POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree thesis

Configuration and optimization of the operating system running on the infotainment ECU

Supervisor:

Prof. Massimo VIOLANTE

Company Tutor:

Ing. Daniele COSTARELLA

Candidate:

Alessandro MIGALE

Academic Year 2023/2024

Abstract

This thesis investigates boot performance optimization for the Android Auto 14 operating system. The rapid advancement of automotive technology, particularly in infotainment systems, requires efficient and reliable performance to improve user experience and safety. This study focuses on achieving a fast start-up, which is critical to minimizing delays and improving overall system efficiency.

The work is divided into several phases: initially, the architecture of infotainment systems and the integration of Android Auto within the Android ecosystem are analyzed. Next, optimization methodologies are explored, with a specific focus on startup time optimization. A practical case study is presented, detailing the configuration environment using Raspberry Pi 4 and Android 14, based on the existing "raspberry-vanilla" project on GitHub. Detailed analysis of logs using tools such as Bootchart and Android studio's Logcat identified key performance indicators (KPIs) related to boot performance.

The optimization phase initially involved removing non-essential projects within the operating system, such as the startup animation file, resulting in significant improvements in startup performance. A new kernel was then configured within the project, with an improved and optimized version. The use of a bootgraph.pl script allowed the identification of bottlenecks system initialization, leading to a net improvement in performance thanks to the disabling of serial console. A critical aspect of the optimization process was the detailed study of Android kernel configurations, analyzing active and inactive settings to understand their impact on system performance and boot time. Changes based on this analysis have resulted in significant efficiency gains and reduced boot times.

Additionally, the study included an evaluation of the kernel's compression and decompression algorithms. Various algorithms were compared to assess their impact on boot time and system responsiveness. This comprehensive analysis provided insight into the selection of optimal compression techniques, but the limitations of the current design made available a single compression algorithm that was not optimal from the point of view of compression speed. Research shows that targeted optimization efforts, including component removal, kernel optimization, and compression algorithm selection, can substantially improve Android Auto boot performance on Raspberry Pi 4. The results indicate that a well-optimized system not only boots faster, but also works more efficiently, providing a smoother user experience.

Future research could explore further optimisation techniques and their application to other components of automotive infotainment systems, further improving overall system performance and user satisfaction.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Infotainment System

In recent years, Infotainment systems have become a central component in the automotive industry, evolving rapidly to satisfy the increasing demands for connectivity, entertainment, and safety. Initially designed to provide basic features such as radio and navigation, these systems have now developed into advanced platforms that integrate GPS navigation, communication, multimedia entertainment, internet connectivity, and vehicle control functions.
Infotainment is a combination of "Entertainment" and "Information", referring to a system that delivers both informative content and entertainment services.
In this context, Android has become one of the most popular operating systems due to its flexibility and extensive customizability. Specifically, Android Auto marks a considerable progression by offering a driving-optimized user interface that integrates navigation, communication and entertainment features.
Currently, the landscape of Infotainment systems is undergoing a profound transformation, marked by the increasing integration of artificial intelligence (AI) and voice recognition technology. AI is capable of analyzing vast quantities of data in real time, offering drivers precise information and recommendations. Voice recognition technology allows drivers to interact with the infotainment system using natural voice commands. This not only simplifies and enhances the safety of using car features while driving but also allows drivers to focus more on their surroundings. Simultaneously, it is crucial to guarantee the transparency of artificial intelligence in decision-making processes and to assume responsibility for any unintended consequences that may result from its use [3].
The integration of cameras within vehicles has greatly increased, enhancing facial recognition features through advanced processing technology. These cameras are now capable of monitoring driver behavior and identifying potential issues.

Gesture recognition technology has revolutionized driver-vehicle interaction. It enables the use of simple, standardized movements to switch tracks or adjust various car settings, thereby significantly minimizing distractions while driving [1].
Moreover, Cybersecurity is gaining prominence, particularly in light of the expanding data-sharing capabilities that could potentially endanger driver safety [3].

1.1.1 Evolution of System

Today's Infotainment systems have reached a level of sophistication that was not always present. Their evolution marks a journey through various technological eras, radically altering the driving experience. Within this evolution, distinct stages of development can be identified [14]:

- **The 1930s:** AM radio was first installed in a Chevrolet car in 1922, although this was an experimental setup and not commercially produced. The first significant commercial success of car radios occurred in 1930 when Paul and Joseph Galvin of the Galvin Manufacturing Corporation introduced the Motorola 5T71 [20]. Initially, due to their large size, these radios were often placed in the luggage compartments and transmitted data through radio waves. However, they frequently suffered from static interference, which affected the sound quality.

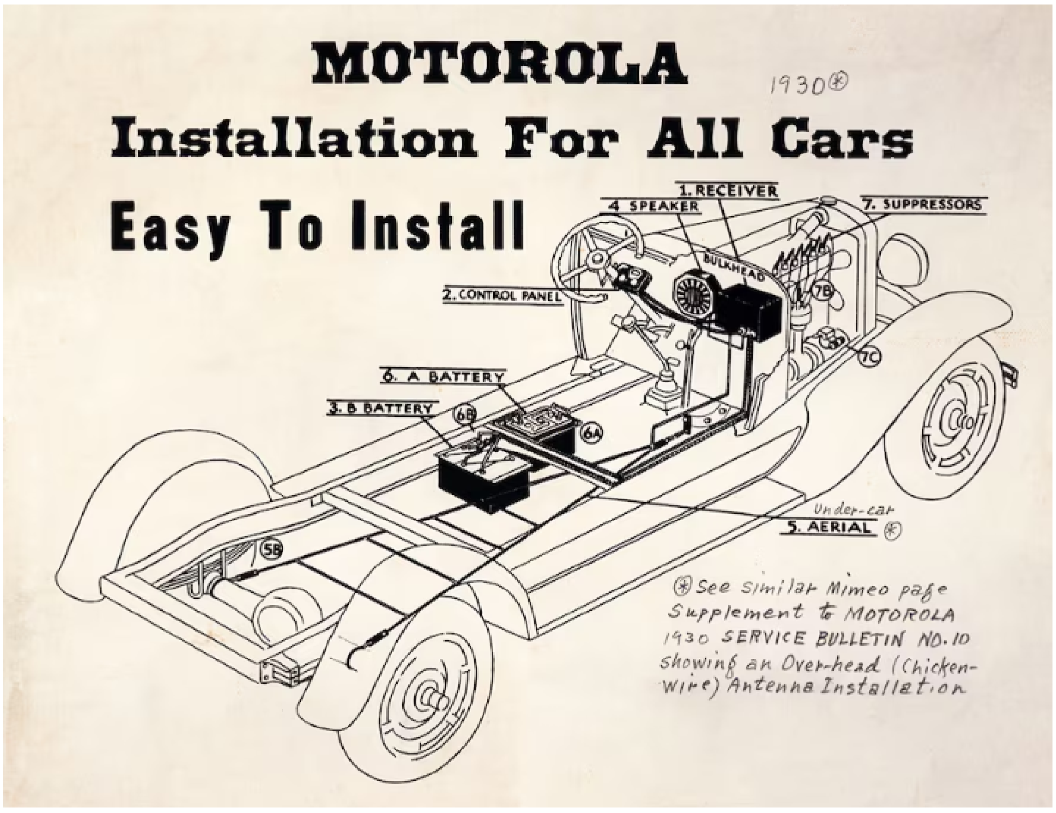

MOTOROLA
1930 ⊛
Installation For All Cars
Easy To Install
1. RECEIVER
4 SPEAKER
7. SUPPRESSORS
BULKHEAD
7B
2. CONTROL PANEL
7C
6. A BATTERY
3. B BATTERY
6B
6A
Under-car
5. AERIAL ⊛
5B
⊛ See similar Mimeo page Supplement to MOTOROLA 1930 SERVICE BULLETIN NO. 10 showing an Over-head (Chicken-wire) Antenna Installation

Figure 1.1. A Motorola car radio installation diagram (circa 1930) [20].

- **The 1950s:** Over the years, in-car radios have undergone a significant evolution, incorporating physical buttons for saving favorite stations and rotary knobs for navigating through frequency bands. Mechanically, these devices represented the primary method of data storage in vehicles.

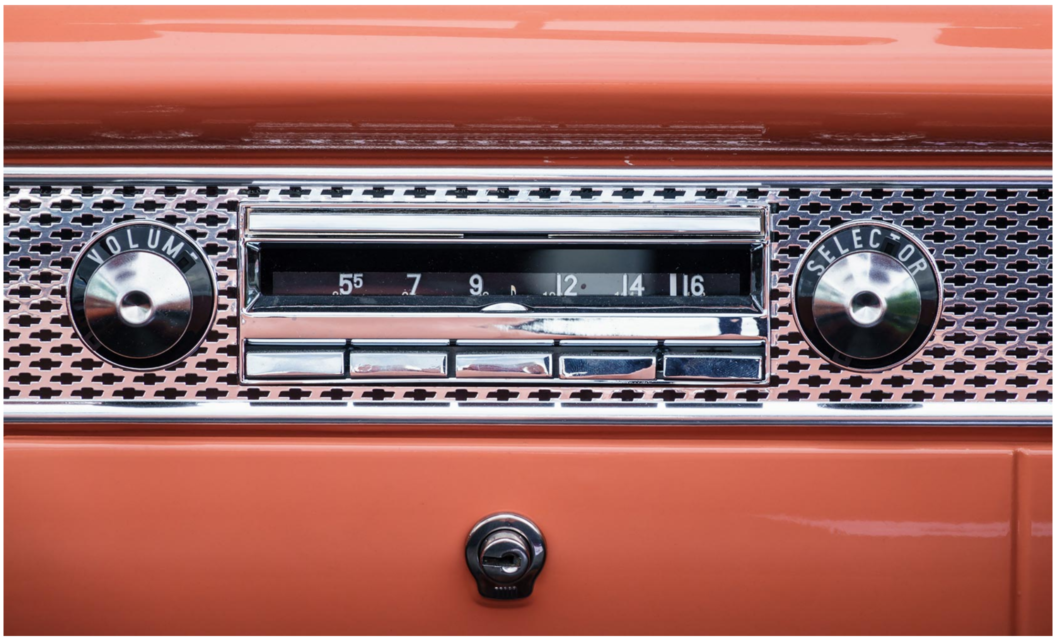


Figure 1.2. Preset radio station buttons in 1950s [2].

- **The 1960s-'80s:** In these years, the advent of compact cassettes marked the beginning of a diversified musical period. With the introduction of cassette players in automobiles, these recordings became essential for motorists, thanks in part to their convenience and ease of use.

Figure 1.3. Car Cassette tape deck,1980s [2].

In that same period, in 1981, Honda introduced the Honda Electro Gyrocator, the world's first map-based automotive navigation system. Developed in collaboration with Alpine, a renowned electronics manufacturer, and Stanley Electric, a leader in automotive components, this system diverged from GPS technology as it didn't rely on satellites for positional determination. Instead, it employed a helium gas gyroscope to detect the car's direction, while a custom servo gear attached to the transmission housing calculated the distance traveled by the vehicle. By leveraging data from direction and driving distance sensors, the equipment could estimate the vehicle's position [16]. Their expertise and collaboration were instrumental in the development of this groundbreaking navigation system, highlighting the importance of industry partnerships in driving technological advancements.

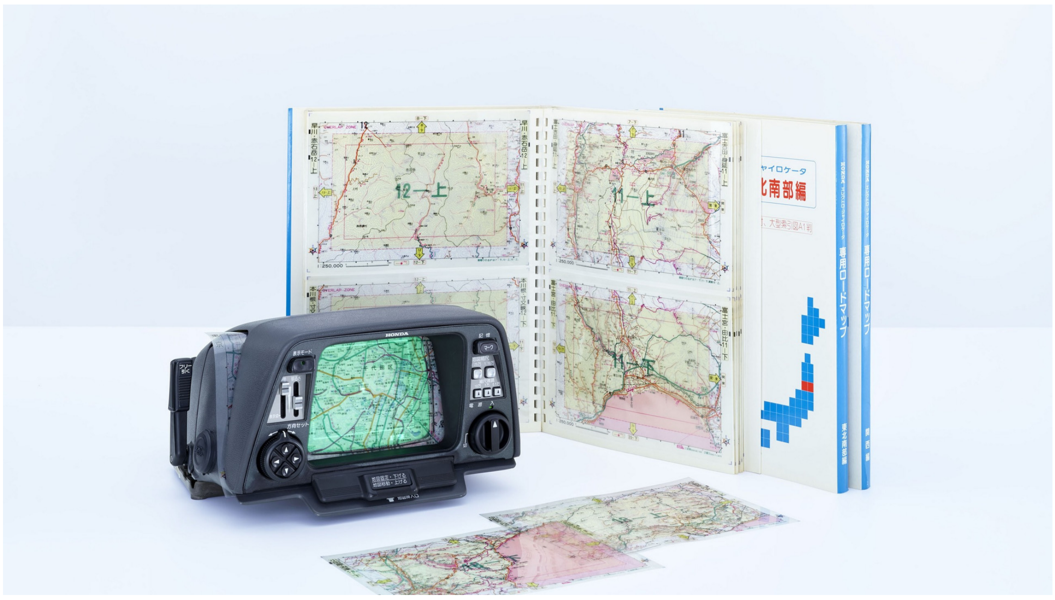


Figure 1.4. Honda Electro Gyrocator map-based navigation system [16].

- **The 1990s:** The introduction of CDs has been a central moment in the evolution of car infotainment systems, providing enhanced portability and audio quality over cassettes. During this time, there was witnessed the proliferation of MP3 units and the debut of screens in these systems, accompanied by the integration of Bluetooth functionality and auxiliary inputs, paving the way for new in-car entertainment possibilities.

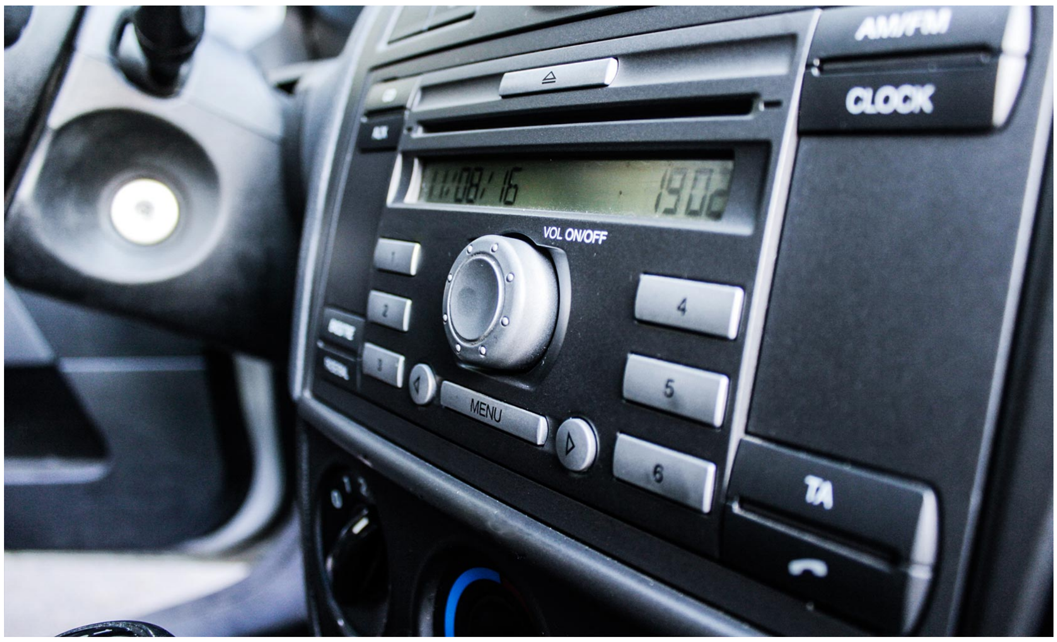


Figure 1.5. Car CD player [2].

- **The 2000s:** CD players quickly became obsolete, overtaken by more sophisticated technologies like MP3 players and USB storage devices for entertainment purposes. The advent of Bluetooth connectivity marked a significant revolution in the industry. In 2007, Ford Sync Technology was introduced, marking the debut of the first voice-controlled infotainment system, which allowed drivers to make hands-free calls and control music with voice commands.

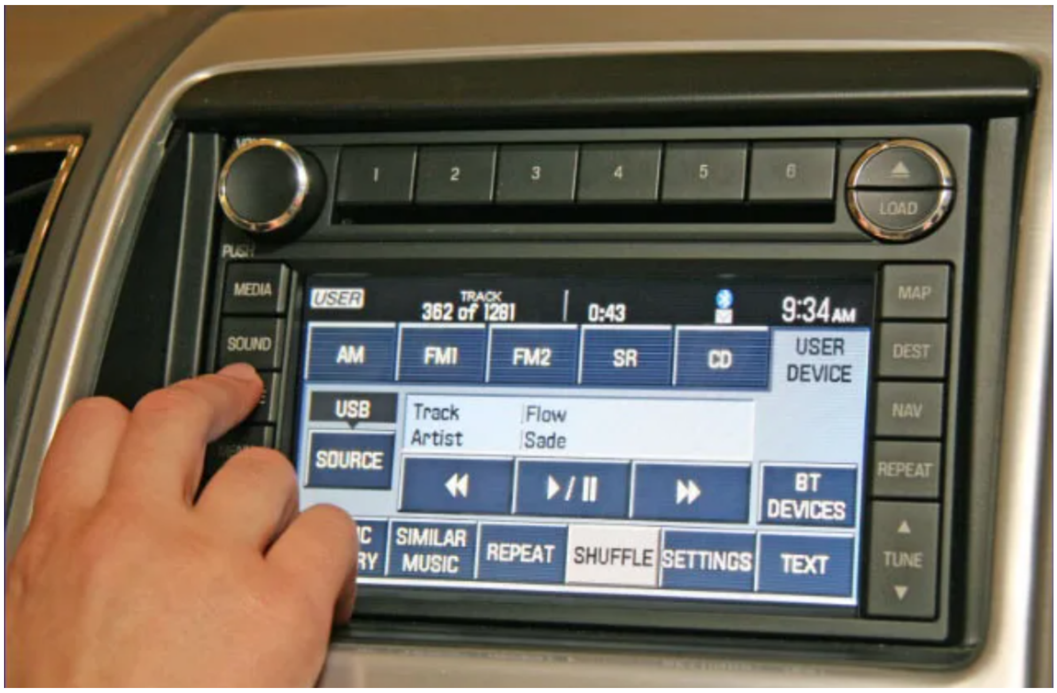


Figure 1.6. Ford Sync, 2007 [26].

- **The 2010s:** In recent years, GPS navigation has gained popularity, supported by phone connectivity with Google Android Auto and Apple Car Play, and the introduction of several apps and in-car Wi-Fi. Physical buttons have gradually been replaced by larger, fuller touchscreens that are compatible with phone connectivity. Internet connectivity has facilitated the use of satellite navigation, while advanced electronics and microchips have made it possible to incorporate flash drives to support internal storage of apps, maps, and other information. Today, you can connect your phone directly to your car's infotainment system, giving you access to features like music playback and hands-free calling.

