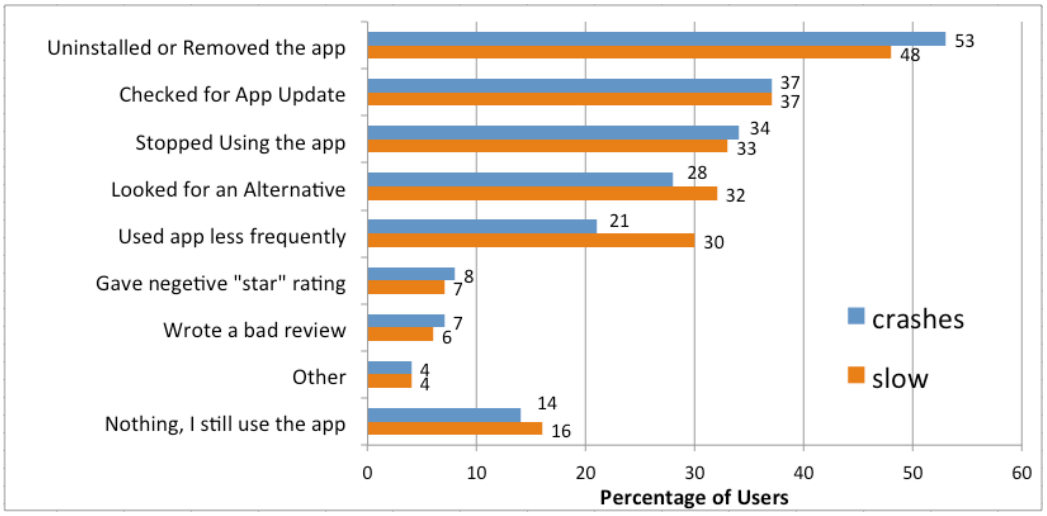
At the end, a table that summarizes the features of the Infotainment system will be provided.

6.1 Memory and CPU usage

As already depicted in chapter 3, our infotainment system uses a hypervisor to enable the same SoC to run two different OS that share hardware resources like CPU and memory. In our implementation, we reserved about two gigabytes of RAM (out of the four gigabytes available on the Qualcomm development board) to the Android OS to run its environment and applications. This means that the multi-tasking environment we created has a limit and that, at a certain point, the Android OS will start to terminate applications and free memory based on its native *Garbage Collector* when it requires additional resources. This android component is able to identify unused data references and reclaim memory from an application or, if necessary, entirely kill it (if the driver has put the application in background or is not actually using it) in order to free up memory for urgent tasks.

The mean usage of CPU and memory is a good metric to be considered in order to understand how much the system is further enlargeable in terms of features, still having a reactive and reliable system without requiring it to kill applications that maybe, even if not actually in foreground, the driver needs. By profiling processes, we can understand how to increase overall system performance and even applications loading and response time. Human engagement studies have shown that actions that responses under 100 milliseconds are perceived as instant, where actions that take a second or more allow the human mind to become distracted. [23]

In March 2015, HP published a study that shows customers react to slow applications the same way they do with applications that crash (*Graph 3*). [24]



Graph 3. Customer reactions to bad applications behaviors

The android environment furnishes various tools to profile the system running the OS and to exhibit data and statistics. Two useful tools for our analysis are:

- **ADB – Android Debug Bridge**: a command-line tool that facilitates a variety of device actions, such as installing and debugging applications, and that provides access to a UNIX shell to run other commands.
- **Android Profiler**: an Android Studio integrated tool to analyze real-time data on how applications use CPU, memory network and battery.

By executing the command "*adb shell dumpsys meminfo"* is possible to retrieve data about memory usage by all the processes running on the system. [25] In particular, some interesting data about free and used RAM is available:

```
Total RAM: 1,858,344K (status normal)
Free RAM:  907,421K   (93,649K cached pss + 414,184K cached
kernel + 399,588K free)
Used RAM:  940,173K (774,693K used pss +   165,480K
kernel)
Lost RAM:  8,714K
ZRAM:      2,036K physical used for 3,152K in swap
                (524,284K total swap)
```

Where the "Lost RAM" value is computed as difference between the Total RAM and the sum of Used and Free memory. Even if the value reported is relatively low and does not affect the correct behavior of the system and its applications, it can suggest some orphaned allocations of memory to be further inspected in order to improve performance.

