POLITECNICO DI TORINO

Master's Degree in Mechatronics engineering

Specialization in: Software Technologies for Automation

MASTER THESIS

Migration from Monolithic Algorithms to Service-Oriented Architectures in Software-Defined Vehicles

Supervisor

Prof. Massimo Violante

Prof. Jacopo Sini

Candidate

Dhanesh Kanwar

Academic Year: 2024-25

Abstract

The increasing complexity and software centrality of modern vehicles have necessitated a shift from traditional monolithic software architectures to modular, scalable paradigms. This thesis explores the migration from Monolithic Algorithms to Service-Oriented Architectures (SOA) in Software-Defined Vehicles (SDVs), using a practical case study: the design and simulation of an Adaptive Light Beam Controller (ALBC) in Simulink. Both monolithic and SOA-based architectures are developed and evaluated, with a comparative analysis focusing on modularity, maintainability, over-the-air (OTA) update readiness, and cross-domain integration.

To validate the practical feasibility of these architectures, Embedded Coder is used to generate C++ code from both Simulink models. The generated code is successfully integrated into an Android Studio environment and deployed on an automotive emulator, simulating real-time execution in an in-vehicle context. The results demonstrate that SOA not only enhances software modularity and service independence but also significantly improves deployment flexibility and future maintainability. This work provides a comprehensive methodology for transitioning automotive control systems to SOA using model-based design, automated code generation, and rapid prototyping on virtual platforms.

Contents

Chapter 1

1. Introduction

1.1 Background and Motivation

- Software in modern vehicles has grown from basic control systems to complex, interconnected platforms enabling autonomous features, infotainment, connectivity, and adaptive behaviours. [4]
- Software-Defined Vehicles (SDVs) represent a paradigm shift where vehicle functions are increasingly implemented, updated, and controlled via software. [4]
- Traditionally, automotive software systems have been monolithic, where each functionality is deeply integrated and dependent on specific hardware (ECUs), making updates and scaling difficult.
- The demand for flexible, modular, and update-ready architectures has led to the adoption of Service-Oriented Architecture (SOA) in the automotive domain.[4]
- The motivation is to study the benefits of SOA in practice, specifically through modelling and implementation of an Adaptive Light Beam Controller (ALBC) system in both monolithic and SOA formats, using Simulink and Embedded Coder, and testing deployment feasibility using Android Studio on an automotive emulator.

1.2 Problem Statement

- Monolithic software architectures suffer from:
    - Lack of modularity and reuse
    - Difficulties in OTA (over-the-air) updates
    - Increased complexity in testing and validation
    - Long development cycles when modifying or extending systems
- There is a lack of practical implementation studies showing the migration process to SOA in automotive systems using industry tools.
- Key problem: How can we practically migrate a monolithic automotive function to a service-oriented model, and what are the measurable benefits in doing so?

1.3 Research Objectives

The main goals of the thesis are:

- To design and simulate an Adaptive Light Beam Controller (ALBC) in both monolithic and SOA styles using Simulink.

- To generate C++ code using Embedded Coder from both architectural models.
- To integrate and deploy this code in **Android Studio** on an **automotive emulator** for real-time execution testing.
- To compare the architectures using criteria such as:
    - Modularity
    - Maintainability
    - OTA readiness
    - Code integration and reusability
- To propose a reference workflow for migrating legacy monolithic automotive functions to a SOA-based implementation.

1.4 Methodology Overview

- Modelling: Develop the ALBC logic in Simulink under two architectural paradigms:
    - Tightly integrated monolithic
    - Loosely coupled service-oriented
- Code Generation: Use Embedded Coder to export both designs into C++ code.
- Integration: Import the generated code into Android Studio, wrap with JNI if needed, and simulate in an Android-based automotive emulator.
- Evaluation: Measure performance, modularity, update complexity, and service isolation.

1.5 Thesis Contributions

This work contributes:

- A side-by-side modelling and implementation of monolithic and SOA versions of a real-world automotive function.
- Demonstration of Embedded Coder's integration with Android-based environments.
- A deployment workflow from Simulink → C++ → Android Studio → Emulator.
- A comparison framework for evaluating monolithic vs SOA designs.
- Insights into real-world challenges and benefits of SOA migration in SDVs.