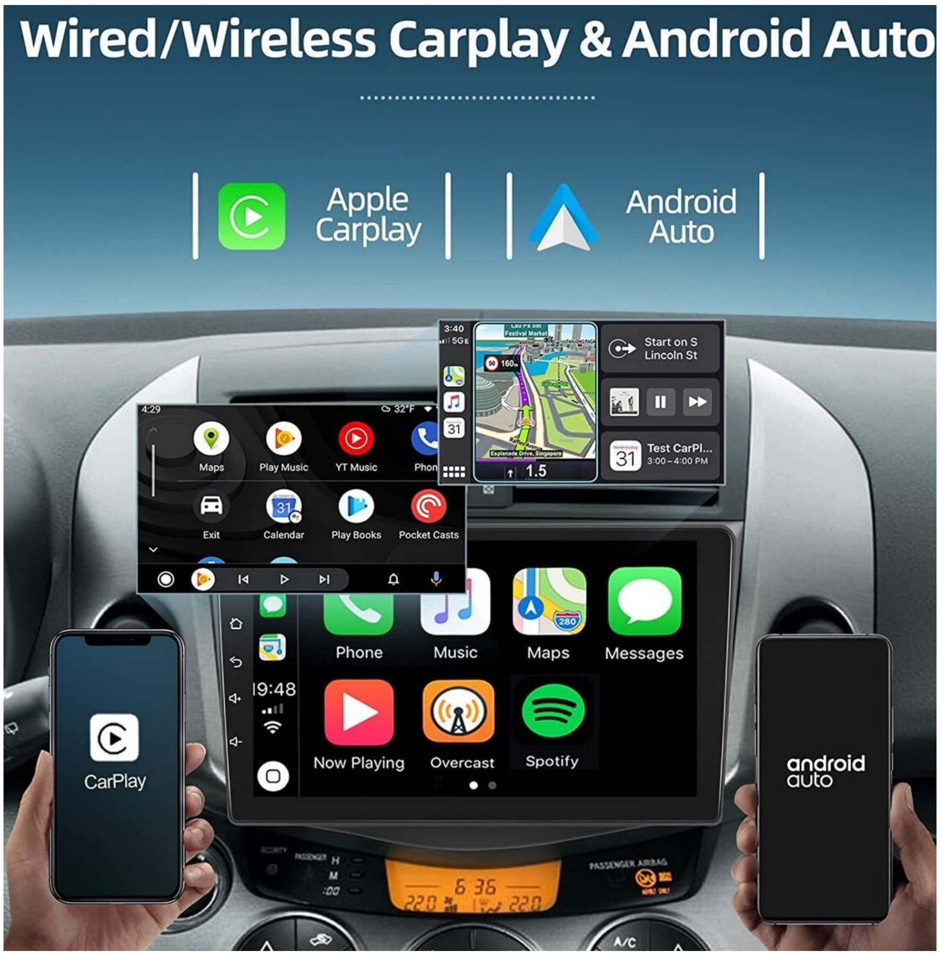


Figure 1.7. Carplay & Android Auto.

- **The 2020s:** Over recent decades, automotive Infotainment systems have seen a significant evolution in technology, providing an array of features to improve the driving experience. These include expansive touch screens with geo-fencing, intelligent 3D navigation offering traffic updates and alternative routes, and the ability to control the vehicle through voice commands. Additionally, sensors, cameras, and radar deliver essential real-time data, like tire pressure monitoring and parking assistance, making driving more effortless. With continuous technological progress, it's expected that car manufacturers will further incorporate advanced features such as artificial intelligence and autonomous driving assistance systems, leading to a future where vehicles may communicate with each other and operate autonomously.

Figure 1.8. ChatGPT integrated into DS Automobiles [12].

1.1.2 Structure of the System

The on-board Infotainment system integrated within automobiles plays a central role in the modern driving experience, providing an extensive array of functions designed to meet the needs of both drivers and passengers. Through the integration of elements from the vehicle's interior and exterior, it serves as a centralized control center for accessing and managing a diverse range of services and features, thereby enhancing convenience and safety while on the road.
The architecture of the infotainment system can vary significantly depending on the vehicle type and manufacturer, with different components and functionalities reflecting the priorities and distinctive characteristics of various automotive brands.
Let's explore the essential components that are present in the system and their function in providing an optimal driving experience [13]:

- **Integrated Head-Unit:** It is a tablet-like device with a touchscreen interface, mounted on the vehicle's dashboard. This central unit acts as well as the control hub for the entire system, managing car settings, audio system and navigation information.

- **Heads-Up Display:** A display that shows real-time information, like climate, speed or navigation maps, on a transparent screen integrated into the vehicle's windshield. Its purpose is to provide essential driving data instantly, allowing drivers to maintain focus on the road without needing to shift attention to the instrument panel. This minimizes the likelihood of accidents and enables a less stressful driving experience.

- **Processing Power: DSPs and GPUs** Modern systems utilize sophisticated digital signal processors (DSPs) and graphics processing units (GPUs). DSPs enhance multimedia quality by processing audio and video signals, while GPUs handle graphic operations, ensuring images and videos are rendered with exceptional clarity and detail. Their robust capabilities ensure seamless graphics and multimedia performance, providing an optimal user experience.

- **Operating Systems:** Operating systems are crucial for a system as they support connectivity, provide useful features, and allow the integration of new functionalities through downloadable software applications. They constitute the digital infrastructure on which the entire Infotainment system relies.
- **CAN and other network protocol support:** Electronic hardware components within the system are interconnected using standard communication protocols like CAN and LVDS. This enables devices and sensors to exchange information in real-time, eliminating the need for a central computer to manage communications. As a result, system efficiency is enhanced and complexity is minimized.
- **Connectivity Modules:** The in-car Infotainment systems are equipped with GPS, Wi-Fi and Bluetooth modules to allow connection to external networks and devices, enabling features such as navigation, internet access and smartphone integration.
- **Automotive Sensor Integration:** Numerous sensors, including proximity sensors, along with cameras, are incorporated into the systems to deliver crucial information regarding the safety of both the driver and passengers.
- **Digital Instrument cluster:** These clusters exhibit vehicle data like speed and additional details from the vehicle's ECU through the OBD-II port. Together with the main unit and heads-up display, they constitute the vehicle's digital interface.

The architecture of the IVI (In-Vehicle Infotainment) system is illustrated in Figure 1.9. This architecture consists of five distinct layers, each responsible for different functionalities of the system. Each layer interacts with the layers above and below it, facilitating efficient communication and operation across the entire system. The main layers of the system are as follows [17]:

- **Hardware Layer** includes:
  1. **CPU (Central Processing Unit):** The primary processor that executes instructions and manages tasks.
  2. **Memory:** Provides temporary storage for data being processed by CPU.

3. **CAN (Controller Area Network):** A robust vehicle bus standard that facilitates communication between different electronic components within the vehicle.
4. **Bootloader:** A small program that initializes the hardware and loads the operating system when the system is powered on.

- **OS Layer** includes:
  1. **BSP (Board Support Package):** Software providing the essential support required for running a specific operating system on particular hardware.
  2. **OS Core:** The central part of the operating system that manages system resources and hardware.
- **Middleware Layer:** Acts as the intermediary between the operating system and the application layers, providing crucial services including:
  1. **Media & Graphics:** Manages multimedia playback and graphic rendering, ensuring high-quality audio and visual output.
  2. **Platform Management:** Coordinates and manages various software components and services running on the platform.
  3. **System Infrastructure:** Provides essential services and support for the overall system operation.
  4. **Automotive Connectivity:** Manages communication and connectivity between the IVI system and other vehicle systems.
- **Application Layer:** Includes user-facing applications providing various functionalities to enhance the driving experience, such as:
  1. **Entertainment:** Applications for playing music, videos, and other media content.
  2. **Mobile Office:** Tools for managing emails, calendars, and other productivity tasks.
  3. **Networking:** Manages network connections, allowing the IVI system to connect to the internet and other devices.
  4. **Navigation:** GPS-based applications providing route guidance and real-time traffic information.
- **HMI Layer:** The highest layer, responsible for user interaction, includes:

1. **Speech:** Voice recognition and synthesis capabilities that enable users to interact with the system using voice commands.
2. **User Interface:** Graphical and touch interfaces through which users interact with the system.
3. **HMI Core:** The essential components that support the HMI functionalities.

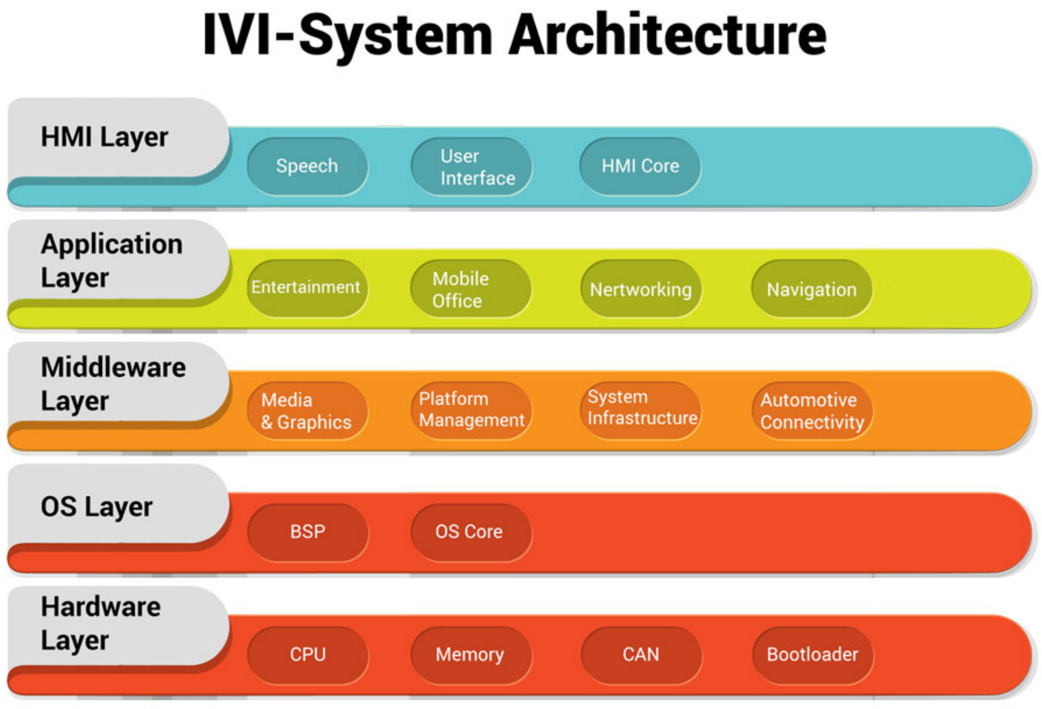


Figure 1.9. IVI-System Architecture [13].

For a more detailed view of the hardware components and their interconnections, the following block diagram in Figure 1.10 provides a comprehensive overview of the various modules and interfaces involved in the infotainment system. The block diagram illustrates the detailed connections and interactions between the processor, various communication modules (e.g., CAN, LIN), and interfaces (e.g., HDMI, USB, Bluetooth). It highlights the role of the processor in managing the flow of data between different components, including the touch screen, audio system, and connectivity modules. A thorough examination of the layered architecture and the detailed block diagram reveals the intricate complexity and integration necessary to provide a contemporary in-vehicle infotainment experience.

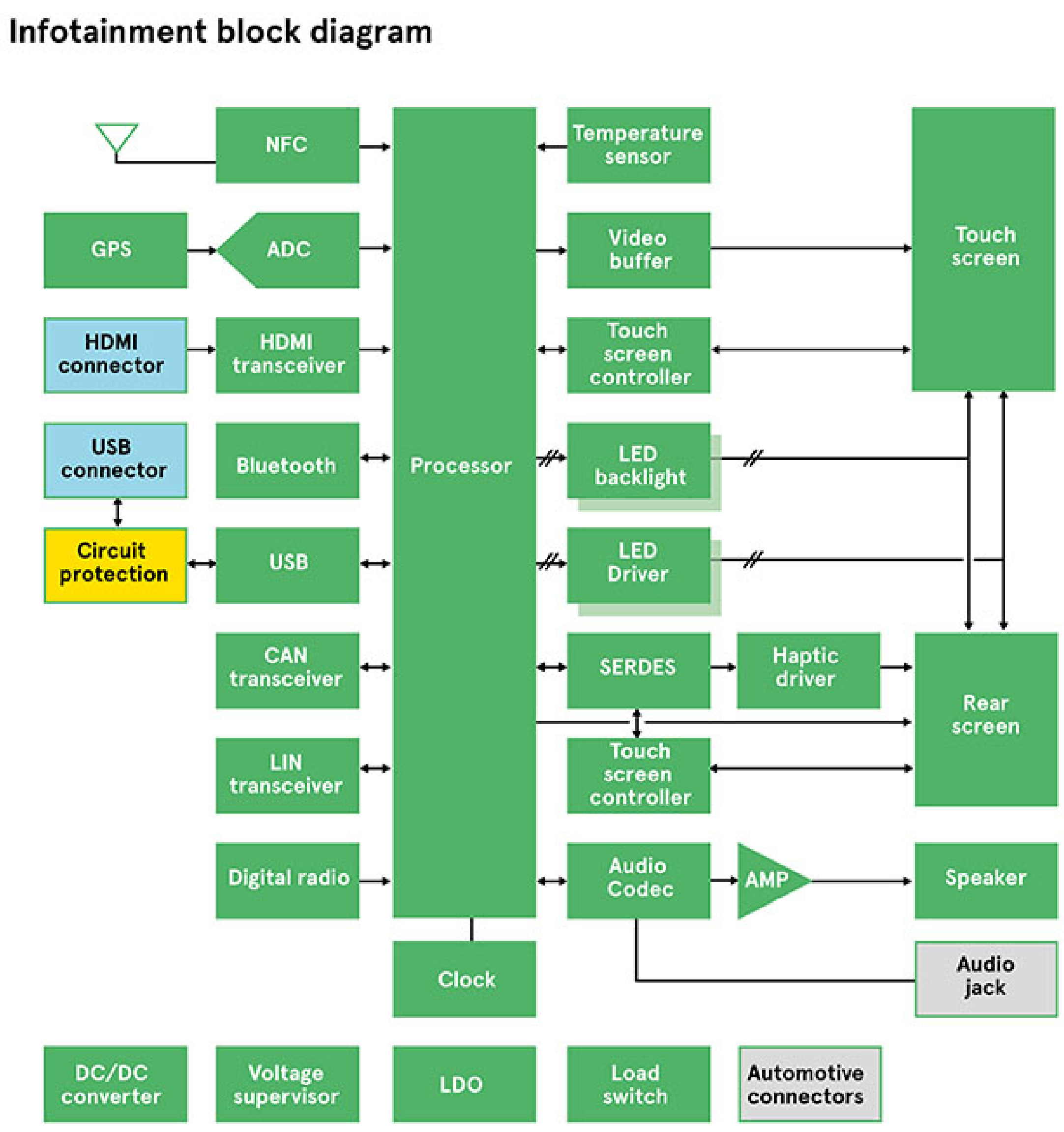


Figure 1.10. Block diagram of a typical IVI system [11].

1.2 Android

The Android operating system holds the largest user base among mobile platforms worldwide. As of the end of 2021, Android commanded 71% of the global market share, a figure that continues to rise.
Initially conceived by the Open Handset Alliance, Android is constructed upon a modified Linux kernel and other open-source software components. Google provided early backing for the project and acquired the company in 2005, subsequently introducing the inaugural Android device in 2008.
Android maintains its dominance in the operating system market due to its extensive features, user-friendly interface, strong community support, and vast customization capabilities. While originally designed for mobile devices, Android has diversified its

applications, supported by advancements in code libraries and widespread developer adoption across various sectors, thus evolving into a versatile software suite for tablets, smart TVs, and notebooks [18].
Android was developed as part of the Android Open Source Project (AOSP), an initiative that made Android's source code publicly available for anyone to use, modify, and distribute, thus contributing to its adoption across a wide range of devices and industries. However, it's important to note that despite the open-source nature of Android, some key components and services, such as the Google Play Store (the official app store for certified devices running on the Android) and some system applications, are proprietary and subject to Google's policies and restrictions.

1.2.1 Linux

Linux, developed by Linus Torvalds in 1991, is a collection of open-source Unix-like operating systems. It is available through various distributions, such as Debian and Ubuntu, which are among the most well-known. These distributions package the Linux kernel alongside various software applications, system libraries, and a graphical user interface, providing a comprehensive operating system that is ready for immediate use. Linux is primarily programmed in C and assembly language and operates on a monolithic kernel (in which the operating system runs in the kernel space, allowing for efficient process management and system calls). Linux distributions serve a wide array of systems, including cloud computing, embedded systems, mobile devices, personal computers, servers, mainframes, and supercomputers. Linux provides a variety of advantages that make it an ideal option for diverse user groups and application contexts:

- **Open Source:** Linux is based on an open-source development philosophy, which means that the source code is freely available for anyone to study, modify, or distribute. This approach encourages collective innovation and ensures transparency within the software development process. Contributions from a global community of developers help in rapidly addressing bugs and introducing new features.
- **Reliability and Stability:** Linux is renowned for its reliability and stability, thanks to its robust architecture and modular design. Linux systems can operate for extended periods without the need for reboots or extensive maintenance, which makes them perfectly suited for critical and mission-critical applications.
- **Flexibility and Customization:** Linux provides a variety of distributions, each customizable to meet individual user needs. This adaptability enables the

creation of work environments tailored for different purposes, ranging from personal desktop use to highly scalable server setups. Users have the ability to adjust nearly every element of the system to enhance performance and usability based on their individual needs.

- **Security:** The open-source nature of Linux and its extensive developer community contribute to thorough code reviews and prompt resolution of vulnerabilities. Moreover, the Linux kernel incorporates security features like SELinux and AppArmor, which fortify the system against both external and internal threats. These tools provide mandatory access controls, ensuring that even if a system component is compromised, the damage can be contained.
- **Performance:** Linux is known for its high performance and efficient management of system resources. Its optimized design and ability to run on a wide range of hardware allow for optimal performance even on resource-constrained devices. Linux's minimal overhead and effective memory management render it a superior option for performance-critical environments.
- **Compatibility:** Due to its flexibility and modular nature, Linux can support a wide range of devices and hardware architectures, from embedded systems and mobile devices to desktop workstations and supercomputers.
- **Active Community and Support:** Linux is supported by a large and active community of users and developers distributed around the world. This community provides extensive technical support, educational resources, and a rich variety of software and tools that are available for free.
- **Interoperability:** Linux's support for open standards and protocols enhances its ability to seamlessly communicate with other systems and devices. This is particularly beneficial in the development of integrated infotainment systems, for instance, which can effortlessly interact with smartphones and other external components.

In conclusion, the choice to use Linux as the basis for Android development has proven to be a winning decision, allowing for the creation of a flexible, stable, and feature-rich mobile operating system that has had a significant impact on the world's technology landscape. The foundational attributes of Linux, including its open-source model, robustness, and adaptability, have been key to Android's dominance in the mobile market. This integration of Linux into Android underscores the vital role Linux plays in contemporary computing and its significant influence on future technological developments.