In addition, *dumpsys* provides a list of used memory per process. An extract is shown below:

Total PSS by process:
94,967K: system (pid 390)
47,306K: zygote (pid 187)
41,098K: com.android.systemui (pid 767)
34,727K: logd (pid 165)
21,053K: perfd (pid 11405)
20,439K: com.android.car.media (pid 1580 / activities)
20,119K: com.zirak.automotive.overview (pid 30945 / activities)
19,066K: com.android.phone (pid 911)
16,589K: zygote64 (pid 186)
16,514K: media.codec (pid 422)
14,965K: surfaceflinger (pid 285)
12,745K: com.android.bluetooth (pid 737)

These results highlight that the system process is one of the hungriest. It constantly runs since system startup and is in charge of handling various tasks related to the Android environment such as job scheduling and garbage collection. This result got confirmed while profiling infotainment processes by using the Android Profiler which provides more precise real-time data:

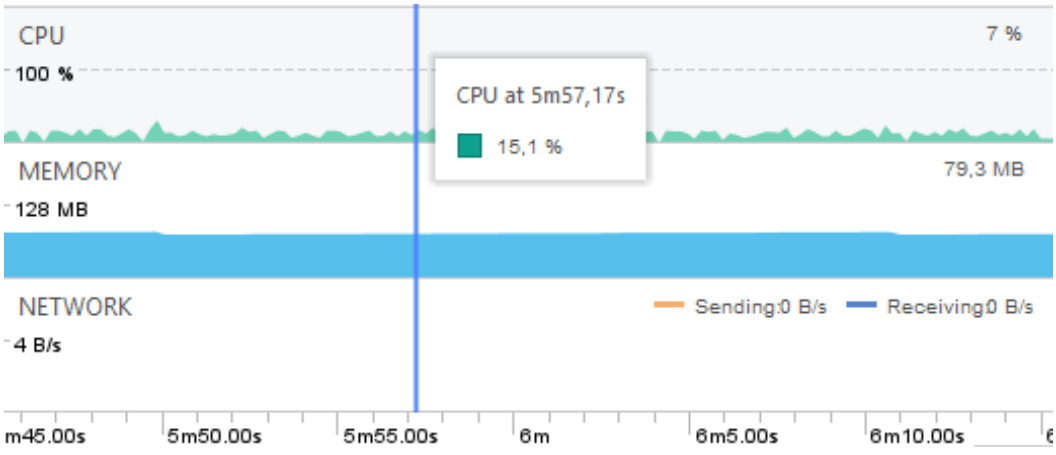


Figure 26. System process profiling graph

As suggested by the dumpsys's processes memory usage list, two other user area applications that require RAM the most are *Overview* and *Media*.

By profiling the process of the *Overview* application, the result is the following:

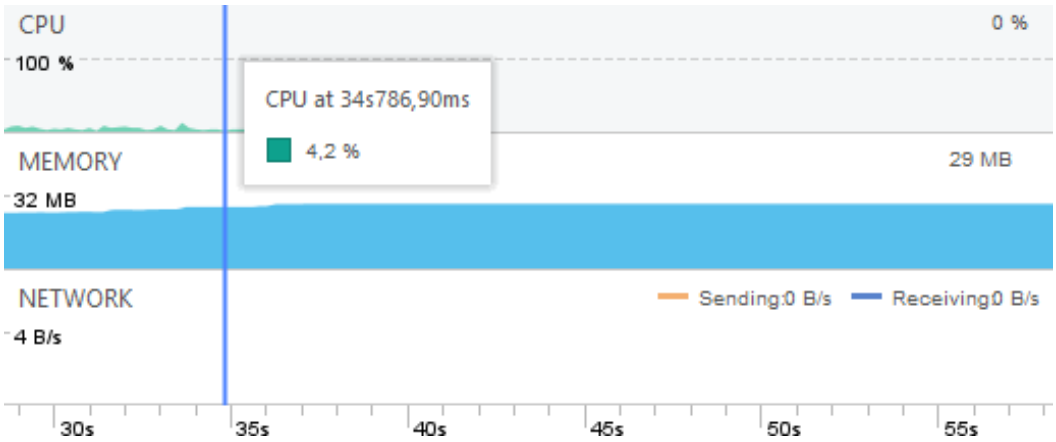


Figure 27. Overview app profiling graph

Apart from the handling of interactions with some of its UI elements (from this test, the car data *tabLayout* was being browsed by swiping its pages) the CPU is often unused. Moreover, no data gets exchanged on the network because, as previously explained, it collects information it needs from the *Stream* service to which this application is bound to. Memory usage is also low and it is mainly dedicated to graphical memory in order to render user interface elements.

The *adb* by means of the *"dumpsys activity"* command also exhibits data about services and content providers connections. For instance, about the *Stream* service it is possible to confirm what has been stated in the previous chapter:

```
ServiceRecord{8cb0920 u0 com.android.car.stream/.StreamService}
app=ProcessRecord{6dbcc26 1425:com.android.car.stream/u0a24}
created=-5h16m45s262ms started=true connections=5
Connections:
act=stream_consumer_bind_action ->
30945:com.zirak.automotive.overview/1000
act=stream_producer_bind_action ->
1425:com.android.car.stream/u0a24
act=stream_producer_bind_action ->
1425:com.android.car.stream/u0a24
act=stream_producer_bind_action ->
1425:com.android.car.stream/u0a24
act=stream_producer_bind_action ->
1425:com.android.car.stream/u0a24
```

The *Stream* service performs five connections with four producers and one consumer that is the *Overview* application.