Chapter 2

2. Literature Review

2.1 Software-Defined Vehicles (SDVs): A Paradigm Shift

The automotive industry is transitioning from hardware-centric engineering to a **software-defined approach**, where vehicle behaviour and capabilities are increasingly controlled by software. A Software-Defined Vehicle (SDV) decouples functionality from hardware through abstraction layers and centralized computing. This enables manufacturers to:

- Roll out new features via **over-the-air (OTA)** updates
- Reduce time-to-market
- Adapt vehicle behaviour dynamically based on data and context

Key industry players (e.g., Tesla, Volkswagen, Toyota, BMW) are investing in SDV platforms built around centralized electronic architectures and zonal computing units. This trend has driven the need for **flexible, modular, and update-ready software architectures**, such as **SOA**. [1][4]

2.2 Monolithic Architectures in Automotive Systems

Historically, vehicle functions have been developed in **monolithic architectures**:

- Code is organized as **tightly coupled modules**
- Functions are often embedded directly into hardware (ECUs)
- Dependencies between modules are high
- Updates typically require complete revalidation or hardware flashing

While monolithic designs were effective for early embedded systems, they suffer from several limitations in the SDV era:

Aspect	**Monolithic Limitation**
Modularity	Difficult to isolate or reuse components
Maintainability	High impact of small changes
OTA updates	Rarely supported; complex and risky
Scalability	Hard to extend due to rigid structure
Testing	Time-consuming due to integration dependencies

This has led to a growing consensus that **legacy monolithic systems must evolve** toward **modular and service-oriented** paradigms.

2.3 Service-Oriented Architecture (SOA) in Automotive

SOA is a software architecture paradigm that structures applications as **loosely coupled, independently deployable services**. Each service performs a specific function and communicates through well-defined interfaces (e.g., APIs or middleware). [1][4]

Benefits of SOA in SDVs:

- **Modularity**: Services can be developed and tested independently.
- **Scalability**: New features can be added without modifying the core system.
- **OTA Readiness**: Individual services can be updated dynamically.
- **Cross-Domain Communication**: Facilitates interaction between powertrain, infotainment, ADAS, and other domains.

SOA Standards in Automotive:

- **AUTOSAR Adaptive Platform**: An industry standard supporting POSIX-based OS, service discovery, and dynamic deployment.
- **DDS (Data Distribution Service)** and **SOME/IP**: Common middleware protocols for real-time communication in SOA-based ECUs.
- **ISO 26262**: Safety standard requiring traceable and testable modules, which SOA supports.

Challenges:

- Real-time constraints and communication latency
- Migration of legacy monolithic code
- Integration complexity and toolchain compatibility
- Ensuring system safety and performance under distributed control

2.4 Model-Based Design and Simulink in Embedded Systems

Model-Based Design (MBD) is a design methodology where system functionality is captured in graphical models rather than textual code. **Simulink**, a widely adopted MBD tool by MathWorks, allows:

- Rapid prototyping of control algorithms
- Simulation of real-world systems and behaviour
- Automatic code generation for embedded deployment

In automotive engineering, Simulink enables system engineers to:

- Design functional blocks (sensors, controllers, actuators)
- Simulate system response under various scenarios
- Validate functional safety requirements before coding
- Reuse validated models across platforms

In thesis, Simulink is used to design the **Adaptive Light Beam Controller (ALBC)** in both monolithic and SOA styles.

2.5 Embedded Coder and C++ Code Generation

Embedded Coder is a MATLAB tool that extends Simulink to generate highly optimized and readable **C/C++ code** from models for embedded targets. Key features include:[16]

- **Code mapping** for model elements (inputs, outputs, functions)
- Integration with **software-in-the-loop (SIL)** and **processor-in-the-loop (PIL)** workflows.
- Generation of reusable software components aligned with **AUTOSAR**, **ISO 26262**, and other standards.
- Configuration of function interfaces for integration into target projects (e.g., Android, Linux-based ECUs).

For my project, Embedded Coder is used to:

- Export both monolithic and SOA-based ALBC models as C++ code
- Integrate them into Android Studio
- Deploy and run them on an automotive emulator

2.6 Android-Based Automotive Emulators

Android-based platforms are increasingly used for prototyping and simulating automotive applications. Tools like **Android Automotive OS (AAOS)** provide a real-time OS environment and UI for vehicle functions.[6]

Why Use Android Emulators?

- Safe, virtual testing environment for embedded software
- Support for C++ libraries via **JNI (Java Native Interface)**
- Integration with Android Studio for app development and deployment
- Emulates user interaction and service behaviour in an SDV-like environment

In this thesis, C++ code from Embedded Coder is compiled into Android-native code, deployed to the emulator, and validated against expected ALBC behaviour.