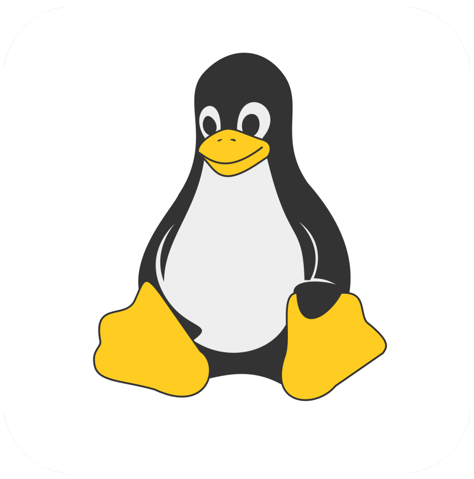


Figure 1.11. Linux icon.

1.2.2 Android Architecture

Android is a set of open-source software based on Linux, designed to be compatible with different devices and formats. Within the Android structure we can distinguish several key components [4]:

- **Linux Kernel:** The Linux kernel serves as the core of the Android architecture, playing a central role in managing all necessary drivers, including those for the display or memory, during runtime.
  It incorporates various security features to safeguard the Android system, such as process isolation and user-based permissions. Additionally, the Linux kernel includes power management features to optimize power consumption, such as adjusting the CPU frequency or suspending the device during periods of inactivity. Acting as an abstraction layer between the device's hardware and the other components of the Android system, it ensures a unified and reliable interface.
  This kernel is open and editable by the community, allowing for customizations and enhancements designed to the specific needs of Android devices. As already mentioned previously, this flexible approach is reflected in the distribution of Android under a combination of open-source and proprietary licenses, highlighting the complexity of its structure and distribution.

- **Hardware abstraction layer (HAL):** It serves as a fundamental element in the Android system's architecture, acting as an intermediary between hardware-specific drivers and the higher-level Java API framework used by applications on the device. It consists of various modules, each aligning with a specific hardware component such as the camera or audio. Whenever an application requests hardware access via the Java API framework, the Android system initiates the relevant module for that hardware component, ensuring efficient and reliable access. Despite hardware-specific implementations, each HAL module exposes a standardized interface to the Android framework. This standardization ensures that the Android operating system can function consistently across different hardware configurations [7].

- **Runtime Android:** The Android Runtime (ART) is the environment where applications operate on Android devices. It executes application code, manages memory, and handles all necessary operations for apps to function properly. ART uses **ahead-of-time (AOT)** compilation to compile application code at the time of installation rather than during execution. This approach enhances app performance and optimizes resource management. ART also features an optimized garbage collection system, which efficiently reclaims memory space no longer in use, minimizing pauses and improving app responsiveness. Additionally, ART provides advanced debugging tools, including detailed diagnostic exceptions and crash reporting [21].

- **Native C/C++ libraries:** They are essential components for the operating system and applications functionality. These libraries, such as **libc** (standard C library providing essential system functions) and **SQLite** (database library used for data storage), support central Android system components like the Android Runtime (ART) and Hardware Abstraction Layer (HAL), ensuring efficient performance and compatibility across a wide range of devices.

- **Java API framework:** Java-language APIs are crucial in Android app development as they enable the reuse of fundamental system components and services. These APIs provide essential building blocks, such as [21]:
  - A flexible **view system** used to create app's user interfaces, including elements such as grids or text boxes.
  - A **resource manager** that grants access to non-code assets like layout files.
  - A **notification manager** that allows apps to display custom alerts in the status bar.
  - An **activity manager** that manages the lifecycle of applications.
  - **Content providers** that permit apps to retrieve data from other applications or to distribute their own data.

- **System apps:** In the system, several apps are integrated for managing emails, SMS, contacts, and other essential functionalities. Users have the flexibility to replace these apps with third-party alternatives, except for some like the system Settings app, which holds a privileged status.

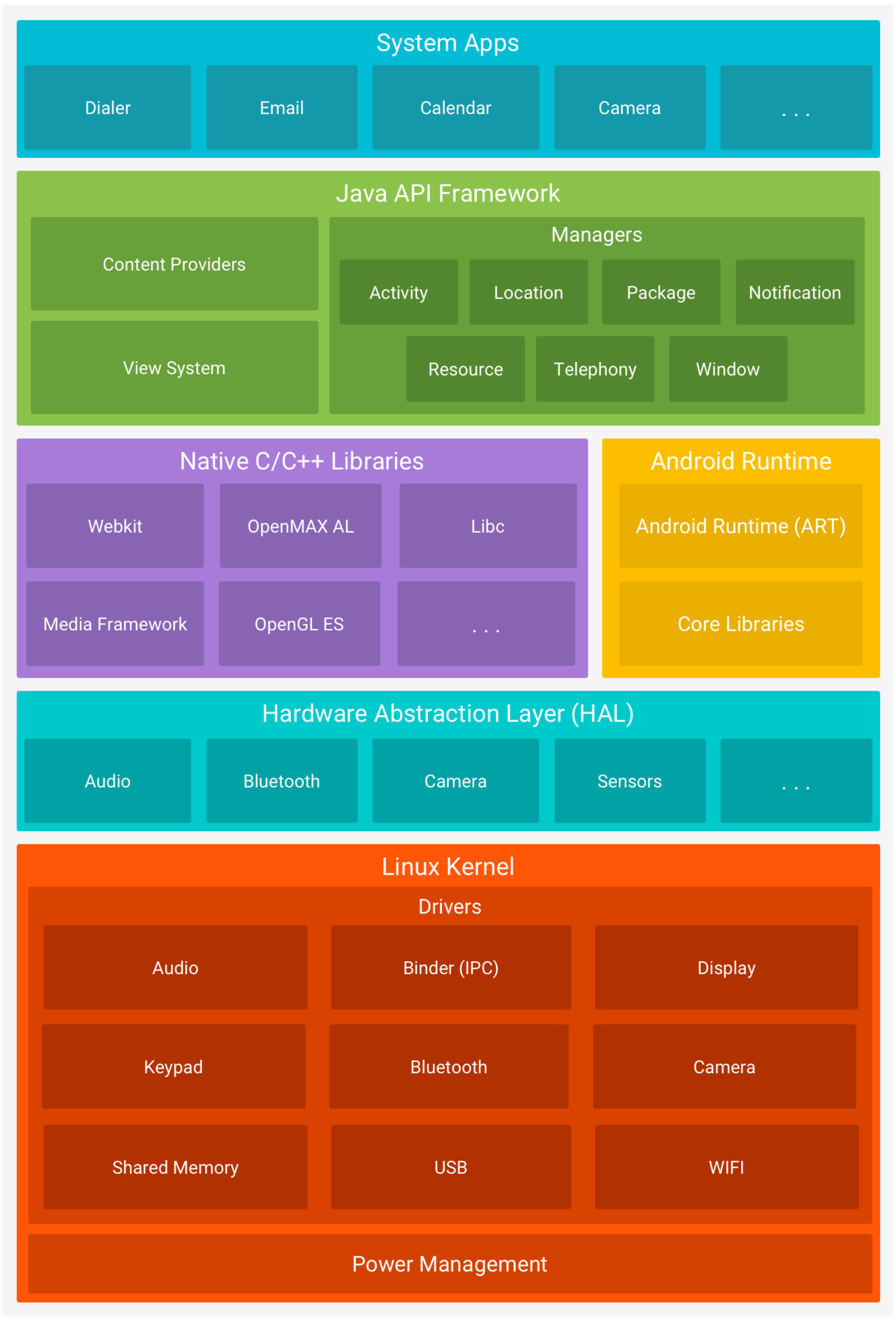


Figure 1.12. The Android software stack [21].

1.2.3 Android Auto

Launched in 2015 by Google, Android Auto represents a step forward in automotive technology, aimed at improving safety and convenience for drivers. Integrating smartphones with the vehicle's infotainment system, Android Auto offers a range of

features designed to minimize distractions while on the road.
Android Auto allows users to connect their smartphones to the car's display using either a USB cable or wirelessly (on compatible vehicles with Android 11 and above). Once connected, the smartphone's applications and features are reflected on the car's display, offering convenient access to navigation, music, messaging, and additional services. Additionally, Android Auto integrates seamlessly with Google Assistant, enabling hands-free operation for tasks such as making calls, sending messages, and controlling various functions through voice commands.
It's important to note that while Android Auto offers access to a variety of apps, including navigation and music streaming services, certain features, such as direct access to the Google Play Store, are not available. Instead, users need to install compatible apps on their smartphones, which can then be accessed and controlled through the car's interface.
In recent years, Android Auto has undergone significant enhancements and updates, further enhancing its functionality and user experience. Features like predictive navigation, personalized recommendations, and expanded app support continue to make Android Auto a valuable companion for drivers. Overall, Android Auto represents a revolutionary change in automotive technology, offering a safer, more convenient, and interconnected driving experience. As automotive technology evolves, Android Auto continues to lead the industry, driving innovation and shaping the future of in-car Infotainment systems.

1.3 Thesis objective

The primary objective of this thesis is to enhance the boot performance of an Android Auto 14 Operating System deployed on the Raspberry Pi 4. The startup speed plays a pivotal role in the overall performance of an Infotainment system; efficient boot-up is essential to ensure users can promptly access system features without encountering delays or interruptions. Moreover, a quick startup can contribute significantly to the vehicle's overall energy conservation by minimizing the duration the system spends in boot mode.
Selecting the Raspberry Pi 4 as the hardware platform for the Android Open Source Project (AOSP) provides a cost-efficient development environment with robust processing capabilities and extensive support from a vibrant online community of developers. Leveraging the "raspberry-vanilla" project on GitHub as a foundation, this study involves meticulous log analysis using tools such as Android Studio's Logcat, Bootchart, and script bootgraph.pl. These tools enable comprehensive assessment of system startup performance and identification of Key Performance Indicators (KPIs) crucial for measuring progress and success.

This thesis aims to demonstrate that targeted optimization efforts can yield substantial improvements in system performance during startup, showcasing practical methodologies and insights applicable to automotive infotainment systems.

Chapter 2

Optimization of the Infotainment system

2.1 Concept of optimization

Optimizing embedded systems and, more specifically, Infotainment systems is a strategic process aimed at enhancing the overall performance, stability and user satisfaction. It involves meticulously refining different components to achieve an ideal balance of system efficiency, resource consumption, and responsiveness. This optimization is not only about making the system faster but also about ensuring it operates smoothly under varying conditions and workloads.
The concept of optimization is based on three fundamental pillars that contribute to improving end-user satisfaction:

- **Performance Enhancement:** This aspect involves increasing system speed and responsiveness through software optimization and fully utilizing hardware capabilities. By optimizing algorithms and workflows, Infotainment systems can deliver smoother interactions and faster response times, thereby enhancing user productivity and satisfaction.

- **Resource Utilization:** Optimizing system resources requires efficient management of CPU, memory, and storage to ensure their optimal use. It's about striking a balance between performance and resource conservation. Through meticulous management of resource allocation and utilization, systems can function efficiently, avoiding undue strain on hardware components and consequently prolonging their service life.

- **Stability Improvement:** Ensuring robust stability in different hardware and software environments is essential for optimizing Infotainment systems. Concentrating on compatibility and reliability allows these systems to deliver consistent performance and user experience across various platforms and usage scenarios. This approach includes thorough testing, debugging, and compatibility assessments to reduce errors and guarantee smooth functionality.

Optimization is indispensable for ensuring Infotainment systems operate efficiently, reliably, and satisfactorily over their lifecycle. By delivering prompt responses to user demands while maintaining stability and longevity, optimized systems significantly elevate user experience and overall system performance.

2.2 Optimization methodologies

Optimizing Infotainment systems involves various methodologies that enhance system performance and efficiency. These methodologies are crucial for providing a seamless and responsive user experience throughout the system's operation.

- **Boot Time Optimization:** This aspect focuses on minimizing the time required for the system to boot up and become operational. It involves eliminating unnecessary boot steps that do not contribute to the core functionality of the system. Additionally, optimizing Bootloader settings and executing initialization tasks in parallel are critical strategies to leverage the system's multitasking capabilities effectively. By reducing boot time, users experience quicker access to essential functions upon starting the vehicle, enhancing convenience and usability.

- **Run time Optimization:** During normal operation, run time optimization aims to maximize system performance. This includes optimizing software algorithms to ensure efficient data processing, which can significantly impact the responsiveness and speed of applications within the Infotainment system. Allocating resources such as computing power and memory optimally helps in achieving smoother execution of tasks. Furthermore, implementing real-time task scheduling techniques minimizes delays and ensures that critical functions operate seamlessly, thereby enhancing the overall user experience.

- **Memory Optimization:** Efficient management of system memory (RAM) is crucial for maintaining optimal performance over time. This involves identifying and resolving memory leaks that can gradually degrade system performance. Optimizing data structures and reducing unnecessary memory usage further contribute to enhancing the efficiency of memory management. By ensuring efficient memory utilization, the Infotainment system can maintain responsiveness and stability under varying workloads.

- **Power consumption Optimization:** Reducing energy consumption is another critical aspect of optimizing Infotainment systems, particularly in vehicles where efficient power management is essential. This optimization focuses on refining components that consume significant amounts of power, such as displays and processors. Implementing smart energy management strategies, including optimizing background processes and controlling power usage during idle periods, helps extend battery life and minimize overall energy consumption.

- **Network Optimization:** Efficient communication with external devices and networks is vital for enhancing the connectivity and functionality of Infotainment systems. Network optimization strategies include optimizing data transmission protocols to minimize latency and maximize throughput. Enhancing connectivity reliability ensures seamless integration with external devices, such as smartphones and external sensors, enabling advanced features like real-time navigation updates and multimedia streaming.

These methodologies are integral in providing an efficient and responsive driving experience, which is vital for fulfilling the expectations of contemporary vehicle users. Through the optimization of boot time, runtime performance, memory usage, power consumption, and network connectivity, Infotainment systems can offer improved functionality and dependability, thus elevating user satisfaction and safety.

2.3 Boot Time Optimization

As previously emphasized, optimizing boot time is crucial for Infotainment systems. The speed at which the system becomes operational is critical to ensuring a seamless and immediate user experience. Decreasing startup times maximizes efficiency and user satisfaction, enabling the system to be ready for use in the shortest time possible. Considering this optimization, selecting the right key performance indicators (KPIs) is crucial to ensure a focused and strategic approach. KPIs help in setting clear goals and tracking progress effectively. There are several key aspects within the optimization process:

- **Measuring Boot Time:** Precise measurement of startup time is essential for identifying improvement opportunities. Defining the specific events that mark the beginning and conclusion of the startup process is critical for accurate performance tracking. Utilizing performance analysis software and advanced technology is vital for ensuring precise and reliable measurements.

- **Select Reference Events:** Baseline events should be selected according to the objectives set. For instance, displaying a boot screen or animation can serve as indicators that the system is operational and ready for subsequent interactions. Alternatively, playing a sound to announce the device booting provides immediate feedback that the system is initializing.

- **Defining goals:** Optimization goals should be clear and measurable, encompassing both technical requirements and end-user experience. Well-defined goals help maintain project focus and enable objective success measurement. In this context, the optimization objective revolves around critical applications. For instance, in Android, this could include optimizing the loading time of the Launcher or the initialization of essential startup applications.

Taking these key aspects into consideration and after measuring the boot time, a methodical guideline is established:

1. **Remove Unnecessary functions:**
    - Identify and remove functions that are not essential to the system's basic functionality: numerous startup processes contain tasks that could be deferred or discarded without impacting the core operations.
    - Streamline the startup process by excluding non-essential tasks or services: reducing the startup load can significantly decrease the overall startup time.
    - Consider the impact of each feature on the user experience and system performance: it is important to evaluate how removing or modifying a feature will affect the overall usability and performance of the system.
2. **Postpone, Parallelize, Reorder:**
    - Evaluate tasks that can be postponed to a later stage without compromising essential functionality: some operations do not need to be performed immediately at startup and can be postponed to improve initial efficiency.
    - Utilize task parallelization to take advantage of available resources simultaneously: running multiple tasks in parallel can reduce overall startup time by better leveraging the system's multitasking capabilities.
    - Reorder tasks based on dependencies and critical path analysis for optimal execution: critical path analysis helps identify the tasks that have the greatest impact on startup time and organize the order of operations to minimize delays.
3. **Optimize Necessary Functionality:**
    - Optimize critical functions required for basic system operation: essential functions must be performed as efficiently as possible.

- Use efficient algorithms, data structures, and resource management techniques: adopting optimized algorithms and effective resource management can significantly improve performance.
- Refine codes and configurations to minimize execution time: optimizing code and system settings can reduce execution times and improve overall efficiency.

In conclusion, optimizing the boot time of Infotainment systems is a multifaceted process that requires careful consideration of various factors. Effective boot time optimization not only improves the immediate user experience but also contributes to the overall reliability and efficiency of the system, ensuring it meets the high standards expected in modern automotive environments.

2.4 Boot Sequence

The boot sequence of an Infotainment system involves several stages, each playing a crucial role in initializing the system and preparing it for user interactions. Understanding this sequence is essential for identifying optimization opportunities:

- **Power-Up:** The boot sequence starts when the system receives power. The power supply unit ensures that a stable voltage is provided to all components of the system. The BIOS (Basic Input/Output System) firmware performs a **power-on test (POST)** to verify the integrity of hardware, including RAM and storage devices. The process will only continue if the necessary hardware is functioning properly; otherwise, the BIOS will generate an error message [27]. Hardware components such as the memory controller and system bus are then initialized to prepare the system for next code execution.

- **Boot ROM:** The Boot ROM code, a small segment of code in CPU, begins execution from a predefined, hardwired location in the ROM. It loads the Bootloader into RAM and initiates its execution. Once the power supplies are stable, the Boot ROM code takes over, starting the process [5].

- **1st Stage Bootloader:** The first phase of the Bootloader establishes the essential environment needed to load the second part. It can performs operations such as configuration of memory and access storage devices.

- **2nd Stage Bootloader:** After confirming the integrity of the kernel and boot partition, the system configures the network, memory, and other necessary settings for the kernel's operation. Subsequently, it loads the kernel into RAM and transfers control to it.

- **Kernel Loading Decompression:** During this phase, the kernel and the initramfs (initial RAM file system), which contains initialization scripts and drivers needed to prepare the boot environment, are loaded into the system's RAM. If compressed to minimize space usage and speed up loading times, they are decompressed at this stage.
- **Kernel Init:** The kernel then mounts the initramfs as a temporary root file system, initializes system components, configures memory and process scheduling, and loads the required drivers. Finally, it searches for and starts the Init process within the system files.
- **Init Process:** It is the first process to start during the operating system's boot sequence. It plays a vital role in initializing the environment and launching key system services [5].
- **Init Scripts:** Init process reads and carries out commands from initialization files, which usually have the .rc file extension [5]. Services like network management and device settings configuration are established.
- **Mounting the Root Filesystem:** The kernel locates and mounts the root file system as designated by the bootloader. Initially, it's mounted in read-only mode to verify its integrity, and subsequently, after successful checks, it's remounted in read-write mode to permit write operations.
- **Critical Applications:** The system operates the primary applications, which are monitored and managed to ensure they function correctly. These applications are essential for the basic operation and user interaction of the Infotainment system.