Of particular interest, is also the part where the *dumpsys* outlines content providers connections. As expected, each application that gets open by the user connects to the *ColorsProvider* in order to retrieve current color combination:

```
ContentProviderRecord{99910b1 u0 com.magnetimarelli.preferences/
.colorpicker.provider.ColorsProvider}
proc=ProcessRecord{c58175:com.magnetimarelli.preferences/1000}
authority=it.zirak.automotive.colorpicker.provider
4 connections, 0 external handles
-> 1580:com.android.car.media/u0a20 s0/0 u1/1 +5h20m3s22ms
-> 30945:com.zirak.automotive.overview/1000 s0/0 u1/1
+3h58m40s311ms
-> 14287:com.android.car.dialer/u0a18 s0/0 u1/1 +8s61ms
-> 14446:com.magnetimarelli.mm_mycar/u0a21 s0/0 u1/1 +4s742ms
```

Another interesting component to be profiled is the *SocketService* application. Being a service, it does not render any UI element having a low memory usage (around 4-5 Megabytes) but instead it uses the network in order to send data on the Socket and WebSocket.

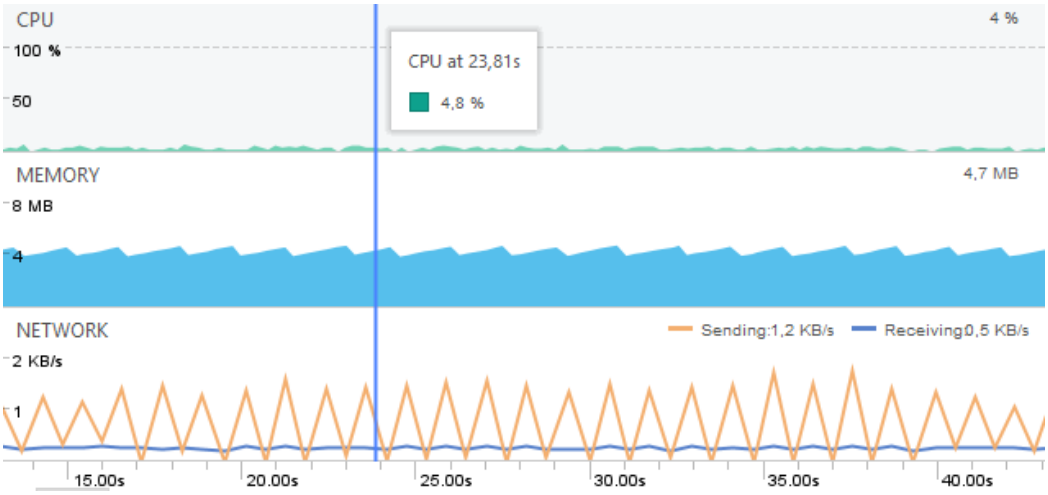


Figure 28. SocketService profiling graph

The sawtooth-like aspect of the network curve is due to the sending and reception of single messages on the network, that require processing and so is not a continuous flow. It is interesting to be noticed that the amount of outgoing data is three or four times greater than the incoming one. This is justified if we consider that vehicle data is sent to the Info-Cluster passing through the Infotainment (and so *SocketService*). As aforementioned, in future development a vehicle HAL will be implemented; this solution will drastically lower the outgoing *SocketService* network flow leaving more bandwidth purely for Infotainment data, guaranteeing data consistency and layout coherence between the three IVI displays.

At actual state, the Cluster is up and running in less than 250 milliseconds while the Android system startup still requires an average time of 10 seconds in order to be operative. This shows that on both sides a tweaking is required in order to lower system boot time.

For what regards applications reactiveness, for each of them a cold start (when the application is started for the first time) requires less than 2 seconds, while a warm start (application already running in background) needs less than 1 second making system performance comparable to what the user are habit to with mobile devices.

Here a comparison between cold and warm start for the Media application extracted from the android log:

```
384-537/system_process I/ActivityManager: Displayed
com.android.car.media/.MediaActivity: +1s363ms

384-537/system_process I/ActivityManager: Displayed
com.android.car.media/.MediaActivity: +870ms
```

Finally, the reliability of the Info-Cluster has been tested by checking that a crash in the Android system did not affect the Cluster and verifying the correct implementation and setting of the hypervisor. In order to do this was necessary to terminate the process running the Android OS from the QNX hypervisor shell.

6.2 Infotainment features

In the following, a summary of the features that the system actually provides and, for completeness, the ones that is expected to implement in future development:

Display	
Number of Displays	3
Touch	Yes for Infotainment and Rotary display. No for Info-Cluster screen.
Multi-Touch	Yes for Infotainment and Rotary display.
Hardware, sensors and connectivity	
GPS	Yes
A-GPS	Yes (through internet connection)
Rear-camera view support	Not implemented yet
Microphone	Yes
GPRS	Yes
3G/4G	Yes
Wi-Fi	Yes
Bluetooth	Yes
Infotainment physical controller	Yes (Rotary)
General system features	
Operative System version	Android Automotive OS 8.1
Updatable OS	Yes
System languages	English only (other languages to be implemented)
Extensible applications set	Yes
Applications store	To be implemented (Google Play Store)
Internet Browser	Not implemented yet
Vocal Assistant	To be implemented (Google Assistant)
Gestures	Being implemented through Rotary touch display
Home application	Yes (*Overview* app)