2.7 Summary and Research Gaps

This chapter highlighted the evolution from monolithic to SOA in automotive systems and the importance of model-based design and code generation in this transition.

Key Gaps Identified:

- A lack of **end-to-end case studies** demonstrating the **full pipeline** from model-based design → code generation → integration → emulator deployment.
- Limited academic examples showing **side-by-side comparisons** between monolithic and SOA implementations using industry-standard tools.
- Insufficient guidelines on how to structure SOA components in Simulink and deploy them as real-time services.

This thesis addresses these gaps by designing, implementing, generating, and deploying both monolithic and SOA versions of an automotive control function — the Adaptive Light Beam Controller (ALBC) — and evaluating their comparative merits in a simulated real-time environment.

Chapter 3

3. Design of Adaptive Light Beam Controller – Monolithic Architecture

3.1 Functional Requirements and Test Case Description

FR1: High Beam Mode

Requirement:

- The system shall activate High Beam mode when all the following conditions are met:
    - No oncoming vehicles detected (VehicleOncoming == 0)
    - Vehicle speed is greater than or equal to 50 km/h (VehicleSpeed >= 50)
    - Ambient light level is below 20 lux (AmbientLight < 20)
- When activated, the system shall:
    - Set beam range to 100 meters (BeamRange = 100)
    - Set beam angle to 0 degrees (BeamAngle = 0)

Test Cases:

- TC1.1: Verify High Beam activates when no oncoming vehicle, speed = 60 km/h, ambient light = 10 lux. Expect beam range = 100 m, beam angle = 0°
- TC1.2: Verify High Beam does NOT activate when an oncoming vehicle is detected, even if speed and light conditions are met.
- TC1.3: Verify High Beam does NOT activate if speed < 50 km/h regardless of other conditions.
- TC1.4: Verify High Beam does NOT activate if ambient light ≥ 20 lux regardless of other conditions.

FR2: Cornering Beam Mode

Requirement:

- The system shall activate Cornering Beam mode when the steering angle is greater than or equal to 10 degrees (SteeringAngle >= 10°)
- When activated, the system shall:
    - Set beam range to 50 meters (BeamRange = 50)
    - Adjust beam angle dynamically using a discrete controller based on road curvature (BeamAngle = BeamPID(RoadCurvature))

Test Cases:

- TC2.1: Verify Cornering Beam activates when steering angle = 15°, beam range is set to 50 m, and beam angle adjusts based on simulated road curvature input.
- TC2.2: Verify Cornering Beam does NOT activate when steering angle < 10°.
- TC2.3: Verify beam angle changes smoothly with varying road curvature inputs using discrete controller behaviour.

FR3: City Mode

Requirement:

- The system shall activate City Mode when:
  - Vehicle speed is less than or equal to 30 km/h (VehicleSpeed <= 30)
  - Ambient light level is above 30 lux (AmbientLight > 30)
- When activated, the system shall:
  - Set beam range to 30 meters (BeamRange = 30)
  - Set beam angle to 0 degrees (BeamAngle = 0)

Test Cases:

- TC3.1: Verify City Mode activates when vehicle speed = 20 km/h and ambient light = 40 lux.
- TC3.2: Verify City Mode does NOT activate if speed > 30 km/h even if ambient light is high.
- TC3.3: Verify City Mode does NOT activate if ambient light ≤ 30 lux even if speed is low.

FR4: Low Beam Mode

Requirement:

- The system shall activate Low Beam mode when an oncoming vehicle is detected (VehicleOncoming == 1)
- When activated, the system shall:
  - Set beam range to 50 meters (BeamRange = 50)
  - Set beam angle to 0 degrees (BeamAngle = 0)

Test Cases:

- TC4.1: Verify Low Beam activates when an oncoming vehicle is detected regardless of speed or ambient light.
- TC4.2: Verify Low Beam deactivates when no oncoming vehicle is detected.
- TC4.3: Verify beam range and angle remain at 50 meters and 0 degrees respectively while Low Beam mode is active.

3.2 Modelling of Adaptive Light Beam Controller in Simulink

The Adaptive Light Beam controller is developed using a Model-Based Systems Design (MBSD) approach within Simulink, leveraging Stateflow for state machine implementation. Each input signal, along with its corresponding measurement unit, is fed into the Stateflow chart to ensure clarity and proper handling of physical quantities (as illustrated in the accompanying figure).