This process provides a detailed analysis of the complexity involved in booting the operating system, emphasizing the importance of each step to ensure proper system booting. Ultimately, understanding the start-up sequence enables the improvement of system efficiency by intervening directly on critical points, focusing optimization efforts on the most relevant parts of the system.

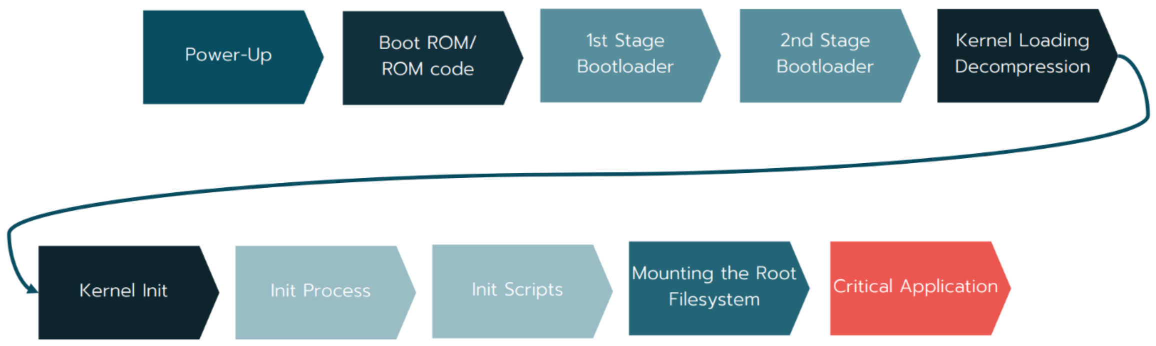


Figure 2.1. Boot Sequence.

2.4.1 Kernel Compression and Decompression

Kernel compression involves reducing the size of the kernel image to occupy less storage space, which is crucial for devices with limited resources such as embedded systems, and to decrease the loading time into memory during booting, facilitating faster startup times.
This process represents a balanced trade-off between speed and space efficiency, enabling the selection of an optimal compression algorithm tailored to the specific operating system and hardware requirements. Some of the most common compression algorithms include:

- **GZIP:** The well-established gzip compression offers a solid balance between the compression ratio and decompression speed.
- **BZIP2:** Provides better compression ratios than GZIP but is slower in terms of compression and decompression. Bzip2 requires a significant amount of memory (in modern Kernels at least 8MB RAM or more for booting).
- **LZMA:** It offers high compression ratios and is used in scenarios where space is critical. Decompression speed is between gzip and bzip2, while compression is slowest.
- **LZ4:** It offers fast compression and decompression speeds, making it suitable for scenarios where speed is prioritized.
- **LZO:** Provides fast compression and decompression with moderate compression ratios.
- **XZ:** Provides slow compression speed but excellent compression ratios.
- **ZSTD:** It offers a good balance between compression ratio and speed.

In addition to compression time, it is also crucial to evaluate decompression time when selecting an algorithm, as this represents the extra time incurred during the boot process. The actual speeds of compression and decompression can also be influenced by kernel-specific optimizations and hardware capabilities. Therefore, benchmarking these algorithms on the kernel and hardware configuration is essential to determine the best choice.
The selection of a compression algorithm is based on factors such as compression speed, decompression speed, and compression ratio, which is the ratio between the original image size and the compressed image size.

Chapter 3

Case study

3.1 Work environment configuration

3.1.1 Raspberry Pi 4

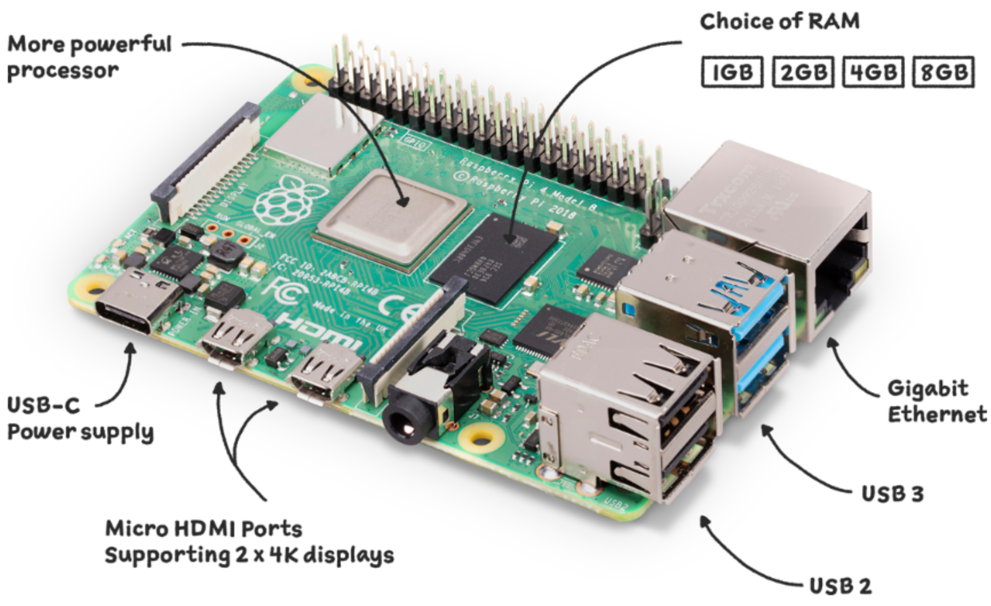


Figure 3.1. Raspberry Pi 4 Model B [22].

The Raspberry Pi 4 Model B (Pi4B) is the first of a new generation of Raspberry Pi computers supporting more RAM and with significantly enhanced CPU, GPU and I/O performance in a similar form factor, power enveloper and cost as the previous generation Raspberry Pi 3B+ [10].
This model is equipped with a high-performance 64-bit quad-core processor, enabling support for dual displays with resolutions up to 4K via its two micro HDMI ports. It offers hardware video decoding at up to 4kp60, up to 8GB of RAM, dual-band 2.4/5.0 GHz wireless LAN, Bluetooth 5.0, Gigabit Ethernet and USB 3.0.
This product maintains backward compatibility with the previous-generation Rasp-

berry Pi3 with similar power consumption, making it an easy upgrade for existing projects. However it brings substantial improvements in processor speed, multimedia performance, memory capacity and connectivity options.
The availability of higher RAM options, reaching up to 8GB, facilitates the running of more demanding applications and improves multitasking capabilities. Additionally, Gigabit Ethernet improves network connectivity, offering quicker and more stable internet access. The inclusion of dual-band wireless LAN and Bluetooth 5.0 support guarantees strong wireless communication, rendering the Raspberry Pi 4 Model B an adaptable and potent device for both developers and enthusiasts.
These advancements make the Pi4B suitable for a wider range of applications, from simple educational projects to complex industrial use cases, including embedded computing and IoT applications.

Specification [10]:

- **Processor:** Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC at 1.5GHz.
- **Memory:** 8GB LPDDR4-3200 SDRAM.
- **Connectivity:** 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless LAN, Bluetooth 5.0, Gigabit Ethernet, 2 USB 3.0 ports, 2 USB 2.0 ports.
- **GPIO:** Standard 40-pin GPIO header (full backwards-compatible with previous boards).
- **Video and Sound:** 2 micro HDMI ports (up to 4Kp60 supported), 2-lane MIPI DSI display port, 2-lane MIPI CSI camera port, 4-pole stereo audio and composite video port.
- **Multimedia:** H.265 (4Kp60 decode), H.264 (1080p60 decode, 1080p30 encode).
- **SD card support:** Micro SD card slot for loading operating system and data storage.
- **Input power:** 5V DC via USB-C connector, 5V DC via GPIO header, Power over Ethernet (PoE)–enabled.
- **Environment:** Operating temperature 0-50 degrees Celsius.

To use Raspberry Pi, it is necessary a power supply (15W USB-C power supply) and a boot media (for example a microSD card with sufficient storage capacity and speed). By default, Raspberry Pis check for an operating system on any SD card inserted in the SD card slot. In our case study, the operating system image was installed directly in microSD card. To set up the Raspberry Pi 4, begin by connecting the peripherals:

- Insert the microSD card with the operating system image into the microSD card slot.
- Connect a monitor or TV using a micro HDMI cable.
- Attach a keyboard and mouse via the USB ports.

Next, power up the Raspberry Pi:

- Connect the USB-C power supply to the power port of the Raspberry Pi.
- Switch on the power supply; the Raspberry Pi should begin the boot process automatically.

In our case study, the Raspberry Pi 4 is pivotal, showcasing its ability to handle complex tasks and processes effortlessly. Its accessible setup and remarkable performance underscore its popularity.

3.1.2 Android 14

In this section of the thesis, the process of configuring and compiling the project for Raspberry Pi4 device will be explored, providing a detailed overview of the steps required to successfully boot the Android operating system. From setting up the development environment to compiling the source code, the step-by-step procedure will be analyzed to ensure a perfect implementation of the Raspberry Vanilla project [23]. To set up Android on the Raspberry Pi4 usign a Linux machine, follow these detailed steps:

1. **Establish Android build environment**: Begin by installing essential packages needed for Android development. These packages include git, which is crucial for downloading the Android Open Source Project (AOSP) source code, and repo, which simplifies managing multiple Git repositories within a single working directory. Here's the command to install these packages: **sudo apt-get install git-core gnupg flex bison build-essential zip curl zlib1g-dev libc6-dev-i386 x11proto-core-dev libx11-dev lib32z1-dev libgl1-mesa-dev libxml2-utils xsltproc unzip fontconfig.**
   Additionally, install repo with: **sudo apt-get install repo.** These commands set up the foundational tools required to build Android on your system [25].

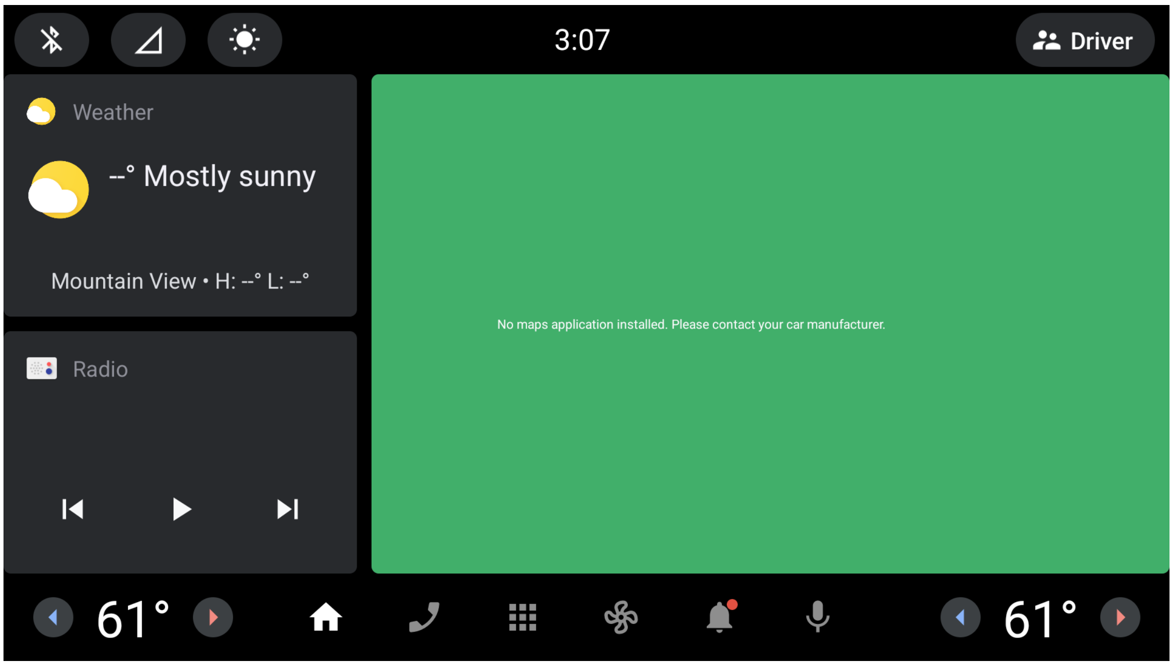
2. **Install additional packages**: These commands are commonly used to setup a software development environment, providing the necessary tools and libraries to compile and manage projects [25].
   - **sudo apt-get install bc coreutils dosfstools e2fsprogs fdisk kpartx mtools ninja-build pkg-config python3-pip**.
   - **sudo pip3 install meson mako jinja2 ply pyyaml dataclasses**.
3. **Initialize repo [25]:** Initialize the local repository for the Android source code. This involves setting up the repository and specifying the remote repository URL as the origin, with a specific branch. Execute the following commands:
   - **repo init -u https://android.googlesource.com/platform/manifest -b android-14.0.0_r22**. This command sets up the local repository and specifies the remote repository URL as the origin, with the branch 'android-14.0.0_r22'.
   - **curl -o -repo/local_manifests/manifest_brcm_rpi.xml -L https://raw.githubusercontent.com/raspberry-vanilla/android_local_manifest/android-14.0/manifest_brcm_rpi.xml**. This command fetches the XML file from the provided URL and saves it locally at the specified path.
4. **Sync source code [25]:**
   - **repo sync**. This process updates local working directory with the latest changes from remote repositories, ensuring you have the most current version of the codebase for your work.
5. **Setup Android build environment [25]:**
   - **. build/envsetup.sh**. It is commonly used in Android development environments to set up the necessary environment variables and functions for building Android source code.
6. **Select the device (rpi4) and build target (car for Android Automotive) [25]:**
   - **lunch aosp_rpi4_car-userdebug**. The "lunch" command is a script provided by the Android build system that allows you to choose a specific build target from a list of available configurations.
7. **Compile [25]:** Finally, compile the Android source code to generate the necessary system images required for running android on device. The following command:

- **make bootimage systemimage vendorimage -j**. This command is used to compile Android source code and generate various system images required for running Android on device. The **boot image** is a specialized package that includes the kernel and RAMDisk, essential for the device's booting process. The **system image** holds the Android system files, encompassing the framework, libraries, and system applications, and is mounted as the /system partition on the device. The **vendor image** contains the hardware-specific binaries and libraries necessary for the device's operation, which are part of the /vendor partition on the device.

8. **Make flashable image for the device (rpi4) [25]:**

   - **./rpi4-mkimg.sh**. It is a command used to execute a shell script that automates the process of creating a customized image for running Android on a Raspberry Pi 4 device.

After creating the image on a Linux machine, the next step involves flashing it onto an SD card using the **dd** command. dd is a powerful utility present in Unix-like systems that is widely used for copying and converting files. This process ensures that the SD card is prepared with the customized Android image ready for installation on the Raspberry Pi. Once the flashing process is complete, insert the SD card into the Raspberry Pi and power on the board to initiate the Android operating system.
Upon booting, the Raspberry Pi initializes with the newly installed Android image. This step marks the transition from the development phase to practical application, where the Raspberry Pi transforms into a functional Android device suitable for various applications.
To visualize the successful boot process and confirm the operating system's functionality, connect the Raspberry Pi board to a screen using an HDMI cable. This setup allows interaction with the Android graphical interface, confirming the successful deployment and enabling further configuration and testing as needed.
The following screenshots showcase the Android launcher, providing access to applications, and the menu interface, allowing navigation through various system settings and functionalities.



No maps application installed. Please contact your car manufacturer.

Figure 3.2. Android Auto Launcher.

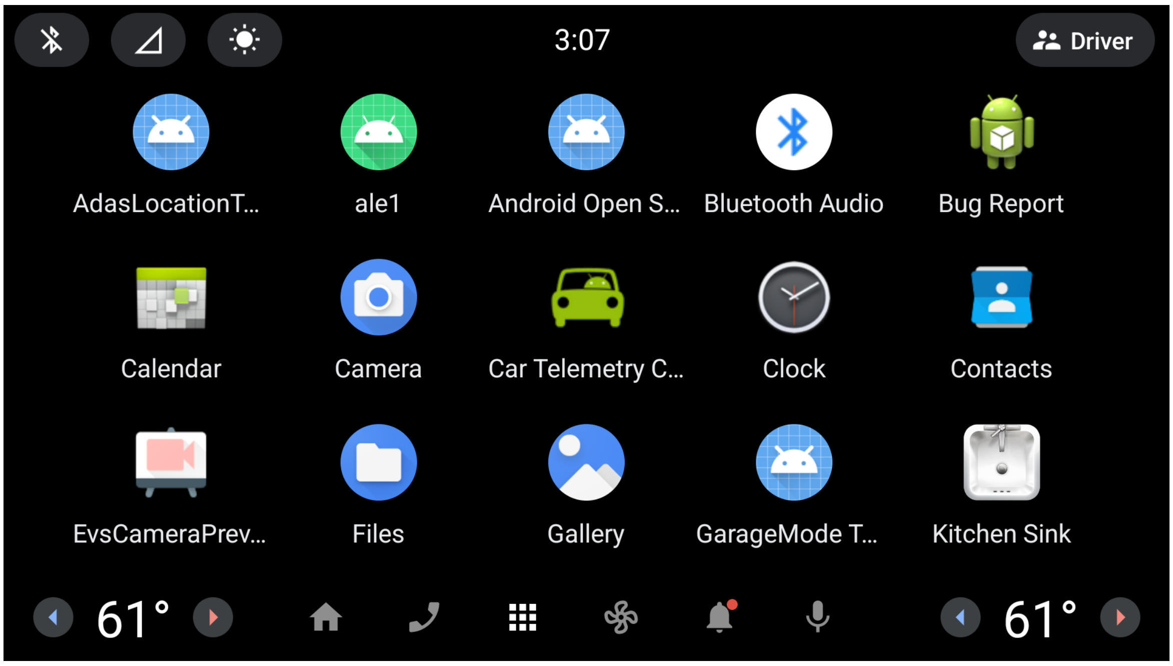


Figure 3.3. Menu Interface.