Car status application	Yes (*MyCar* app)
System colors customization	Yes
Media	
Radio	Yes
Supported bands	AM/FM
Web Radio	Yes
Podcasts	Yes
Media player	Yes (from external USB or internal storage). Actually supports Audio reproduction only.
Supported audio formats	MP3, WAV, M4A, AAC, OGG
Supported video formats	No video player
Bluetooth streaming from connected device	Yes
Navigation	
Software	Genius Maps by Mireo (Google Maps to be implemented, license required)
Map	Yes
Voice guide	Yes
Turn-by-Turn	Yes
Street View	No (will be available with Google Maps)
Night Mode	Yes
Multi-language	Yes
Communication (through connected mobile device)	
Voice calls	Yes
SMS or other messaging apps	Not implemented yet
Recent contacts	Yes

Chapter 7

Conclusions and future work

Conclusions and future work

At current implementation state, the initial requirements have been satisfied by developing a reliable and feature-full system. In conclusion, its strength points that make it different from what the Infotainment market actually offers, obtained by the adoption of the Android Automotive operating system, are summarized in these points:

- Real multi-tasking environment: This system can run multiple applications at once and keep active many services in background (like playing music).
- Functioning detached from a connected mobile device (apart for phone calls) and no mirror-linking required. This solution offers a complete hands-free usage of the Infotainment reducing distraction possibilities and in-vehicle mobile phone usage.
- Extendible features because it allows to install external applications like any other Android device does. This feature will be completely available when the Infotainment will provide the Google Play Store.
- Built-in 3G/4G antennas for internet connection.
- Better end-user learning curve because based on a well-known system like Android with millions of users.

In addition, the system offers clean user interfaces design and clever positioning of displays and infotainment components around the driver. Additional interaction schemes by the drawer menu usage, entirely controllable through the rotary, drastically decreases distraction possibilities with respect to other infotainment systems that do not provide natively a physical controller, keeping driver focus on the road.

The performance analysis, provided in previous chapter, also evidences system's points of weakness and underlines the need of further improvements such as obtaining lower system startup times. Although, it has stated that there is space for adding new features without overloading the system. As aforementioned in previous chapters, many other features are currently in development. Here a brief extract of what is currently under implementation:

- *Shortcuts and user gestures over the Rotary controller touchscreen*: This logic, partially implemented, will enable the possibility to interact with the infotainment by using gestures over the rotary touchscreen, such as long tap, double tap or swipe in order to provide commands to infotainment applications. Each application will react in a different way to gestures based on the functionalities it provides. One of its usage examples can be the possibility to skip media tracks by swiping on the rotary screen to the left or right. This will enhance distraction avoidance by not requiring driver focus on the Infotainment screen.
- *DNA functionality by means of rotary controller*: At current implementation state, the rotary renders on its display a window for DNA functionality. In future development, through this interface the driver will be able to select among three driving modes: Dynamic, Normal and All Weather. Each mode intervenes on vehicle electronics and mechanical parts with the aim of regulating road behavior and vehicle dynamics;
- *HAL implementation for vehicle data and controls*: The implementation of a vehicle HAL will be necessary in order to let the Android Infotainment communicate directly with vehicle systems and components.

Further improvements ideas that will be added in future development to the system are:

- Implementation of the vocal Google Assistant;
- Messaging application even through vocal assistance;
- As Android guidelines declare, must be added watchdog against denial of service attacks from the Android framework or third-party applications. A protection of this type can protect against malicious software flooding the vehicle network with traffic, which may lead to malfunctioning of vehicle subsystems;
- Implementation of a Head-Up Display, a transparent display on which data is projected without requiring users to look away from their usual viewpoints.

As result of this analysis, it is possible to infer that the implemented system is still highly enlargeable leaving space for numerous customizations and add-on applications that can bring more features and extend usage possibilities of this Infotainment.

Finally, this implementation work has confirmed that Android Automotive OS is a cost-effective solution for implementing an extendible and customizable system that can easily satisfy customer requirements in terms of features and interfaces design.

Appendix A.

Android native elements, computer science and technical terminology

In the following, a brief description of some Android native elements, computer science related words and other technical terminology used throughout the whole thesis work is provided in order to clarify their meaning.

Activity (Android)

An android component that provides a GUI and handles user interactions, acquires resources and manages notifications regarding its lifecycle. It plays the role of *controller* in the MVC pattern. An application can have multiple activities but one only entry point that takes the name of *MainActivity*.

Activity Lifecycle (Android)

The Android system constantly sends notifications to track the status of an application that, for instance, can be in background or going to be terminated. The programmer reacts to these events by performing some actions in callback functions related to the lifecycle. One of these callbacks is the *onCreate()* method that gets called at activity startup.

Adapter (Android)

An Adapter object acts as a bridge between a View with multiple children (such as a List or a Grid) and its underlying data. The Adapter provides access to the data items and it is also responsible for making a View for each item in the data set.