The Stateflow chart consists of four distinct states: **LowBeam, HighBeam, CorneringBeam,** and **CityMode**. The system transitions between these states based on specific input conditions, maintaining a single active state at any given time. Within each state, output signals are generated and adjusted accordingly to control the adaptive lighting behaviour.

A key feature of the system is the **CorneringBeam** state, which goes beyond a simple state transition condition. When the steering angle exceeds a threshold value of 10 degrees, the beam angle dynamically adjusts to follow the road curvature. This adaptive behaviour is implemented through a discrete controller designed to modulate the beam angle based on the steering input, enabling enhanced visibility during cornering manoeuvres.

The design procedure of this discrete controller involves:

- Sampling the road curvature input at fixed intervals to compute the required beam angle adjustment.
- Implementing a control algorithm that maps the road curvature to a corresponding beam deflection angle, ensuring smooth and timely response.
- Incorporating saturation limits which is of +-60° and safety checks to prevent excessive beam movement and maintain driver safety.
- Validating the controller performance within the Simulink environment through simulation, confirming that the beam adjustment closely follows the vehicle’s steering dynamics.

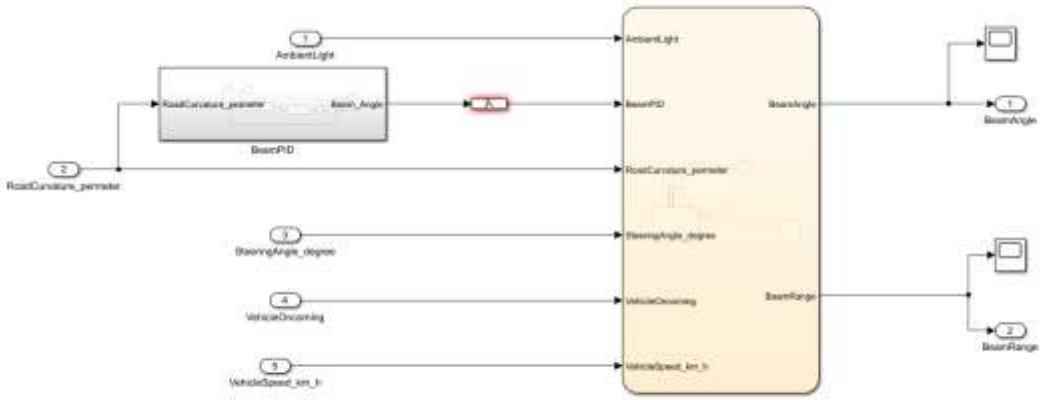


Figure 1: Monolithic ALBC Simulink design

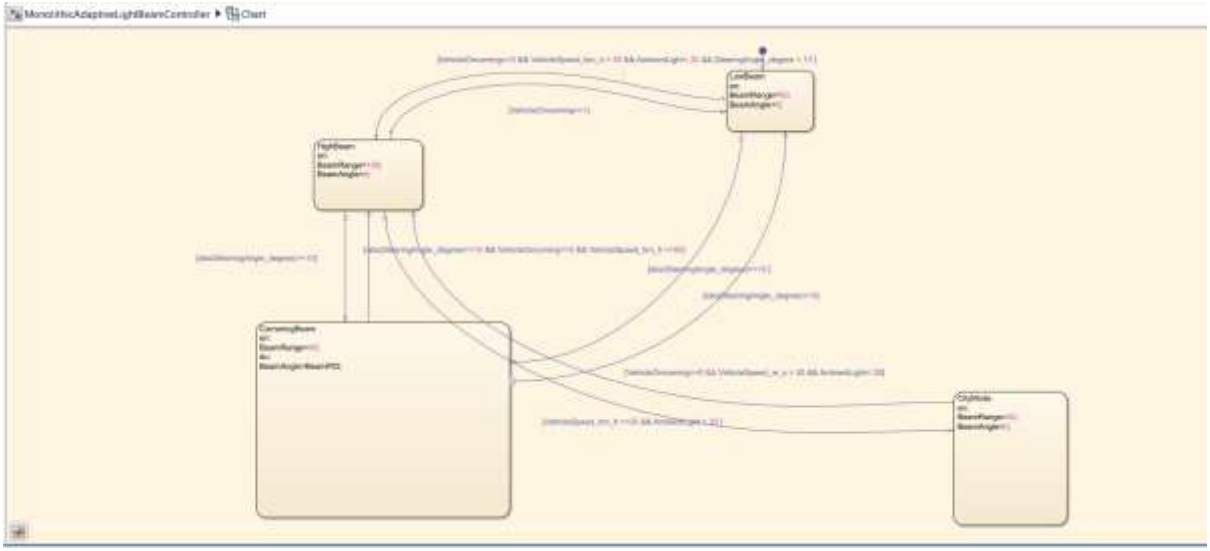


Figure 2: State chart corresponding to ALBC

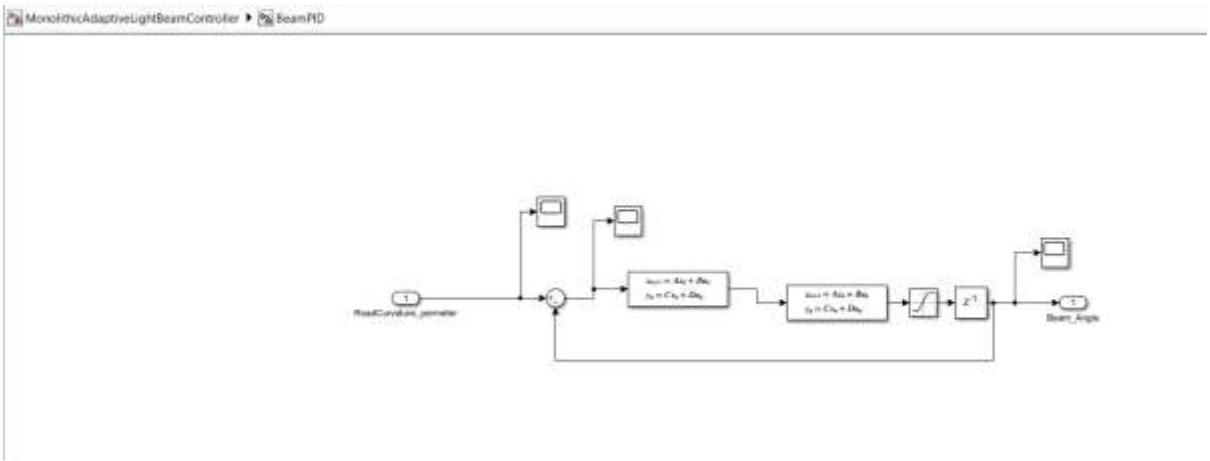


Figure 3: Beam discrete controller design

3.3 Discrete Controller Design Using the Diophantine Equation Approach