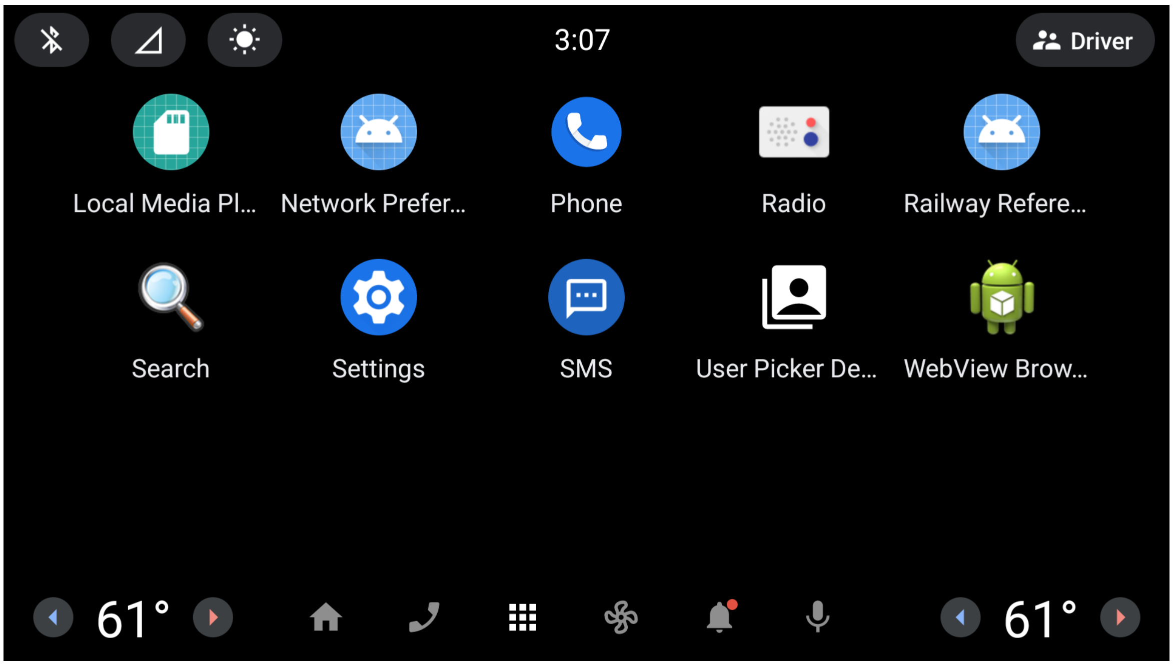


Figure 3.4. Menu Interface.

3.2 Android connection via Android Debug Bridge

Android Debug Bridge (adb) is an important command tool that enables the communication between the device. This tool provides also access to a Unix shell, that allowing the execution of different commands on the device [6]. It is composed by three different components:

- **A client** that is responsible for sending commands and operates on our development machine. It can be activated via command-line by executing an adb command.
- **A daemon (adbd)** which executes commands on the devices. It is managed as background process in each device.
- **A server** that manages communication between the client and daemon. Server operates as background process in our development machine.

The connection with device can be established either over USB or Wi-Fi. In order to utilize adb with a device connected via USB, it is necessary to activate **USB debugging** in the Developer options located in the device's system settings.
To activate the usb debugging it is necessary to access the Developer options. This can be achieved by tapping 7 times consecutively the **Build number** option within Settings > About phone > Build number.
After the activation of the developer options, **Usb debugging** can be activated directly within it. Debugging options provide methods to configure on device debugging

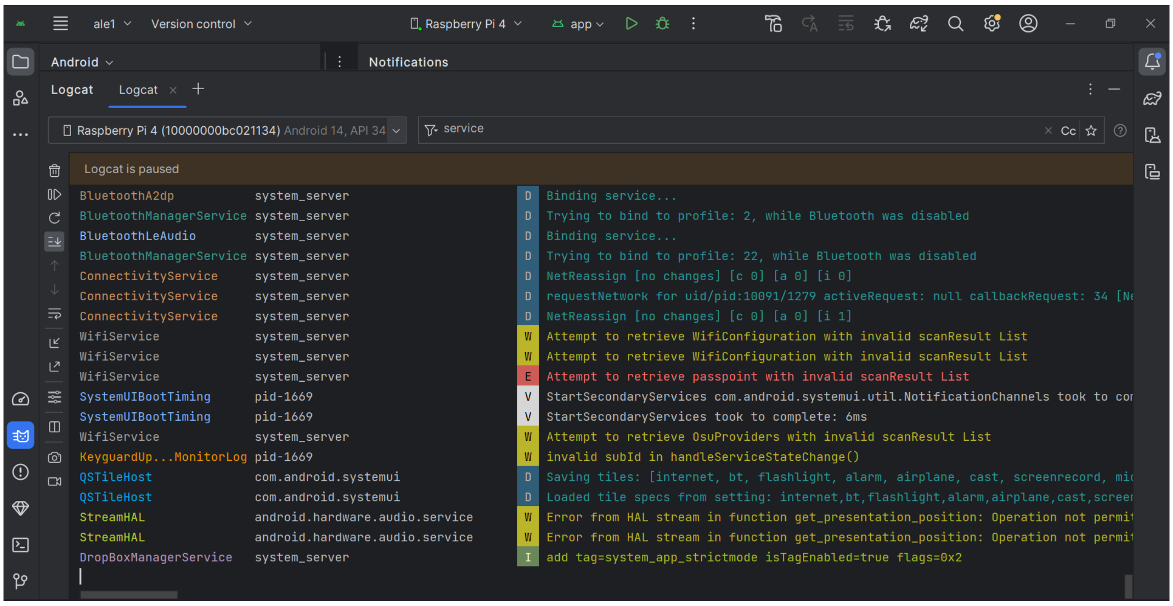
and establish communication between device and development machine [6].
Alternatively, Android 11 and higher support wireless debugging via adb, without ever needing to physically connect device via USB. To use wireless debugging, workstation and device must be connected to the same wireless network. To establish a connection:

- Enable **developer options** on device.
- Open Android studio (integrated development environment officially provided by Google for android application development) and select **Pair Devices Using Wi-Fi**.
- On the device, tap **Wireless debugging** (found in the developer options) and pair the device (with QR code or pairing code) [6].

In both cases we can verify that device is connected by executing *adb devices.* If connected, the device name is displayed). Once the device is connected, it is possible to access the system logs via the Logcat tool.

3.2.1 Logcat

Logcat is a system logging utility integrated into the Android operating system, enabling the viewing of system logs. These logs include debug messages, errors, and warnings from various operating system components and applications on the device. It is incorporated into the Android Studio development environment, offering a comprehensive view of system activities. The Logcat tool is accessible directly via the command line, or you can observe messages directly within Android Studio [19].
Logs contain multiple metadata fields, in addition to the tag and priority. The format of message output can be customized as needed. For the specific case study, the **'adb logcat -v time'** command was utilized; This format displays the date, recording time, priority, tag, and PID of the process that generated the message. The next figure shows an example of using logcat (in Android Studio) to view device logs.



```
BluetoothA2dp           system_server                   D  Binding service...
BluetoothManagerService system_server                   D  Trying to bind to profile: 2, while Bluetooth was disabled
BluetoothLeAudio        system_server                   D  Binding service...
BluetoothManagerService system_server                   D  Trying to bind to profile: 22, while Bluetooth was disabled
ConnectivityService     system_server                   D  NetReassign [no changes] [c 0] [a 0] [i 0]
ConnectivityService     system_server                   D  requestNetwork for uid/pid:10091/1279 activeRequest: null callbackRequest: 34 [Ne
ConnectivityService     system_server                   D  NetReassign [no changes] [c 0] [a 0] [i 1]
WifiService             system_server                   W  Attempt to retrieve WifiConfiguration with invalid scanResult List
WifiService             system_server                   W  Attempt to retrieve WifiConfiguration with invalid scanResult List
WifiService             system_server                   E  Attempt to retrieve passpoint with invalid scanResult List
SystemUIBootTiming      pid-1669                        V  StartSecondaryServices com.android.systemui.util.NotificationChannels took to cor
SystemUIBootTiming      pid-1669                        V  StartSecondaryServices took to complete: 6ms
WifiService             system_server                   W  Attempt to retrieve OsuProviders with invalid scanResult List
KeyguardUp...MonitorLog pid-1669                        W  invalid subId in handleServiceStateChange()
QSTileHost              com.android.systemui            D  Saving tiles: [internet, bt, flashlight, alarm, airplane, cast, screenrecord, mic
QSTileHost              com.android.systemui            D  Loaded tile specs from setting: internet,bt,flashlight,alarm,airplane,cast,screer
StreamHAL               android.hardware.audio.service  W  Error from HAL stream in function get_presentation_position: Operation not permit
StreamHAL               android.hardware.audio.service  W  Error from HAL stream in function get_presentation_position: Operation not permit
DropBoxManagerService   system_server                   I  add tag=system_app_strictmode isTagEnabled=true flags=0x2
```

Figure 3.5. Example of Logcat messages in Android studio.

3.3 Measurements and tools used

3.3.1 Key Performance Indicator

The choice of key performance indicators (KPIs) is crucial to monitor the performance of our android device. After a careful analysis of the logs obtained from the device, different KPIs (each selected KPI has a corresponding printed log message) were chosen. Each KPI corresponds to a specific log message indicating the successful execution of a system service or functionality. Below is a detailed description of each KPI and its associated log message:

1. **Automotive Functionality Handling:**

   - Description: The system must be capable of managing automotive vehicle-related functionalities, such as communication with automotive components and integration with vehicle systems.
   - Log: *I/HidlServiceManagement: Registered android.hardware.automotive.*

2. **Bluetooth Service Initialization:**

   - Description: The Bluetooth service must be initialized correctly to ensure the device is ready to handle Bluetooth functionalities, including connection and communication with other Bluetooth devices.
   - Log: *I/HidlServiceManagement: Registered android.hardware.Bluetooth.*

3. **USB Device Management:**

- Description: The system should effectively manage USB devices and their functionalities, such as data transfer and connection with external peripherals.
- Log: *I/HidlServiceManagement: Registered android.hardware.usb.gadget.*

4. **External Camera Service:**
   - Description: The system should start a service that manages access to external cameras on the Android device, allowing the use of webcams or other connected cameras.
   - Log: *I/android.hardware.camera.provider-V1-external-service: external webcam service is starting.*
5. **Audio Functionality Handling:**
   - Description: The system must handle audio functionalities effectively, including sound playback and recording, ensuring an optimal audio experience.
   - Log: *I/HidlServiceManagement: Registered android.hardware.audio.*
6. **Media Functionality Management:**
   - Description: The system should manage media functionalities, particularly those related to ffmpeg, for the playback and editing of multimedia content.
   - Log: *I/HidlServiceManagement: Registered android.hardware.media.*
7. **Boot Animation Start:**
   - Description: The system should initiate the boot animation process, providing a visual indication of the device startup.
   - Log: *D/BootAnimation: BootAnimationStartTiming start time.*
8. **PowerManager Service Start:**
   - Description: The PowerManager service, responsible for managing power-related functions, must start correctly to ensure efficient energy management.
   - Log: *D/SystemServerTiming: StartPowerManager.*
9. **TelephonyRegistry Service Start:**
   - Description: The TelephonyRegistry service, which manages telecommunication functionalities such as call information, network connections, and SIM card management, must start correctly.

- Log: *D/SystemServerTiming: StartTelephonyRegistry.*

10. **WiFi Services Start:**
    - Description: The system should start WiFi services to ensure network connectivity, including the management of wireless connections.
    - Log: *D/SystemServerTiming: StartWifi.*

11. **System Update Manager Service Start:**
    - Description: The System Update Manager, responsible for searching, downloading, and installing new system updates, must start correctly to keep the device up-to-date.
    - Log:*D/SystemServerTiming: StartSystemUpdateManagerService.*

12. **Notification Manager Start:**
    - Description: The Notification Manager, which manages system notifications, must start correctly to ensure notifications are handled and displayed properly.
    - Log: *D/SystemServerTiming: StartNotificationManager.*

13. **Location Manager Service Start:**
    - Description: The Location Manager service, responsible for managing location-related tasks and data from GPS, WiFi, and cellular networks, must start correctly.
    - Log: *D/SystemServerTiming: StartLocationManagerService.*

14. **System UI Start:**
    - Description: The System UI, which manages user interface components like the navigation bar, status bar, and launcher, should start correctly to provide a seamless user experience.
    - Log: *D/SystemServerTiming: StartSystemUI.*

15. **Boot Animation Stop:**
    - Description: The system should stop the boot animation process once the booting is complete, indicating that the device is ready for use.
    - Log: *D/BootAnimation: BootAnimationStopTiming.*

3.4 Bootchart

Bootchart is a performance analysis and visualization tool for the system's boot process. Initially created for the Linux kernel, it captures and visually represents the events and their interdependencies throughout the boot sequence. It gathers data on resource utilization and process details during startup, which is then rendered into a chart encoded in PNG or SVG format.
This diagram is useful for identifying bottlenecks and inefficiencies during the boot process, simplifying the task of taking targeted actions to enhance your device's overall performance. Bottlenecks can significantly slow down the startup time, affecting user experience, especially in devices with limited hardware resources. By visualizing the boot process, users can identify exactly where delays occur and which processes consume the most resources.
Integrated into the Android development environment, this tool can be activated via certain settings, serving as an essential instrument for refining the user experience right from the initial stages of device startup.
Bootchart is already present on Android but it needs to be enabled with the following commands after connecting the device with adb:

- **adb shell 'touch /data/bootchart/enabled'.**
- **adb reboot.**

After the reboot these files will be present in the directory /data/bootchart:

1. **Header.**
2. **proc_diskstats.log.**
3. **proc_ps.log.**
4. **proc_stat.log.**

To avoid data collection each time, delete /data/bootchart/enabled after the operation.
To render the graph on a Linux machine, download the "Bootchart" project directly from Github, using the following URL: `https://github.com/xrmx/bootchart.git`.
After this operation, a folder called Bootchart will be created within the chosen directory.
The four files obtained inside the /data/boochart/ folder are moved, with command **adb pull**, to the /bootchart folder, on Linux machine, and then compressed with the tar command:

- **tar –cvf bootchartandroid.tar header proc_diskstats.log proc_ps.log proc_stat.log.**

Once the compressed file is generated, the SVG file will be create using the following commands:

- **make pybootchartgui/main.py.**

- **python pybootchartgui.py bootchartandroid.tar -f svg.**

The following diagram, divided in two parts, provides a detailed visual representation of the boot process captured from the Android device.

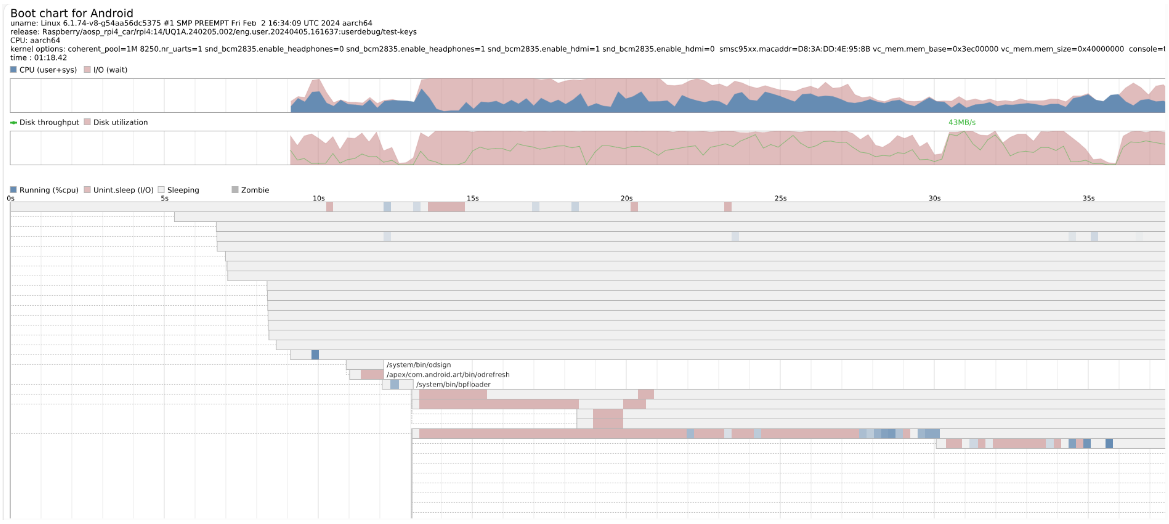


Figure 3.6. Bootchart graph full project.

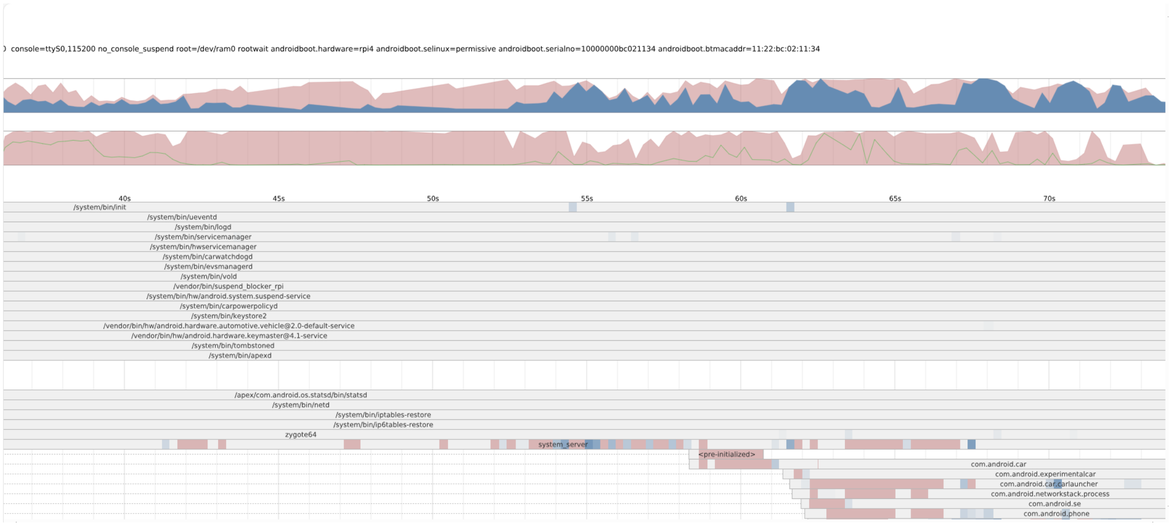


Figure 3.7. Bootchart graph full project.

The graph represents three different sections:

- **CPU (user+sys) usage (the sum of all the running tasks) and I/O (wait):**

1. User CPU time: the duration the processor dedicates to executing user application code.
2. System CPU time: the duration the processor spends handling operating system tasks on behalf of the application.
3. I/O (wait): the amount of time the CPU spends waiting for input/output operations to complete. During this wait time, the CPU is idle and cannot proceed with the process until the necessary data is available.

- **Disk Utilization and Disk Throughput:**

  1. Disk Utilization: the percentage of time the disk is actively engaged in read/write operations. High disk utilization indicates frequent disk activity, which can lead to I/O bottlenecks if the workload exceeds the disk's throughput capacity.
  2. Disk Throughput: the amount of data read from or written to the disk over a given period, typically measured in MB/s. Peaks in disk throughput indicate periods of intensive disk activity.

- **Boot process tree:** The most important part of bootchart graphs, showing parent-child relationships, states and CPU usage. The process tree includes:

  1. **Running (%cpu):** Percentage of CPU used by a specific process.
  2. **Unint.sleep (I/O):** Indicates processes waiting for non-interruptible events, typically I/O operations.
  3. **Sleeping:** Indicates processes waiting for events.
  4. **Zombie:** Indicates processes that have terminated but still have an entry in process table, waiting for parents to read their exit status.

Each part horizontally evolves on a timeline from 0 seconds to the end of the process analysis, providing a chronological view of events.
The Bootchart graph serves as a potent instrument, offering an extensive overview of the system's boot sequence. This allows developers to improve device performance by making informed decisions and implementing targeted optimizations.