Bitmap

An image file format that represents data into a bit array. In its simplest form, for a black and white image, a single bit represents a pixel that can be white if its value is one, or black otherwise.

Broadcast receiver (Android)

An Android component that allows registering for system or application events. All registered receivers for an event are notified by the Android runtime once this event happens.

Bundle

A mapping from String keys to various parcels. A parcel is a container for a message (data or object references) that can be sent via Parcelable, a high-performance protocol for IPC transport.

Callback function

It is any executable code that is passed as an argument to other code that is expected to call back (execute). It can be also a function that gets called by the Operative system with the aim of handle particular events (i.e. user interactions).

CAN(Controller Area Network)

It is a multicast serial bus standard (mainly in the automotive environment), introduces in the eighties by Robert Bosch GmbH, to connect various Electronic Control Units (ECUs).

CardView (Android)

An Android material design graphical element that wraps its children views inside a card-like shape.

Context (Android)

In Android, it represents current state of the application/object. Typically, it gets called to retrieve information regarding another part of the program (activity and package/application).

Content provider (Android)

An Android component that allows exposing its data to other applications by means of Content Resolver. The Content Resolver includes the CRUD (create, read, update, delete) methods corresponding to the abstract methods (insert, query, update, delete) in the Content Provider class.

Cursor (Android)

An interface that provides random read-write access to the result set returned by a database query.

CursorLoader (Android)

A loader that queries the Content Resolver and returns a Cursor.

Denial Of Service (DOS)

Is a cyber-attack in which the perpetrator seeks to make a machine or network resource unavailable to its intended users by temporarily or indefinitely disrupting services of a host connected to the network. Denial of service is typically accomplished by flooding the targeted machine or resource with superfluous requests in an attempt to overload systems and prevent some or all legitimate requests from being fulfilled.

DRAM (Dynamic random-access memory)

Is a type of random access semiconductor memory that stores each bit of data in a separate tiny capacitor within an integrated circuit. A charge state of the capacitor corresponds to a value of 1, otherwise means 0. The electric charge on the capacitors slowly leaks off. To prevent this, DRAM requires an external memory refresh circuit which periodically rewrites the data in the capacitors, restoring them to their original charge. Because of this refresh requirement, it is dynamic memory.

Drawer menu (Android)

A menu common in Android applications that easily provides shortcuts and informational data.

DSP (Digital Signal Processor)

A specialized microprocessor optimized for the operational needs of digital signal processing. The goal of DSP is usually to measure, filter or compress continuous real-world analog signals.

ECU

Is any embedded system in automotive electronics that controls one or more of the electrical systems or subsystems in a vehicle.

EntryNAV

Original infotainment system in a vehicle.

Fragment (Android)

An Android element born to simplify the task of adapting an interface to various screens by dividing it in smaller blocks. A Fragment is an object that, conceptually, stands between Activity and View because has a lifecycle and can be directly inserted in a layout XML file.

Garbage Collector

In computer science, garbage collection (GC) is a form of automatic memory management. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program.

GPS (Global Positioning System)

A satellite-based radio-navigation system owned by the United States government and operated by the United States Air Force. Is a global navigation satellite system that provides geolocation and time information to a GPS receiver anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.

GPU(Graphics Processing Unit)

A specialized electronic circuit designed to rapidly manipulate and

alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.

Handler (Android)

A Handler is a class that allows sending and processing *Message* and *Runnable* objects associated with a thread's *MessageQueue*. Each Handler instance is associated with a single thread and that thread's message queue.

IPC

The **Inter-Process communication** is a mechanism that an operating system provides to allow the processes to manage shared data. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests.

LinearLayout (Android)

A view group that aligns all children in a single direction, vertically or horizontally.

ListView (Android)

A view group that displays a list of scrollable items. The list items are automatically inserted to the list using an Adapter that pulls content from a source such as an array or database query and converts each item result into a view that is placed into the list.

Log

In computer science, is an instrument for administration and monitoring of a system or application that exhibits statistics, error messages and other informational data.

MVC (Model – View – Controller)

Is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major

components allowing for efficient code reuse and parallel development.

PLC

A Programmable logical controller is an industrial digital computer which has been ruggedized and adapted for the control of manufacturing processes, such as assembly lines, or robotic devices, or any activity that requires high reliability control and ease of programming and process fault diagnosis.

PSS (Proportional Set Size)

Is a measurement of RAM usage that takes into account shared pages across processes. Any RAM pages that are unique to a process directly contribute to its PSS value, while pages that are shared with other processes contribute to the PSS value only in proportion to the amount of sharing. For example, a page that is shared between two processes will contribute half of its size to the PSS of each process.

RPC (Remote procedure call)

In distributed programming, is when a program causes a procedure (or subroutine) to be executed in a different address space (usually another computer or shared network) without the need that the programmer explicitly codes the details for the remote interaction. It is a form of inter-process communication where different processes have different address spaces.