To achieve precise control of the beam angle in the **Cornering Beam** state, a discrete controller was designed following a Diophantine equation-based methodology. This approach enables the direct synthesis of a digital controller that meets specified dynamic performance criteria by solving polynomial equations in the discrete domain.

The continuous-time plant model representing the beam angle dynamics was initially defined as:

$$Gcont(s) = \frac{1}{0.1 * s^2 + 0.5 * s + 11}$$

This transfer function captures the relevant dynamics of the beam actuation system. The system was converted into its state-space representation for detailed analysis and controller synthesis.

Using a sampling time of Ts=0.01seconds, the system was discretized. Rather than relying solely on standard discretization methods, the discrete transfer function was manually derived and expressed as:

$$G(z) = 0.00049173(z + 0.9835)/(z^2 - 1.95z + 0.9512)$$

The poles and zeros of the discrete system were extracted and analysed to characterize the system's behaviour in the z-domain.

Performance specifications were defined based on overshoot (sovr=20%), settling time (tso=0.4t seconds), and rise time (tro=0.14t seconds). These parameters were used to calculate the damping ratio ζ and natural frequency ωn necessary to meet the design criteria:

$$\zeta = abs\left(\frac{\ln(sovr)}{sqrt(\pi^2 + \ln(sovr)^2)}\right)$$

where ωn,ts and ωn,tr are derived from settling time(ts) and rise time(tr) constraints respectively.

$$wn, ts = \frac{4.6}{(ts * \zeta)}$$

$$wn.tr = \frac{(\pi - \operatorname{acos}(\zeta))}{tr * \sqrt{(1} - \zeta^2)}$$

The core of the controller design involved solving the Diophantine equation to determine controller polynomials R(z) and S(z) such that the closed-loop characteristic polynomial matched the desired dynamics. This was done by constructing Sylvester matrices from the plant polynomials and solving for the unknown coefficients.