3.5 Bootgraph

Bootgraph is another valuable tool for analyzing startup performance. It converts the output from the **'dmesg'** command into an SVG graph, emphasizing the initialization time of different kernel functions. The dmesg command is essential for system debugging, offering a comprehensive log of kernel events, such as errors and warnings.

Initcalls are initialization calls that the kernel makes during boot time to configure various operating system subsystems and modules. These calls are essential to prepare the operating environment before the system is ready to run user applications. Each initcall is logged with a timestamp in the dmesg log upon invocation and completion, enabling the calculation of its execution time. To enable this feature, it is necessary to pass the **"initcall_debug"** option to the kernel command line. In this project, the 'cmdline.txt' file inside the /boot folder of the SD card is edited directly. The SD card is directly connected to the Linux PC, and the file within the boot folder has been modified to include the following information: "printk.time=1 initcall_debug".
However, using the initcall_debug option increases the amount of messages produced by the kernel during startup.
To avoid log buffer overrun, the size of the log printk buffer is increased. This can be done by increasing the **CONFIG_LOG_BUF_SHIFT** value from 14 to 18, thus increasing the buffer size from 16k to 256k.
Additionally, the following configurations must be set: **CONFIG_PRINTK_TIME** and **CONFIG_KALLSYMS**. The first option displays printk times, while the second ensures that function names are printed instead of memory addresses.
Then, reconnect the SD card to the Raspberry Pi, power on the device, and execute the following commands after establishing the ADB connection of the device to the PC:

- **adb root** - This command restarts the connection with root permissions.
- **adb shell (and after 'su')** - Opens a shell on the device and switches to the superuser.
- **mount -o remount,rw /** - Remounts the root file-system in read-write mode to allow modifications.
- **dmesg > boot.log** - Redirects the dmesg output to a file named boot.log.
- **exit** - Exits from the shell.
- **adb pull boot.log /path/to/local_directory** - Pulls the boot.log file from the device to a specified directory on the PC.

The dmesg command includes entries that start with "calling" and "initcall".

- **calling:** Indicates the kernel is invoking an initialization function.
- **initcall:** Indicates the completion of an initialization function.

These messages help in diagnosing and understanding the boot process, as they detail which modules are being initialized, their success or failure status, and the time taken for each initialization.

```
[    0.786068] calling  ir_rc6_decode_init+0x0/0x48 @ 1
[    0.786081] IR RC6 protocol handler initialized
[    0.786085] initcall ir_rc6_decode_init+0x0/0x48 returned 0 after 4 usecs
[    0.786101] calling  ir_rcmm_decode_init+0x0/0x48 @ 1
[    0.786114] IR RCMM protocol handler initialized
[    0.786118] initcall ir_rcmm_decode_init+0x0/0x48 returned 0 after 4 usecs
[    0.786132] calling  ir_sanyo_decode_init+0x0/0x48 @ 1
[    0.786147] IR SANYO protocol handler initialized
[    0.786151] initcall ir_sanyo_decode_init+0x0/0x48 returned 0 after 4 usecs
[    0.786165] calling  ir_sharp_decode_init+0x0/0x48 @ 1
[    0.786179] IR Sharp protocol handler initialized
[    0.786183] initcall ir_sharp_decode_init+0x0/0x48 returned 0 after 4 usecs
[    0.786197] calling  ir_sony_decode_init+0x0/0x48 @ 1
[    0.786210] IR Sony protocol handler initialized
[    0.786214] initcall ir_sony_decode_init+0x0/0x48 returned 0 after 4 usecs
[    0.786229] calling  ir_xmp_decode_init+0x0/0x48 @ 1
[    0.786242] IR XMP protocol handler initialized
[    0.786246] initcall ir_xmp_decode_init+0x0/0x48 returned 0 after 4 usecs
[    0.786261] calling  gpio_ir_recv_driver_init+0x0/0x3c @ 1
[    0.786385] initcall gpio_ir_recv_driver_init+0x0/0x3c returned 0 after 109 usecs
[    0.786402] calling  unicam_driver_init+0x0/0x3c @ 1
[    0.786523] initcall unicam_driver_init+0x0/0x3c returned 0 after 106 usecs
```

Figure 3.8. Example of log messages produced by dmesg command.

After downloading the script from Github (bootgraph.pl) to the Linux PC, it is placed in a folder where the file generated via dmesg is also present. It is important to ensure that Perl is installed on PC, as it is required to run the script. Finally, to generate the SVG file, the following command is executed directly inside the chosen folder:

- **perl bootgraph.pl boot.log > output.svg**

The *bootgraph.pl* script analyzes the output of dmesg, identifying the start and end of initialization calls, and generates a graph. This graph provides a clear representation of possible bottlenecks during startup, allowing for the optimization of system performance.

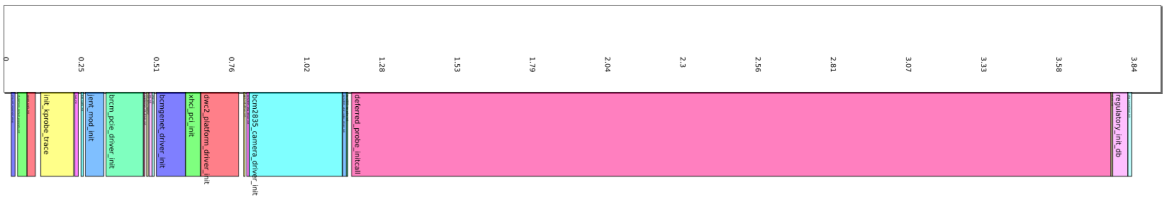


Figure 3.9. Example of generated graph.

Interpreting the graph involves identifying long-running initcalls, which are potential candidates for optimization. By reducing the time taken by these functions, overall boot time can be significantly improved. This tool is essential for improving overall kernel efficiency and reducing boot times.

Chapter 4

Boot time Optimization

In this chapter, we will discuss one by one the methodologies applied to reduce the initial boot time of the Android system. The goal, as specified above, is to ensure a quick and smooth start, thus improving the user experience.

4.1 Removing unnecessary projects

The first phase of optimization involved eliminating unnecessary OS components. In the process of configuring and compiling the Android 14 project for RPI4, the manifest file remove_projects.xml was added (this file is already present in the Git project [23]), through the command:

- **curl -o .repo/local_manifests/remove_projects.xml -L https://raw.githubusercontent.com/raspberry-vanilla/ android_ local_manifest/android-14.0.0_r22/remove_projects.xml.**

This file specifies a list of projects that are not required for the current build and that can be excluded. The main removed projects are listed below:

- **Amlogic Devices:** A chipset manufacturer is renowned for its systems on chips, commonly utilized in multimedia devices.
  - **'device/amlogic/yukawa' and 'device/amlogic/yukawa-kernel':** Regarding configuration and support for devices based on Amlogic Yukawa chipsets, which are not relevant for Raspberry.
- **Google Devices:** Involving codenamed Pixel Devices [15].
  - **'device/google/barbet' and 'device/google/barbet-sepolicy':** Regarding configuration and SELinux security policies support for Google Barbet (Pixel 5A), which are not relevant for Raspberry.

– **'device/google/bluejay', 'device/google/bluejay-sepolicy' and 'device/google/bluejay-kernel':** Regarding configuration and SELinux security policies support for Google Bluejay (Pixel 6A), which are not relevant for Raspberry.

– **'device/google/bramble' and device/google/bramble-sepolicy':** Regarding configuration and SELinux security policies support for Google Bramble (Pixel 4a 5G), which are not relevant for Raspberry.

– **'device/google/contexthub':** Regarding support for Context Hub, a platform hardware for sensor that is not relevant in Raspberry.

– **'device/google/coral', 'device/google/coral-kernel' and 'device/google/coral-sepolicy':** Regarding configuration and SELinux security policies support for Google Coral (Pixel 4), which are not relevant for Raspberry.

– **'device/google/gs101', 'device/google/gs101-sepolicy', 'device/google/gs201', 'device/google/gs201-sepolicy' and 'device/google/gs-common':** Regarding configuration of Google chipsets that are not relevant for Raspberry.

– **'device/google/lynx', 'device/google/lynx-sepolicy', and 'device/google/lynx-kernel':** Regarding configuration and SELinux security policies support for Google Lynx (Pixel 7a), which are not relevant for Raspberry.

– **'device/google/felix', 'device/google/felix-sepolicy', and 'device/google/felix-kernel':** Regarding configuration and SELinux security policies support for Google Felix (Pixel Hold), which are not relevant for Raspberry.

– **'device/google/redfin' and 'device/google/redfin-sepolicy'** Regarding configuration and SELinux security policies support for Google Redfin (Pixel 5), which are not relevant for Raspberry.

– **'device/google/tangorpro', 'device/google/tangorpro-sepolicy' and 'device/google/tangorpro-kernel'** Regarding configuration and SELinux security policies support for Google Tangorpro (Pixel Tablet), which are not relevant for Raspberry.

– **'device/google/sunfish', 'device/google/sunfish-sepolicy' and 'device/google/sunfish-kernel'** Regarding configuration and SELinux security policies support for Google Sunfish (Pixel 4a), which are not relevant for Raspberry.

- **Linaro Platforms:** Involving projects of Linaro, an engineering organization with the goal of improving the integration and performance of open source software on ARM hardware. These projects are not relevant for Raspberry, such as:
  - **'device/linaro/dragonboard'.**
  - **'device/linaro/hikey'.**
- **Hardware Platforms:** Involving projects for hardware components that are not important in Raspberry, such us:
  - **platform/hardware/invensense**'.
  - **'platform/hardware/qcom/audio**'.
  - **'platform/hardware/qcom/gps**'.
  - **'platform/hardware/ti/am57x**'.

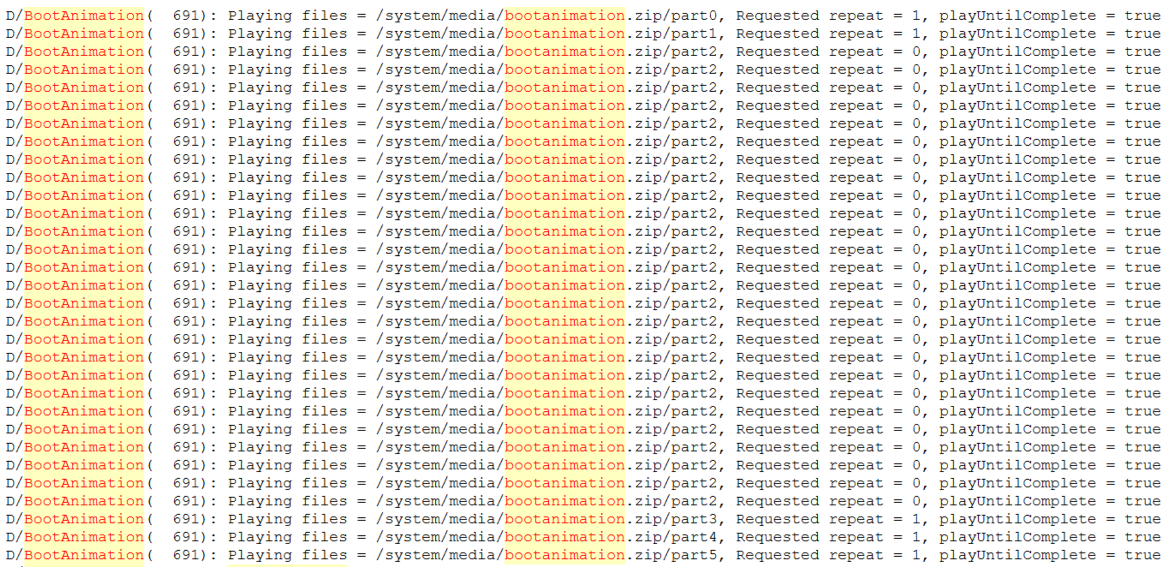
Eliminating unnecessary projects from our device leads to a simplified operating system and smaller size, which ensures a faster boot time for the device.

4.2 Removing Initial Animation

Subsequently, the initial boot animation was eliminated. This change resulted in a decreased visible boot time by removing a step that, while visually appealing, added to the overall time required to make the operating system accessible to the user.
Within the logs obtained through the command 'adb logcat -v time', the following debug message is obtained:

- **D/BootAnimation(691): /system/media/bootanimation.zip is loaded successfully**.

The "bootanimation.zip" file is used for the Android startup animation and when the system boots, this file is loaded by the process responsible for handling the initial animation. The zipped file contains multiple folders, each filled with images that sequentially create the initial animation. The following image contains the logs relevant to the initial animation process, providing insights into how it is executed by the system.



```
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part0, Requested repeat = 1, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part1, Requested repeat = 1, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part2, Requested repeat = 0, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part3, Requested repeat = 1, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part4, Requested repeat = 1, playUntilComplete = true
D/BootAnimation(  691): Playing files = /system/media/bootanimation.zip/part5, Requested repeat = 1, playUntilComplete = true
```

Figure 4.1. Log messages related to the initial animation.

To remove the initial boot animation, the following steps were performed:

- **adb root:** This command grants root privileges on the Android device via adb, allowing advanced system operations.
- **adb shell (and after 'su'):** Opens a shell on the device and switches to the superuser.
- **mount -o remount,rw /:** Remounts the root file-system in read-write mode to allow modifications.
- **cd /system/media :** Changing the working directory to 'system/media', where the file 'bootanimation.zip' is located.
- **chmod 777 bootanimation.zip:** Sets permissions on the file to allow read, write, and execute access for all users, facilitating its removal. This command might be necessary if permission issues are encountered when attempting to delete the file.
- **rm -rf bootanimation.zip:** Finally deleting the file on the Android device.

Removing the initial boot animation not only accelerates the boot process but also conserves system resources. Another alternative to deleting it could be replacing the animation with a customization or adjusting the settings to minimize startup time.

4.3 Configuration of new version of Kernel

Following the analysis of Android 14's configuration on device, this section will delve into the new kernel image configuration. In particular, the newly configured

kernel version 6.1.84, replacing the previous 6.1.74, has resulted in improved initial boot performance of the system.
To configure the Android Image on a Linux machine, the following steps are followed [24]:

1. **Establish Android build environment and install repo:** As described in the previous section, it is essential to have the Android development environment set up and repo installed to proceed with kernel compilation.

2. **Initialize repo [24]:**

   - **repo init -u https://android.googlesource.com/kernel/manifest -b common-android14-6.1-lts.** This command initializes the local repository for the kernel source code, specifying the remote repository URL and the branch to use.
   - **curl -o .repo/local_manifests/manifest_brcm_rpi.xml -L https://raw.githubusercontent.com/raspberry-vanilla/android_kernel_manifest/android-14.0/manifest_brcm_rpi.xml –create-dirs.** The command retrieves the XML file from the given URL and stores it locally at the designated path.

3. **Sync the source code [24]:**

   - **repo sync.** This command synchronizes the local working directory with the remote repositories, ensuring the kernel source code is up to date.

4. **Compile the kernel [24]:**

   - **tools/bazel build –config=fast –config=stamp //common:rpi4.** This command compiles the kernel using Bazel, an advanced build system. The '–config=fast' flag optimizes the build process for speed, while '–config=stamp' ensures the build outputs are stamped with metadata for debugging purposes.

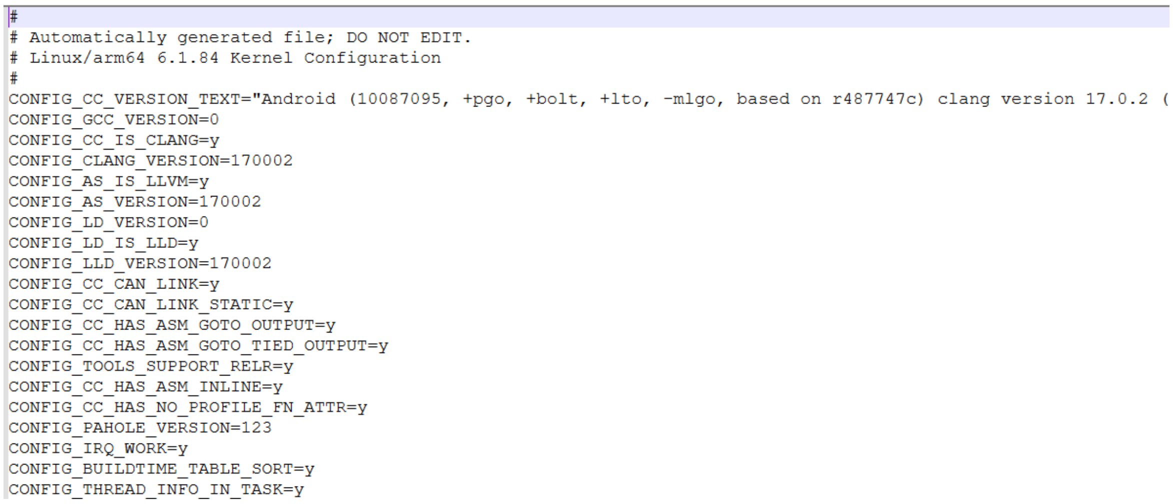
Following the operations, the compiled kernel image, dtbs, and overlays will be located in the **'bazel-bin/common/rpi4/arch/arm64/boot'** directory. Configuring and compiling the kernel separately avoids the need to recompile the entire Android project, thus saving time and resources. This approach ensures that any changes to the kernel can be quickly integrated and tested without affecting the rest of the project, thus simplifying the development process.
The new kernel Image is then moved directly into the /boot folder of SD card, after connecting SD card to the Linux machine, with the following command:

- **cp bazel-bin/common/rpi4/arch/arm64/boot/Image /boot/ :** The command ‘cp‘ copies the newly compiled kernel image from its location in the build directory to the /boot folder of the SD card. This step is crucial as it updates the kernel that the system will use during boot.

After transferring the new kernel Image to the SD card and ensuring its successful boot on the Raspberry, follow these steps to verify the kernel configuration:

- Reconnect the SD card to the Raspberry Pi.
- Power on the Raspberry and monitor the boot process in order to confirm that the system starts without errors using the new kernel.
- Once the Raspberry has booted successfully, execute these steps:
  - **adb root** - This command restarts the connection with root permissions.
  - **adb shell (and after ’su’)** - Opens a shell on the device and switches to the superuser.
  - **mount -o remount,rw /** - Remounts the root file-system in read-write mode to allow modifications.
  - **exit** - Exits from the superuser mode.
  - **exit** - Exits from the adb shell.
  - **adb pull /proc/config.gz /path/to/local_directory:** Pulls the kernel configuration file from the device to the specified directory on PC.
- After these operations, open the file **config.gz** with text editor and verify that the kernel version is correctly updated to 6.1.84.