RTOS

Is an operating system (OS) intended to serve real-time applications that process data as it comes in, typically without buffer delays in tenth of seconds or shorter increments of time. A real time system is a time bound system which has well defined fixed time constraints. Processing must be done within the defined constraints or the system will fail. They either are event driven or time sharing. Event driven systems switch between tasks based on their priorities while time sharing systems switch the task based

on clock interrupts. Most RTOS's use a pre-emptive scheduling algorithm.

SeekBar (Android)

Is an Android layout element that extends the ProgressBar by adding a draggable thumb. The user can touch the thumb and drag left or right to set the current progress level.

Service (Android)

Is an Android application component that can perform long-running operations in the background, and it does not provide a user interface. Another application component can start a service that will run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform inter-process communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

SoC (System-on-Chip)

Is an integrated circuit (also known as a "chip") that integrates all components of a computer or other electronic system. These components typically include a central processing unit (CPU), memory, input/output ports and secondary storage, all on a single substrate. Systems on chip are commonly used in embedded systems and the Internet of Things.

Socket

Is an internal endpoint for sending or receiving data within a node on a computer network. Concretely, it is a representation of this endpoint in networking software (protocol stack), such as an entry in a table (listing communication protocol, destination, status, etc.), and is a form of system resource.

SQL

Structured Query Language is a domain-specific language used in programming and designed for

managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data where there are relations between different entities/variables of the data.

StreamCard (Android)

Is a parcelable object that is used for communication between various stream components (in general producers and consumers). Each card is uniquely identified by a type and id.

TabLayout (Android)

An Android view element that provides an horizontal layout to display different tabs.

TCP

Transmission control protocol is one of the major protocols of the Internet protocol suite that complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as TCP/IP. It provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network. Major internet applications such as the WWW, email, remote administration, and file transfer rely on TCP.

TextView (Android)

A user interface element that displays text to the user. The user-editable version is the EditText.

UDP

User Datagram Protocol is one of the core members of the Internet protocol suite. It was designed by David P. Reed in 1980. With UDP, computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol (IP) network. Being connectionless, prior communications are not required in order to set up communication channels or data paths.

UI

User Interface is the space where interactions between humans and machines occur. The goal of this interaction is to allow effective operation and control of the machine from the human end, whilst the machine simultaneously feeds back information that aids the operators' decision-making process.

URI

Uniform Resource Identifier (URI) is a string of characters that unambiguously identifies a particular resource.

ViewPager (Android)

Is a layout manager that allows the user to flip left and right through pages of data, often used in conjunction with fragments. It needs an implementation of a PagerAdapter to generate the pages that the view shows.

VM – Virtual Machine

Is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer. Their implementations may involve specialized hardware, software, or a combination.

WebSocket

Is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. The WebSocket protocol enables interaction between a web client (such as a browser) and a web server with lower overheads, facilitating real-time data transfer from and to the server.

ZRAM

Formerly called compcache, is a Linux kernel module for creating a compressed block device in RAM that be used for swap or as general-purpose RAM disk.

Zygote (Android)

Is a special process in Android which handles the forking of each new application process (which are regular Linux processes). It is launched by the Android runtime, which also starts the first Virtual Machine (VM). The VM then calls Zygote's main() method which causes Zygote to preload all shared Java classes and resource into memory.

Appendix B.

SocketService messaging protocol

This appendix is oriented to detail the description of the messages exchanged by the SocketService messaging protocol. As already depicted the service communicates on a Socket with the Cluster, on a WebSocket with the Rotary.

For what regards the Socket communication, each message is encoded in TLV (Type-Length-Value) format as follows:

- Int32 message ID
- Int32 payload length
- Payload

The implemented messaging protocol, at current stage, prescribes the following messages to be exchanged:

MESSAGE NAME	*ID*	PAYLOAD DESCRIPTION
SOURCE_CURRENT_ACTIVE	*101*	Source id: 0 = Media, 1 = Radio, 2 = Phone
Color combination data		
COLOR_CURREN_ACTIVE	*401*	Background color start, background color end, primary color, secondary color
Current media data		
MEDIA_CURRENT_DATA	*501*	Artist | Album | Track name | length
MEDIA_CURRENT_IMG	*502*	Base64 encoded cover bitmap
MEDIA_CURRENT_TIME	*503*	Current track elapsed time