The resulting controller transfer function $C(z) = \frac{S(z)}{R(z)}$ was simplified and converted to state-space form for implementation and simulation within Simulink.

Key steps included:

- Extracting discrete system zeros and poles for polynomial formation.
- Constructing and solving the Sylvester matrix equation representing the Diophantine condition.
- Designing the controller to place closed-loop poles inside a cardioid region defined by the damping ratio ζ, ensuring system stability and desired transient response.
- Simulating the closed-loop response in Simulink to validate performance, with plots of error, reference, and output signals confirming controller effectiveness.

This methodical approach provided a systematic framework to synthesize a discrete controller capable of dynamically adjusting the beam angle based on steering inputs, thus enhancing vehicle safety and driver visibility during cornering.

Obtained controller function after the computation is:

$$C(z) = 57.958 \frac{z^2 - 1.95z + 0.9512}{(z-1)(z-0.7665)}$$

3.4 Limitations of the Monolithic Implementation

While the monolithic implementation of the Adaptive Light Beam controller within a single Stateflow chart offers straightforward integration and centralized logic management, it presents several inherent limitations:

1. **Scalability Issues**
   As the complexity of the system grows, the monolithic design becomes increasingly difficult to maintain and extend. Adding new features or modifying existing behaviour requires navigating and updating a large, intertwined state machine, which can be error-prone and time-consuming.
2. **Reduced Modularity and Reusability**
   The tightly coupled design limits the ability to reuse individual components or states in other projects or contexts. Reusability is crucial in model-based design for efficient development cycles, but the monolithic approach often forces duplication or extensive refactoring.
3. **Testing and Debugging Challenges**
   Debugging complex state transitions and signal interactions in a single, large Stateflow chart can be cumbersome. Isolating faults or verifying specific functionality

requires significant effort due to the interdependencies between states and shared variables.

4. **Limited Team Collaboration**
   In a monolithic model, parallel development is hindered because multiple developers working on the same Stateflow chart can cause merge conflicts or overwriting of changes. Modular implementations facilitate better task distribution and integration.

5. **Performance Constraints**
   Large state machines may introduce computational overhead and increase simulation times, which is critical for real-time embedded systems. Optimizing and profiling performance in a monolithic setup can be more difficult compared to modular designs.

6. **Difficulty in Formal Verification and Validation**
   Formal methods for verifying system correctness are more challenging to apply on large monolithic models due to state space explosion and the complexity of interactions. Modular approaches can simplify verification by reducing state space and isolating functionality.

Overall, while the monolithic implementation serves well for initial prototyping and small-scale systems, transitioning to a modular or hierarchical design is recommended for enhanced maintainability, scalability, and robustness in larger and more complex adaptive lighting control systems.

3.5 Challenges in Migrating Legacy Application to SOA

Migrating legacy applications, such as the monolithic Adaptive Light Beam controller, to a Service-Oriented Architecture (SOA) framework presents several key challenges:

1. **Monolithic Design and Complexity of Decomposition**

   Legacy systems are often tightly coupled with no clear modular boundaries, making it difficult to identify and extract discrete, reusable services. Decomposing intertwined logic requires deep domain knowledge and thorough understanding of dependencies.

2. **Sequential Order of Execution**

   Legacy applications usually follow a predefined sequential execution order. This rigid sequence complicates converting the system into loosely coupled services that support dynamic discovery and reconfiguration at runtime.

3. **Data and Interface Standardization**

   Heterogeneous data formats and proprietary communication protocols are common in legacy systems. Ensuring consistent data representation and defining standardized service interfaces requires significant re-engineering for interoperability within SOA.

4. **Performance Overhead**

   SOA introduces communication overhead through network calls and message processing. Real-time control systems, such as adaptive lighting, have strict timing requirements that may be compromised if service granularity and communication mechanisms are not carefully optimized.

5. **Ensuring Functional Equivalence**

   Maintaining exact legacy system behaviour during and after migration is challenging. Reconstructed services must faithfully replicate original control logic, especially in safety-critical automotive applications where deviations can cause hazards.

6. **Integration with Existing Infrastructure**

   Legacy systems often rely on proprietary hardware and tightly integrated components. Adapting these into a loosely coupled SOA may require additional middleware or adapters, increasing system complexity.

7. **Testing and Validation Complexity**

   Breaking the system into multiple interacting services complicates system-level testing. Comprehensive validation must cover both individual services and their interactions under diverse conditions to ensure reliability.