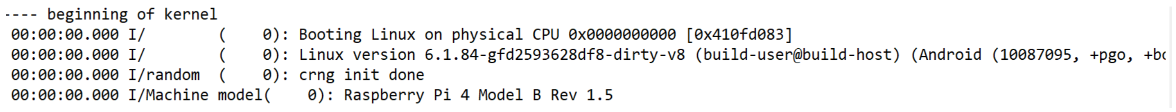
The following image 4.2 shows that the new kernel version has been correctly configured.



```
#
# Automatically generated file; DO NOT EDIT.
# Linux/arm64 6.1.84 Kernel Configuration
#
CONFIG_CC_VERSION_TEXT="Android (10087095, +pgo, +bolt, +lto, -mlgo, based on r487747c) clang version 17.0.2 (
CONFIG_GCC_VERSION=0
CONFIG_CC_IS_CLANG=y
CONFIG_CLANG_VERSION=170002
CONFIG_AS_IS_LLVM=y
CONFIG_AS_VERSION=170002
CONFIG_LD_VERSION=0
CONFIG_LD_IS_LLD=y
CONFIG_LLD_VERSION=170002
CONFIG_CC_CAN_LINK=y
CONFIG_CC_CAN_LINK_STATIC=y
CONFIG_CC_HAS_ASM_GOTO_OUTPUT=y
CONFIG_CC_HAS_ASM_GOTO_TIED_OUTPUT=y
CONFIG_TOOLS_SUPPORT_RELR=y
CONFIG_CC_HAS_ASM_INLINE=y
CONFIG_CC_HAS_NO_PROFILE_FN_ATTR=y
CONFIG_PAHOLE_VERSION=123
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_TABLE_SORT=y
CONFIG_THREAD_INFO_IN_TASK=y
```

Figure 4.2. New version 6.1.84 of Kernel.

The new kernel version can also be checked directly from the logs obtained using the command **'adb logcat -v time'**. The following image shows the reference logs.



```
---- beginning of kernel
 00:00:00.000 I/        (    0): Booting Linux on physical CPU 0x0000000000 [0x410fd083]
 00:00:00.000 I/        (    0): Linux version 6.1.84-gfd2593628df8-dirty-v8 (build-user@build-host) (Android (10087095, +pgo, +bo
 00:00:00.000 I/random  (    0): crng init done
 00:00:00.000 I/Machine model(    0): Raspberry Pi 4 Model B Rev 1.5
```

Figure 4.3. Log messages of kernel.

4.4 Disabling Serial Console

The next step in the optimization concerned the study of the logs obtained through the dmesg command. As expected, utilizing the bootgraph.pl script enabled the identification of bottlenecks in the startup process.
First, the graph obtained through the bootgraph.pl script allowed the identification of long-running initcalls, showing potential candidates for optimization. The following image shows the graph obtained from the dmesg command:

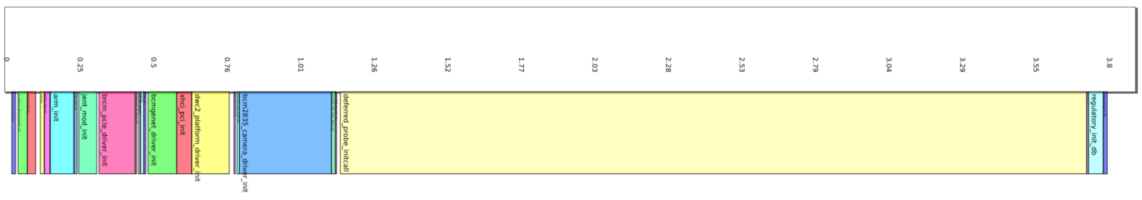
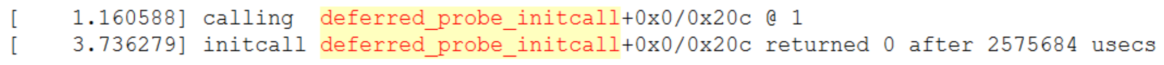


Figure 4.4. Graph obtained through bootgraph.pl

In particular, the initcall **deferred_probe_initcall** shows remarkable latency, negatively affecting the overall performance of the system.



```
[    1.160588] calling  deferred_probe_initcall+0x0/0x20c @ 1
[    3.736279] initcall deferred_probe_initcall+0x0/0x20c returned 0 after 2575684 usecs
```

Figure 4.5. Messages of deferred_probe_initcall.

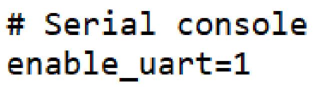
Between the "calling deferred_probe" and "initcall deferred_probe" messages, certain initializations appear to introduce a delay in the system. The excerpts below from the logs underscore particular occurrences:

- **probe of serial0-0 returned 0 after 255281 usecs:** The message confirms that the serial0-0 device has successfully completed the probing process, as indicated by the returned value of 0, after approximately 255 milliseconds. The serial0-0 device is one of the system's serial ports.

- **probe of fe215040.serial returned 0 after 280653 usecs:** The message confirms that the serial device, assigned the address fe215040, has successfully completed the probing process, as indicated by the returned value of 0, in approximately 281 milliseconds. This device represents an additional serial port within the system.

- **probe of gpu returned 0 after 2066263 usecs:** The message confirms that the GPU (Graphics Processing Unit) device successfully finished the probing process, as indicated by the returned value of 0, in approximately 2.07 seconds. This duration is notably lengthy, particularly in comparison to the probing times of other devices in the system.

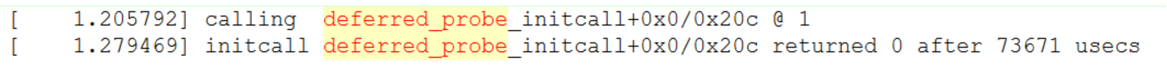
Delays in probing serial ports contribute to the total system initialization time. In particular, the GPU showed very long probing times, which could be related to the delays introduced by the serials.
The issue was resolved by disabling the device's serial console through direct modifications to the **config.txt** file in the /boot directory. In production environments, where debugging is not required, serial consoles are not necessary.
The config.txt file in the boot partition is a configuration file that is primarily used on Raspberry devices to configure various aspects of the hardware during the boot process. This file is read by the device's firmware at startup, and the settings in it directly affect how the operating system is loaded and how hardware peripherals are initialized [9].
Analyzing the config.txt file, the serial console is currently enabled as we can see from the following image:



```
# Serial console
enable_uart=1
```

Figure 4.6. Serial console enabled in config.txt file.

To disable the serial console, it was necessary to add the # character at the beginning of the "enable_uart=1" line. After saving the changes, close the file and reboot the system to apply them effectively.
Turning off the serial console has led to a notable enhancement in system performance, especially in terms of system boot time. Disabling serial ports has resulted in a major change in the **deferred_probe_initicall**, as illustrated in the following image:



```
[    1.205792] calling  deferred_probe_initcall+0x0/0x20c @ 1
[    1.279469] initcall deferred_probe_initcall+0x0/0x20c returned 0 after 73671 usecs
```

Figure 4.7. Messages of deferred_probe_initcall after disabling serial console.

Furthermore, the GPU probing duration, previously lengthy, has been reduced significantly from approximately 2.07 seconds to just under 19 milliseconds, highlighting the substantial performance gains achieved. This change was essential in improving the initial boot performance of the device, ensuring faster and more efficient startup.

4.5 Disabling unnecessary Kernel options

Following the optimization achieved by disabling the serial console, the next phase of the study focused on analyzing the active kernel configurations. The goal was to identify and fine-tune specific settings that could further enhance the system's performance and stability.

The kernel configuration plays a crucial role in determining how the system interacts with hardware components, manages resources, and executes processes. By carefully reviewing and adjusting these settings, it is possible to optimize the system for its intended use case, minimizing overhead and improving efficiency.
This optimization process involves removing kernel components and features that are not essential to the intended operation of the device. The project's target is the optimization of the system startup, which must be as fast as possible. In this specific case, everything that does not concern the system startup is unnecessary and has been removed. The main reasons why these changes lead to improvements are as follows:

- **Kernel Size Reduction:** Each enabled configuration adds code to the kernel, increasing the overall kernel file size. A smaller kernel loads faster, reducing boot time.

- **Reducing Workload:** Unnecessary configurations require the management of extra drivers, modules, and services, which can decelerate the boot process. By eliminating these components, the system has to perform fewer operations during boot, thus speeding up the process.

- **Resource Optimization:** Removing unused features frees up system resources, such as memory and CPU, which can be allocated for more critical operations.

As mentioned, the **config.gz** file located in the /proc directory within the device shell serves as the kernel configuration file, detailing which configurations are active or disabled. As mentioned above, configuration and compilation of the kernel were performed separately to avoid compiling the entire project. Each change in kernel configurations was made by directly editing a file within this project, compiling, and creating the modified kernel Image.
The following steps outline the process for modifying the kernel configuration and verifying the changes:

- **Modify the kernel configuration:** Edit the file **android_rpi4_defconfig** in the directory /common/arch/arm64/configs/ directly within the kernel project.

- **Build the modified kernel:** Execute the following command to build the modified kernel: **'tools/bazel build –config=fast –config=stamp //com-**

mon:rpi4'. The resulting Image will be located in **'bazel-bin/common/rpi4/arch/arm64/boot'** directory.

- **Connect the SD card:** Connect the SD card into the Linux machine where is located the project.
- **Identify the boot partition on SD card:** Locate the partition corresponding to the /boot directory on the SD card.
- **Copy the modified kernel Image:** Execute the following command to copy the modified kernel to the SD card: **'cp bazel-bin/common/rpi4/arch/arm64/boot/Image /boot**.
- **Verify the kernel Image update:** Ensure that the Image file in the directory has been updated.
- **Reconnect the SD card and boot the Raspberry:** Reinsert the SD card into the Raspberry and verify that it boots correctly.
- **Verify kernel configurations:** Access the device shell and control the **config.gz** file in the /proc directory to confirm the applied configurations.

Below are the main configurations removed, divided by category, with the reasons behind these choices:

- **Input Device Drivers**
  - **CONFIG_JOYSTICK_XPAD, CONFIG_JOYSTICK_XPAD_FF, CONFIG_JOYSTICK_XPAD_LEDS, CONFIG_INPUT_JOYDEV, CONFIG_INPUT_TABLET, CONFIG_TOUCHSCREEN_USB_E2I and CONFIG_TOUCHSCREEN_USB_JASTEC:** These configurations enable support for Xbox controllers, joysticks, tablets, and touchscreen devices, which are not needed for the project. Disabling them reduces unnecessary driver loading and initialization during boot.
- **IRQ subsystem**
  - **CONFIG_GENERIC_IRQ_INJECTION and CONFIG_IRQ_DEBUGFS:** These configurations enable generic interrupt request (IRQ) injection functionality and debug filesystem support for IRQs, mainly used for testing or debugging. As they are not needed for production environment, they have been disabled to improve performance.

  - **CONFIG_CONTEXT_TRACKING:** This configuration enables the tracking of the context switches in the kernel. This option adds overhead to the kernel and is not necessary for project.

- **Power management options**
  - **CONFIG_PM_DEBUG and CONFIG_PM_SLEEP_DEBUG:** These configurations support debugging for the power management framework. These features are not necessary in a production environment, so they have been disabled.

- **Debug Options**
  - **CONFIG_CIFS_DEBUG, CONFIG_DEBUG_KERNEL, CONFIG_DEBUG_INFO_NONE and CONFIG_DEBUG_MISC:** These configurations support debugging options. These features are not necessary in production environment, so they have been disabled.

- **Tracing options**
  - **CONFIG_TRACER_MAX_TRACE, CONFIG_FUNCTION_TRACER, CONFIG_FUNCTION_GRAPH_TRACER, CONFIG_DYNAMIC_FTRACE, CONFIG_DYNAMIC_FTRACE_WITH_REGS and CONFIG_STACK_TRACER** These configurations enable various tracing features that are useful for detailed performance analysis and debugging. However, they are not required for production environment and contribute to kernel bloat and overhead, thus they have been disabled.

- **Network Device Support and Vendor Drivers**
  - **CONFIG_NET_VENDOR_NVIDIA, CONFIG_NET_VENDOR_ AMAZON, CONFIG_NET_VENDOR_ALTEON and CONFIG_NET_VENDOR_SOLARFLARE:** These configurations enable support for various network devices and vendor-specific drivers. Since project does not require these specific network device drivers, they have been disabled to reduce the kernel size and improve boot time.

- **USB network device support**
  - **CONFIG_USB_ALI_M5632, CONFIG_USB_BELKIN, CONFIG_USB_LAN78XX and CONFIG_USB_KC2190:** These configurations enable support for various USB network devices and chipsets.

Since project does not require these specific USB devices, they have been disabled to save resources and reduce kernel bloat.

- **Leds Triggers**
  - **CONFIG_LEDS_TRIGGER_ONESHOT, CONFIG_LEDS_ TRIGGER_CPU and CONFIG_LEDS_TRIGGER_INPUT:** These configurations enable LED triggers that are unnecessary for the project's requirements, thus disabled.
- **PWM options**
  - **CONFIG_PWM_BRCMSTB, CONFIG_PWM_BCM2835 and CONFIG_PWM_RASPBERRYPI_POE:** These configurations enable Pulse Width Modulation (PWM) support on various platforms in the Linux kernel. Since PWM functionality is not required for project, these options have been disabled.

These configurations represent a selection aimed at improving the efficiency and resource utilization of the system, but they do not encompass all configurations that were disabled.
This approach underscores the importance of targeted kernel optimization to achieve optimal performance tailored to specific operational needs. It is important to note that these adjustments are specific to the current case study. In other contexts, such as a car infotainment system, certain disabled configurations could be essential.

4.6 Compression of Kernel Image

The last phase of the study focused on analyzing the compression algorithm of the Kernel Image. Compressing the kernel image can significantly reduce the boot time by decreasing the time required to load into memory during booting. This section details the methodology used to analyze different compression algorithms. In particular, the different compression algorithms were benchmarked in order to determine the best algorithm regarding compression speed, decompression speed, and compression ratio.

4.6.1 Benchmark Kernel Compression Algorithms

To evaluate the impact of different compression algorithms on the boot time, a series of tests were conducted using various commonly used compression techniques. Each algorithm was applied to compress the kernel image using the **'tar'** utility. The 'tar' command, which stands for "tape archiver", is commonly used in compression on GNU/Linux. This utility supports various compression algorithms through command-line options to compress and decompress archives efficiently. To create a compressed file using 'tar', the command **'tar -cf'** is executed along with a flag that represents the desired compression algorithm, followed by the files to be included. The standard flags for compression are: The following results were obtained by combining the **'time'**

Long option	Algorithm
–gzip	gzip
–bzip2	bzip2
–xz	xz
-I"lz4"	lz4
–lzma	lzma
–zstd	zstd

Table 4.1. Compression Algorithms and their flags.

command with the **'tar'** command to measure the duration of the tar operation. The benchmarking results offer insights into the efficiency of each compression algorithm, aiding in the selection of the optimal method for reducing boot time while ensuring a good balance of speed and compression efficiency.

Algorithm	Time (s)	Size	Command
gzip	1.015	11907883	**time tar c –gzip -f Image.tar.gzip Image**
bzip2	1.981	11143162	**time tar c –bzip2 -f Image.tar.bzip2 Image**
lz4	0.185	15287271	**time tar c -I"lz4" -f Image.tar.lz4 Image**
xz	10.741	8444332	**time tar c –xz -f Image.tar.xz Image**
lzma	10.675	8444332	**time tar c –lzma -f Image.tar.lzma Image**
zstd	0.228	12131811	**time tar c –zstd -f Image.tar.zstd**

Table 4.2. Performance of Various Compression Algorithms.

For each algorithm used, the real time (obtained from the **time** command), the size of the resulting compressed file, and the command used to perform the compression were recorded and analyzed. The exact numbers obtained will vary depending on CPU, number of cores, and SSD/HDD speed, but the relative performance differences are expected to be somewhat similar [8].
The kernel must be decompress, making the decompression speed a critical factor in selecting the algorithm, as it adds additional time to the boot process. In the same way as compression, for each algorithm used, the real time and the command used to perform the decompression of the previous compressed files were recorded and analyzed.

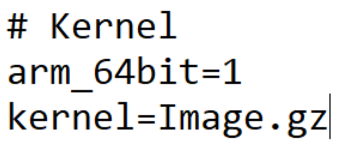
The benchmarking results demonstrate significant variations among the tested compression algorithms in terms of both compression and decompression speeds, as well as

Algorithm	Time (s)	Command
gzip	0.237	**time tar x –gzip -f Image.tar.gzip**
bzip2	0.956	**time tar x –bzip2 -f Image.tar.bzip2**
lz4	0.165	**time tar x -I"lz4" -f Image.tar.lz4**
xz	0.593	**time tar x –xz -f Image.tar.xz**
lzma	0.564	**time tar x –lzma -f Image.tar.lzma**
zstd	0.142	**time tar x –zstd -f Image.tar.zstd**

Table 4.3. Performance of Various Decompression Algorithms.

resulting file sizes. Each algorithm exhibit distinct advantages depending on specific performance requirements. For instance, lz4 stands out for its exceptional compression and decompression speeds, but with a larger compressed file size compared to some alternatives like xz and lzma. Alternatively, zstd offers a compelling balance of compression ratio and speed, making it a promising candidate for minimizing kernel image load times during system boot. These findings provide valuable insights for optimizing the kernel image compression process to enhance overall system performance, focusing on achieving the most efficient balance between compression efficiency and operational speed.
In conclusion, it is important to note that the benchmarking results serve primarily as a demonstration exercise. Therefore it is essential to conduct extensive testing of project to accurately assess how each algorithm affects the overall performance of the system, particularly in terms of reducing system boot time.
In the context of this project, it's important to note that although LZ4 and ZSTD show better performance in compression and decompression speeds, only GZIP is available for utilization.
To compress the kernel image with the GZIP algorithm, the following steps are performed:

- **Connect the SD card:** Insert the SD card into the Linux machine.
- **Identify the boot partition on SD card:** Locate the partition corresponding to the /boot directory on the SD card.
- **Compress the Image file:** Execute the command **'gzip Image'** to create the compressed file **Image.gz**.
- **Modify the file config.txt:** Open the file **'config.txt'** and modify the line related to the kernel from **"kernel=Image"** to **"kernel=Image.gz"**. This modification is necessary to instruct the bootloader to use the compressed kernel Image. The modified section of the file should look like this:



```
# Kernel
arm_64bit=1
kernel=Image.gz
```

Figure 4.8. Section related to the kernel Image.

- **Reconnect the SD card and boot the Raspberry:** Reinsert hte SD card and verity that it boots correctly.

While GZIP may not offer the best performance compared to LZ4 or ZSTD, its compatibility and availability make it the practical choice for the project. This approach underscores the balance between achieving optimal performance and practical implementation constraints, ensuring that the system operates reliably within the specified parameters.