Current radio station data		
TUNER_CURRENT_DATA	*601*	Station name | Track/Program name | Album name
TUNER_CURRENT_IMG_	*602*	Base64 encoded cover bitmap
Current phone call data		
PHONE_CURRENT_DATA	*701*	Name and Surname | Phone Number | Call state (-1 = invalid, 0 unused, 1 = ringing, 2 = accepted)
PHONE_CURRENT_IMG	*702*	Base64 encoded contact image
PHONE_ELAPSED_TIME	*704*	Current call elapsed time
Current vehicle data (taken from the CAN bus)		
CAR_SPEED	*801*	Current speed
CAR_GEAR	*802*	Current gear
CAR_TORQUE	*803*	Current torque
CAR_LONG_ACC	*804*	Current longitudinal acceleration
CAR_TRANSV_ACC	*805*	Current transversal acceleration
CAR_DOOR_LOCK_STATE	*806*	DOOR_LOCK_STATE_INVALID = -1 DOOR_LOCK_STATE_UNKNOWN = 0 DOOR_LOCK_STATE_AT_LEAST_ONE_DOOR_UNLOCKED = 1 DOOR_LOCK_STATE_AT_LEAST_ONE_DOOR_LOCKED = 2 DOOR_LOCK_STATE_INTERNAL_ZV_MASTER_SECURED = 3
CAR_DRIVER_DOOR	*807*	DOOR_SWITCH_STATE_INVALID = -1 DOOR_SWITCH_STATE_UNKNOWN = 0 DOOR_SWITCH_STATE_CLOSED = 1 DOOR_SWITCH_STATE_OPENED = 2
CAR_PASSENGER_DOOR	*808*	
CAR_BEHIND_DRIVER	*809*	
CAR_BEHIND_PASSENGER	*810*	
CAR_TRUNK_DOOR_STATE	*811*	
CAR_OIL_LEVEL	*812*	INVALID = -1, MINIMUM = 0, 25CL_OVER_MINIMUM = 1
CAR_INTERNAL_TEMP	*813*	Range [0, 50] ° C
CAR_EXTERNAL_TEMP	*814*	Range [-40, 85] ° C
CAR_ENGINE_TEMP	*815*	Range [-48, 144] ° C
CAR_ENGINE_OIL_TEMP	*816*	Range [-48, 170] ° C
CAR_DATA_TIME	*817*	HH:MM DAY, DD MONTH

The WebSocket implements the communication between the Rotary (acting as a **CLIENT**) and the Infotainment (**SERVER**). It provides control messages from the Rotary to the Infotainment and informational data in the opposite direction by means of HTTP GET or POST methods. Here a list each available message that at current implementation the protocol supports (Notice: the server broadcasts updated information to the Rotary client in the same format as GET commands):

- **APP selection**: These messages have been implemented in order to
  - Notify the Rotary which application is being currently displayed on the Infotainment after a GET request

    CLIENT {"request":"GET/APP/current_active}

    SERVER {"code":200, "request", "GET/APP/current_active", "answer":"media"}
  - Notify the Infotainment that an application has been opened by acting on the Rotary screen

    CLIENT {"request":"POST/APP/selected","parameters":"nav"}

    SERVER {"code":200, "request", "POST/APP/selected", "answer":"success"}

- **SOURCE:** Informs the Rotary which source is currently active (among USB, TUNER, PHONE).

  CLIENT {"request":"GET/SOURCE/current_active}

  SERVER {"code":200, "request", "GET/SOURCE/current_active", "answer":"TUNER"}

- **MENU:**
  - Informs the Rotary that the drawer menu is active

    CLIENT {"request":"GET/MENU/current_active}

    SERVER {"code":200, "request", "GET/MENU/current_active", "answer":"media"}
  - The Rotary informs the IVI that the drawer button (on the Rotary itself) has been pressed. In this case, the IVI forwards

the command to the current active application in order to open the drawer menu.

CLIENT {“request”:”POST/MENU/open”,”parameters”:””}

SERVER {“code”:200, “request”, “POST/ MENU / open”, “answer”:”success”}

- **SHORTCUT:** The rotary informs the Infotainment that a shortcut has been used (this functionality is currently under implementation).

  CLIENT {“request”:”POST/SHORTCUT/next”,”parameters”:””}

  SERVER {“code”:200, “request”, “POST/ SHORTCUT / next”, “answer”:”success”}

- **COLOR:** Rotary requests current color combination to adjust its layout colors consistently with selected ones on the Infotainment.

  CLIENT {“request”:”GET/COLOR/current_active}

  SERVER {“code”:200, “request”, “GET/ COLOR /current_active”, “answer”:{”background_color_top”:”#AARRGGBB”, “background_color_bottom”:”#AARRGGBB”, “primary_color”:”#AARRGGBB”, “secondary_color”:”#AARRGGBB”}}

- **MEDIA:** These messages are exchanged in order to provide In-Vehicle Infotainment data repetitions.
  - Rotary requests current media being played data

    CLIENT {“request”:”GET/MEDIA/current_data}

    SERVER {“code”:200, “request”, “GET/ MEDIA/current_data”, “answer”:{“artist”:”…”, “album”:”…”, “track”:”…”, “total_time”:”milliseconds_LONG”, “time”:”milliseconds_LONG“}
  - Rotary requests current media time

    CLIENT {“request”:”GET/MEDIA/current_time }

    SERVER {“code”:200, “request”, “GET/ MEDIA/current_time”, “answer”:milliseconds_LONG}
  - Rotary requests current media cover image

    CLIENT {“request”:”GET/MEDIA/current_image }

SERVER {“code”:200, “request”, “GET/ MEDIA/current_image”, “answer”:{“file_name”:current_media_cover.png, ”file_data”:base64 encoded image}

- **TUNER:** Same as Media messages, but for TUNER data.
  - Rotary requests current tuner data

    CLIENT {“request”:”GET/TUNER/current_data}

    SERVER {“code”:200, “request”, “GET/ TUNER /current_data”, “answer”:{“rds”:”…”, “station_name”:”…”, “frequency”:”…”}
  - Rotary requests current tuner station cover

    CLIENT {“request”:”GET/ TUNER /current_image }

    SERVER {“code”:200, “request”, “GET/ TUNER /current_image”, “answer”:{“file_name”:current_tuner_cover.png, ”file_data”:base64 encoded image}

- **PHONE:** Same as Media messages, but for PHONE data.
  - Rotary requests current phone data

    CLIENT {“request”:”GET/PHONE/current_data}

    SERVER {“code”:200, “request”, “GET/ PHONE /current_data”, “answer”:{“name”:”…”, “number”:”…”, “time”:”…”, “phone_call_state”:”[accepted | ringing | declined | invalid]”}
  - Rotary requests current media time

    CLIENT {“request”:”GET/ PHONE /current_time }

    SERVER {“code”:200, “request”, “GET/ PHONE /current_time”, “answer”:milliseconds_LONG}
  - Rotary requests current call contact image (if any)

    CLIENT {“request”:”GET/ PHONE /current_image }

    SERVER {“code”:200, “request”, “GET/ PHONE /current_image”, “answer”:{“file_name”:current_phone_image.png, ”file_data”:base64 encoded image}

References

[1] Manfred Broy, Ingolf H. Kruger, Alexander Pretschner, Christian Salzmann: *Engineering Automotive Software*

[2] JohnRobert Wilson, Koji Hamamoto, Keizo ISHIMURA, Robert New: *Development Methodology: Keeping Users in Mind – UX (User Experience).*

[3] *https://www.nytimes.com/2015/09/27/business/complex-car-software-becomes-the-weak-spot-under-the-hood.html*

[4] *https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/ rethinking-car-software-and-electronics-architecture*

[5] Telematics update (November 2013). *The automotive HMI Report 2013 Extract*

[6] T.A. Dingus, S.G. Klauser, V.L. Neale, A. Petersen, S.E. Lee, J. Sudweeks, M.A. Perez, J. Hawkey, D. Ramsey, S. Gupta, C. Bucher, Z.R. Daersaph, J. Jermeland, R.R. Knipling. Virginia Tech. Transportation Institute – Sponsored by National Highway Traffic Safety Administration (2006). *100-Car Naturalistic Driving Study - Phase 11 - Results of the 100-Car Field Experiment*

[7] *http://about.att.com/newsroom/it_can_wait_expands_to_smartphone_ use_while_driving.html*

[8] National Highway Traffic Safety Administration," Department of transportation", pp. 20-21 (2012). *Visual-Manual NHTSA Driver Distraction Guidelines for In-Vehicle Electronic Devices*

[9] Ksenija Udovicic, Nenad Jovanovic, Milan Z. Bjelica. *In-Vehicle Infotainment System for Android OS: User Experience Challenges and a Proposal*

[10] ESoP 2005. *European Statement of Principles on the Design of Human Machine Interaction.*
http://www.esafetysupport.info/0C59F991-A788-45CC-B764-F0A4C38BB61E/ FinalDownload/DownloadId-FE5154B9A8ADBE16EA15F4E9E49E1473/0C59 F991-A788-45CCB764-F0A4C38BB61E/download/working_groups/esop_hmi_ statement.pdf.

[11] Nicolas Navet, RTaW Bertrand Delord, PSA Peugeot Citroën Markus Baumeister. *Virtualization in Automotive Embedded Systems : An outlook*

[12] *http://www.embedded-computing.com/embedded-computing-design/ivi-system - sandboxing-the-next-frontier-for-in-vehicle-upgrades*

[13] *http://www.archer-soft.com/en/blog/what-you-need-know-about-hmi-development*

[14] Ksenija Udovicic, Nenad Jovanovic, Milan Z. Bjelica. *In-Vehicle Infotainment System for Android OS: User Experience Challenges and a Proposal*

[15] *https://www.intel.com/content/dam/www/public/us/en/documents/brief/ automated-driving-android-v8-business-brief.pdf*

[16] from Wikipedia: In-Car infotainment

[17] *https://www.chimerarevo.com/guide/android/kernel-linux-android-140263*

[18] *https://www.qualcomm.com/snapdragon/processors/820-automotive*

[19] *https://canbuskits.com/what.php*

[20] *http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.qnxcar2.system_architecture%2Ftopic%2Fneutrino.html*

[21] *https://developer.android.com/guide/topics/providers/content-provider-basics*

[22] *https://developer.android.com/topic/libraries/data-binding/#java*

[23] Jakob Nielsen (1993), "E*xcerpt from Usability Engineering", "Response Times: The 3 Important Limits*". *http://www.nngroup.com/articles/response-times-3-important-limits/*

[24] Hp (March 2015). "*Failing to meet Mobile app user expectations*"

[25] *https://developer.android.com/studio/command-line/dumpsys*