Chapter 5

Results

5.1 Analysis of results

This section presents a detailed analysis of the results using selected Key Performance Indicators (KPIs), comparing the initial project state with the optimized state. The analysis provides a comprehensive view of the improvements achieved in terms of boot time for each phase of the optimization. Each subsection includes averaged results of system boot times, derived from 20 measurements taken at system reboot to ensure accuracy.
Below is the initial project state, accompanied by a descriptive table of various KPIs:

KPI	**Log Message**	**Time (s)**
1	I/HidlServiceManagement:Registered android.hardware.automotive	00:08.797
2	I/HidlServiceManagement:Registered android.hardware.Bluetooth	00:15.289
3	I/HidlServiceManagement:Registered android.hardware.usb.gadget	00:15.290
4	I/Android.hardware.camera.provider-V1-external-service	00:16.008
5	I/HidlServiceManagement:Registered android.hardware.audio	00:16.847
6	I/HidlServiceManagement:Registered android.hardware.media	00:17.473
7	D/BootAnimation:BootAnimationStartTiming start time	00:21.441

Continue on the next page

KPI	Log Message	Time (s)
8	D/SystemServerTiming:StartPowerManager	00:32.810
9	D/SystemServerTiming:StartTelephonyRegistry	00:50.982
10	D/SystemServerTiming:StartWifi	00:54.756
11	D/SystemServerTiming:StartSystemUpdateManagerService	00:54.938
12	D/SystemServerTiming:StartNotificationManager	00:54.944
13	D/SystemServerTiming:StartLocationManagerService	00:55.150
14	D/SystemServerTiming:StartSystemUI	01:03.757
15	D/BootAnimation:BootAnimationStopTiming	01:20.820

Table 5.1. Android KPIs Performance of the AOSP full project.

The table above provides a comprehensive list of Key Performance Indicators (KPIs) for the initial phase of the AOSP (Android Open Source Project) full project. Each row corresponds to a critical event or log message throughout the system's boot sequence, paired with the specific time in seconds when the event took place.

5.1.1 Removing unnecessary projects

The initial optimization phase resulted in a substantial reduction in the system's boot time, significantly enhancing overall system speed. The results obtained from this optimization are presented below:

KPI	Log Message	Time (s)
1	I/HidlServiceManagement:Registered android.hardware.automotive	00:07.629
2	I/HidlServiceManagement:Registered android.hardware.Bluetooth	00:12.786
3	I/HidlServiceManagement:Registered android.hardware.usb.gadget	00:12.790
4	I/Android.hardware.camera.provider-V1-external-service	00:13.314
5	I/HidlServiceManagement:Registered android.hardware.audio	00:13.844
6	I/HidlServiceManagement:Registered android.hardware.media	00:14.418
7	D/BootAnimation:BootAnimationStartTiming start time	00:17.054
8	D/SystemServerTiming:StartPowerManager	00:28.342
9	D/SystemServerTiming:StartTelephonyRegistry	00:35.203
10	D/SystemServerTiming:StartWifi	00:36.734
11	D/SystemServerTiming:StartSystemUpdateManagerService	00:36.734
12	D/SystemServerTiming:StartNotificationManager	00:37.041
13	D/SystemServerTiming:StartLocationManagerService	00:37.140
14	D/SystemServerTiming:StartSystemUI	00:41.683
15	D/BootAnimation:BootAnimationStopTiming	00:53.477

Table 5.2. Android KPIs Performance after removing unnecessary projects.

The optimization led to a more efficient boot process, with all key components initiating more rapidly than before. This stage of optimization underscores the significance of removing superfluous projects, resulting in a streamlined and faster boot sequence.

5.1.2 Removing Initial Animation

Removing the initial boot animation further streamlined the boot process. Although boot animations are aesthetically pleasing, they extend the startup time by consuming system resources. By removing this feature, the system can redirect resources to other processes, significantly reducing the total startup time. The results from this optimization phase are presented below:

KPI	**Log Message**	**Time (s)**
1	I/HidlServiceManagement:Registered android.hardware.automotive	00:07.578
2	I/HidlServiceManagement:Registered android.hardware.Bluetooth	00:12.640
3	I/HidlServiceManagement:Registered android.hardware.usb.gadget	00:12.655
4	I/Android.hardware.camera.provider-V1-external-service	00:13.260
5	I/HidlServiceManagement:Registered android.hardware.audio	00:13.794
6	I/HidlServiceManagement:Registered android.hardware.media	00:14.343
7	D/BootAnimation:BootAnimationStartTiming start time	00:16.902
8	D/SystemServerTiming:StartPowerManager	00:27.963
9	D/SystemServerTiming:StartTelephonyRegistry	00:34.633
10	D/SystemServerTiming:StartWifi	00:36.141
11	D/SystemServerTiming:StartSystemUpdateManagerService	00:36.424
12	D/SystemServerTiming:StartNotificationManager	00:36.429

Continue on the next page

KPI	Log Message	Time (s)
13	D/SystemServerTiming:StartLocationManagerService	00:36.517
14	D/SystemServerTiming:StartSystemUI	00:40.896
15	D/BootAnimation:BootAnimationStopTiming	00:47.222

Table 5.3. Android KPIs Performance after removing Initial Animation.

This optimization step highlights the importance of removing non-essential visual elements to achieve a faster and more efficient boot process.

5.1.3 Configuration of new version of Kernel

Updating and configuring a new version of the kernel brought several improvements to the system's performance and boot time. The kernel is the core component of the operating system, managing hardware resources and system processes. Upgrading to a newer version introduced optimizations and enhancements that were not present in the older version. The results from this optimization phase are presented below:

KPI	Log Message	Time (s)
1	I/HidlServiceManagement:Registered android.hardware.automotive	00:07.165
2	I/HidlServiceManagement:Registered android.hardware.Bluetooth	00:11.378
3	I/HidlServiceManagement:Registered android.hardware.usb.gadget	00:11.379
4	I/Android.hardware.camera.provider-V1-external-service	00:11.871
5	I/HidlServiceManagement:Registered android.hardware.audio	00:12.470
6	I/HidlServiceManagement:Registered android.hardware.media	00:12.884

Continue on the next page

KPI	Log Message	Time (s)
7	D/BootAnimation:BootAnimationStartTiming start time	00:15.378
8	D/SystemServerTiming:StartPowerManager	00:25.422
9	D/SystemServerTiming:StartTelephonyRegistry	00:31.292
10	D/SystemServerTiming:StartWifi	00:32.509
11	D/SystemServerTiming:StartSystemUpdateManagerService	00:32.732
12	D/SystemServerTiming:StartNotificationManager	00:32.737
13	D/SystemServerTiming:StartLocationManagerService	00:32.826
14	D/SystemServerTiming:StartSystemUI	00:36.453
15	D/BootAnimation:BootAnimationStopTiming	00:41.867

Table 5.4. Android KPIs Performance after configuring the new version of the Kernel.

5.1.4 Disabling Serial Console

Disabling the serial console was a strategic optimization aimed at improving system boot performance. While the serial console is useful for debugging, its continuous operation during boot can introduce delays, especially on systems where debugging over serial is not actively required. This target optimization is useful for the bottleneck problem analyzed above. The results from this optimization phase are presented below:

KPI	Log Message	Time (s)
1	I/HidlServiceManagement:Registered android.hardware.automotive	00:04.109

Continue on the next page

KPI	Log Message	Time (s)
2	I/HidlServiceManagement:Registered android.hardware.Bluetooth	00:07.612
3	I/HidlServiceManagement:Registered android.hardware.usb.gadget	00:07.613
4	I/Android.hardware.camera.provider-V1-external-service	00:08.045
5	I/HidlServiceManagement:Registered android.hardware.audio	00:08.618
6	I/HidlServiceManagement:Registered android.hardware.media	00:09.097
7	D/BootAnimation:BootAnimationStartTiming start time	00:11.559
8	D/SystemServerTiming:StartPowerManager	00:21.396
9	D/SystemServerTiming:StartTelephonyRegistry	00:27.281
10	D/SystemServerTiming:StartWifi	00:28.444
11	D/SystemServerTiming:StartSystemUpdateManagerService	00:28.628
12	D/SystemServerTiming:StartNotificationManager	00:28.633
13	D/SystemServerTiming:StartLocationManagerService	00:28.718
14	D/SystemServerTiming:StartSystemUI	00:32.297
15	D/BootAnimation:BootAnimationStopTiming	00:37.739

Table 5.5. Android KPIs Performance after disabling serial console.

5.1.5 Disabling unnecessary Kernel options

The Linux kernel offers a wide array of configuration options, many of which may not be essential for the specific hardware or use case of the system. By carefully reviewing and disabling these non-essential options, the initialization process of the kernel was made more efficient. The results of this optimization phase are shown below:

KPI	Log Message	Time (s)
1	I/HidlServiceManagement:Registered android.hardware.automotive	00:03.820
2	I/HidlServiceManagement:Registered android.hardware.Bluetooth	00:07.267
3	I/HidlServiceManagement:Registered android.hardware.usb.gadget	00:07.270
4	I/Android.hardware.camera.provider-V1-external-service	00:07.717
5	I/HidlServiceManagement:Registered android.hardware.audio	00:08.264
6	I/HidlServiceManagement:Registered android.hardware.media	00:08.724
7	D/BootAnimation:BootAnimationStartTiming start time	00:11.119
8	D/SystemServerTiming:StartPowerManager	00:20.991
9	D/SystemServerTiming:StartTelephonyRegistry	00:26.755
10	D/SystemServerTiming:StartWifi	00:27.901
11	D/SystemServerTiming:StartSystemUpdateManagerService	00:28.053
12	D/SystemServerTiming:StartNotificationManager	00:28.074
13	D/SystemServerTiming:StartLocationManagerService	00:28.156
14	D/SystemServerTiming:StartSystemUI	00:31.655
15	D/BootAnimation:BootAnimationStopTiming	00:36.826

Table 5.6. Android KPIs Performance after disabling unnecessary Kernel options.

5.1.6 Compression of Kernel Image

The final phase of kernel compression optimization using the GZIP algorithm did not yield significant improvements to the system. This outcome was primarily due to the unavailability of more efficient compression algorithms suitable for our project's requirements. While GZIP compression reduced the size of the kernel image, the impact on system boot performance was minimal. The results from this phase of optimization are presented below:

KPI	Log Message	Time (s)
1	I/HidlServiceManagement:Registered android.hardware.automotive	00:03.709
2	I/HidlServiceManagement:Registered android.hardware.Bluetooth	00:07.130
3	I/HidlServiceManagement:Registered android.hardware.usb.gadget	00:07.133
4	I/Android.hardware.camera.provider-V1-external-service	00:07.534
5	I/HidlServiceManagement:Registered android.hardware.audio	00:08.132
6	I/HidlServiceManagement:Registered android.hardware.media	00:08.570
7	D/BootAnimation:BootAnimationStartTiming start time	00:11.074
8	D/SystemServerTiming:StartPowerManager	00:20.862
9	D/SystemServerTiming:StartTelephonyRegistry	00:26.610
10	D/SystemServerTiming:StartWifi	00:27.760
11	D/SystemServerTiming:StartSystemUpdateManagerService	00:27.908
12	D/SystemServerTiming:StartNotificationManager	00:27.924
13	D/SystemServerTiming:StartLocationManagerService	00:28.012

Continue on the next page

KPI	Log Message	Time (s)
14	D/SystemServerTiming:StartSystemUI	00:31.507
15	D/BootAnimation:BootAnimationStopTiming	00:36.524

Table 5.7. Android KPIs Performance after kernel image compression.

5.2 Comparison with initial system performance

To evaluate the effectiveness of the optimizations, the KPI **D/BootAnimation:BootAnimationStopTiming** message at the end of the boot process was compared with the results obtained after implementing all optimizations. The following graph illustrates the comparative analysis:

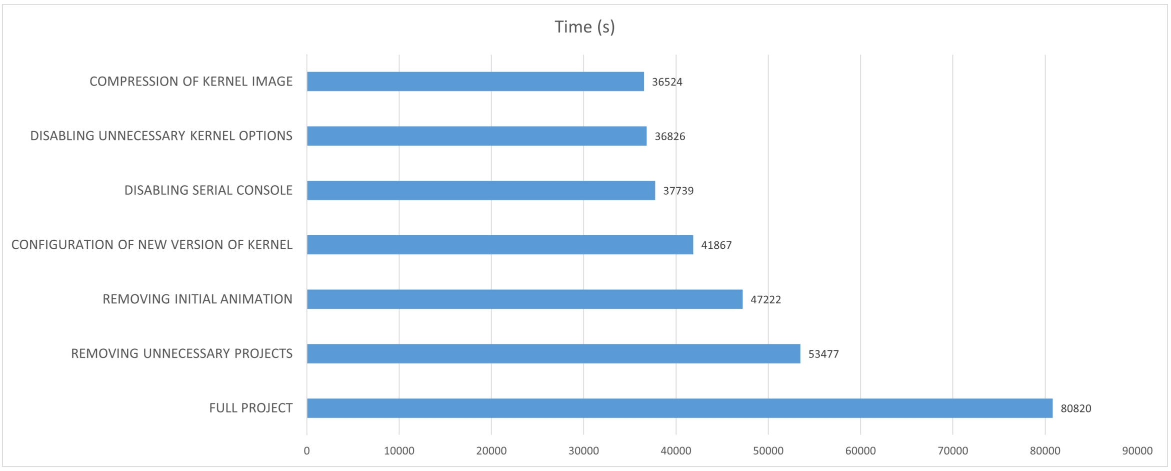


Figure 5.1. Comparison of Boot before and after optimizations.

As illustrated in the graph, the series of optimizations implemented led to a significant reduction in boot time, demonstrating the effectiveness of each optimization phase. The initial system performance, represented by the longest boot time, was systematically improved through targeted interventions, including removing unnecessary projects, disabling initial animations, updating the kernel, disabling the serial console, and disabling unnecessary kernel options. Despite the minimal impact of kernel image compression, the overall enhancements cumulatively resulted in a more efficient and faster boot process. These optimizations not only highlight the importance of each step but also emphasize the potential for further improvements with the availability of more advanced algorithms and tools.

Chapter 6

Conclusion

The aim of this thesis work was to optimize the boot time of an Android system through a series of targeted optimizations. The optimizations implemented have led to a reduction of more than 50% in the startup time of the system, making it much more efficient and faster. Each key performance indicator (KPIs) was meticulously monitored to track the impact of each optimization step. The goal of this thesis was to demonstrate that starting from the operating system, optimization is a clear and precise methodology with guidelines that can be implemented to make the system as fast as possible. It is essential to underline that this methodology can be used in any operating system and with any available board. This thesis work serves as an example of how this methodology has been implemented. The lack of official documentation regarding the open source project led to several challenges, such as the disabling of kernel configurations. Some configurations, which at first appeared non-essential for our project, rendered the system unbootable when disabled. This highlights the complexity of system, emphasizing the need of a comprehensive documentation of the project. The presence of a single kernel compression algorithm, GZIP, led to a minimal improvements in boot time. This limitation underscores the importance of having access to a different compression algorithms. Future works could explore the implementation of alternative algorithm, such as LZ4 and ZSTD, which are known for their better performance. Implementing these algorithms could potentially produce significant reductions in boot time.

Further possible implementations could include using a more powerful board, such as a Raspberry Pi 5, to monitor and compare the results obtained. These future developments could provide even greater insights and improvements in system performance. The methodology devised and utilized in this thesis can be adapted to additional facets of system performance beyond boot time, including the optimization of memory usage and power consumption. These domains present prospects for future research and advancements, furthering the enhancement of system performance. Overall, this thesis demonstrated that substantial improvements in system boot time can be achieved through targeted optimizations. By addressing the challenges encountered and exploring future enhancements, further advancements can be made, contributing to more efficient and responsive Android systems. The methodologies presented in this thesis work provide a base for ongoing research and development in the field of system performance optimization. With continued effort even grater improvements can be achieved, benefiting a wide range of applications and industries.

Bibliography

[1] *5 Futuristic In-Vehicle Infotainment Features in the Age of Software-Defined Vehicles.* URL: https://autocrypt.io/5-futuristic-in-vehicle-infotainment-features/.

[2] *A brief history of in-vehicle infotainment and car data storage.* URL: https://www.tuxera.com/blog/a-brief-history-of-in-vehicle-infotainment-how-tuxera-fits-in/.

[3] *AI in Automotive Infotainment: Revolutionizing the Driving Experience.* URL: https://datajob.se/ai-in-automotive-infotainment-revolutionizing-the-driving-experience/.

[4] *Android Architecture.* URL: https://www.geeksforgeeks.org/android-architecture/.

[5] *Android Boot Process.* URL: https://www.geeksforgeeks.org/android-boot-process/.

[6] *Android Developers.* URL: https://developer.android.com/tools/adb.

[7] *Android Hardware Abstraction Layer (HAL).* URL: https://medium.com/@dugguRK/about-android-hardware-abstraction-layer-hal-5d191dafeb2c.

[8] *Comparison of Compression Algorithms.* URL: https://linuxreviews.org/Comparison_of_Compression_Algorithms.

[9] *config.txt.* URL: https://www.raspberrypi.com/documentation/computers/config_txt.html.

[10] *Datasheets Raspberry PI4.* URL: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf.

[11] *Designing in-vehicle infotainment systems.* URL: https://my.avnet.com/abacus/solutions/markets/automotive-and-transportation/automotive/comfort-infotainment-and-safety/automotive-infotainment/.

[12] *DS Automobiles integra Chatgpt.* URL: https://www.dsautomobiles.it/brand-ds/scopri-marchio-ds/news/ds-automobiles-chatgpt.html.

[13] *Everything You Need to Know About In-Vehicle Infotainment Systems.* URL: https://www.einfochips.com/blog/everything-you-need-to-know-about-in-vehicle-infotainment-system/.

[14] *Evolution of In-Car Infotainment systems.* URL: https://www.exhibit.tech/auto-tech/evolution-of-in-car-infotainment-systems.

[15] *Google Devices.* URL: https://wiki.lineageos.org/devices/.

[16] *Honda praised for its Electro Gyrocator, first map-based nav system.* URL: https://www.motor1.com/news/137950/honda-electro-gyrocator-ieee-milestone/.

[17] *In-Vehicle infotainment launches a new situation for Internet of Vehicle applications.* URL: https://www.arrow.com/en/research-and-events/articles/in-vehicle-infotainment-launches-a-new-situation-for-internet-of-vehicle-applications.

[18] *Introduction to Android Development.* URL: https://www.geeksforgeeks.org/introduction-to-android-development/?ref=lbp.

[19] *Logcat.* URL: https://developer.android.com/tools/logcat.

[20] *Motorola solutions.* URL: https://www.motorolasolutions.com/en_us/about/history/explore-motorola-heritage/sound-motion.html.

[21] *Platform architecture.* URL: https://developer.android.com/guide/platform?hl=it.

[22] *Raspberry Pi 4 Model B.* URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[23] *Raspberry Vanilla.* URL: https://github.com/raspberry-vanilla.

[24] *Raspberry vanilla Kernel.* URL: https://github.com/raspberry-vanilla/android_kernel_manifest/tree/android-14.0?tab=readme-ov-file.

[25] *Set up for AOSP development.* URL: https://source.android.com/docs/setup/start/requirements.

[26] *Sync: 2007's Most Important Car Technology.* URL: https://au.pcmag.com/gallery/17050/sync-2007s-most-important-car-technology?p=1.

[27] *What is POST(Power-On-Self-Test)?* URL: https://www.geeksforgeeks.org/what-is-postpower-on-self-test/.