8. **Change Management and Team Skillsets**

   Migrating to SOA requires organizational change, including retraining teams on new paradigms, tools, and communication protocols. Resistance and learning curves can slow the migration process.

9. **Security Concerns**

   SOA exposes services over networks, increasing the attack surface. Legacy systems need robust authentication, authorization, and encryption to ensure security in a distributed environment.

3.6 Summary

While SOA migration offers benefits such as modularity, scalability, and easier maintenance, addressing these challenges demands careful planning, iterative development, and thorough testing. Utilizing Model-Based Design methodologies and domain expertise can help facilitate a smooth and effective migration, preserving system performance, safety, and reliability.

Chapter 4

4. Design of Adaptive Light Beam Controller – Service-Oriented Architecture

4.1 Decomposition of Traditional Software components into services

Decomposing traditional application software compositions into services is a critical step in transitioning to a Service-Oriented Architecture (SOA). This process involves breaking down a monolithic architecture into smaller, modular components, enabling greater flexibility, scalability, and adaptability, particularly in the context of Software-Defined Vehicles (SDVs).[1][4][10]

The decomposition process can be broadly divided into four sequential steps, each represented as distinct phases in the transformation from monolithic applications to service-oriented systems:

4.1.1 Identify and Analyse Services

The first and most challenging step is to identify the potential services, including key components, functionalities, execution order, and dependencies within the existing monolithic system. Engineers must carefully analyse these elements to determine logical service boundaries and to decompose the monolithic application into smaller, manageable components. [1][4]

4.1.2 Define Services and Interfaces

Once services are identified, the next step is to clearly define the interfaces between them. This involves specifying communication protocols and data formats to ensure seamless interaction. Well-defined interfaces are essential for enabling interoperability and loose coupling between services. [1][4]

4.1.3 Define Service Contracts

Service contracts formalize the interaction rules between services. They specify terms and conditions, including service versioning, error handling, and performance expectations. In automotive systems, these concepts are embodied in standards like the AUTOSAR 22-11 schema, which supports versioning to allow new service versions without disrupting existing clients. [1][4]

4.1.4 Implement and Deploy Services

The final step involves implementing each service as an independent application. This includes creating the necessary artifacts such as interface descriptions, communication bindings, and deployment packages. Each service can then be

deployed, managed, and updated independently, supporting scalability and maintainability. [1][4]

This structured approach ensures a systematic and controlled migration from traditional monolithic software to a modern SOA, facilitating the development of modular, reusable, and maintainable components suitable for complex automotive applications.

4.2 Using Model-Based Design to Decompose Adaptive Light Beam Controller Monolithic Application into Services

Model-Based Design (MBD) has long been used to develop applications for both non-AUTOSAR and AUTOSAR Classic frameworks. More recently, it has also been extended to support Service-Oriented Architecture (SOA) based applications, including those developed on the AUTOSAR Adaptive platform and generic SOA frameworks. In the context of Software-Defined Vehicles (SDVs), the industry commonly leverages either a generic SOA or AUTOSAR Adaptive-based SOA. Model-Based Design offers a unified development platform that efficiently manages the entire development lifecycle across these diverse platforms—ensuring consistency, reusability, and increased development efficiency. [1][4][9]

The process of using Model-Based Design to decompose monolithic application components into modular services involves several key steps:

4.2.1. Identify and Analyse Services

The initial step is to thoroughly understand the components of the legacy monolithic application, including their functionalities, execution order, and interdependencies. Typically, monolithic applications are deployed as a single executable artifact containing all components bundled together (Figure 4: Monolithic ALBC Simulink design).

For example, consider the **Adaptive Light Beam Controller**—a key automotive system responsible for managing multiple lighting modes such as *Low Beam*, *High Beam*, *Cornering Beam*, and *City Mode*. Initially, this functionality may exist as a **monolithic Stateflow model**.

To migrate such legacy monolithic designs toward a **Service-Oriented Architecture (SOA)**, **Model-Based Design (MBD)** principles are applied—specifically:

- **Single Responsibility Principle:** Each service should perform one well-defined function. [1][4]
- **Dependency Inversion Principle:** High-level services should depend on **abstract contracts** (e.g., APIs or events), rather than on the **concrete implementations** of lower-level services. [1][4]

By applying these principles, the monolithic system can be systematically decomposed into distinct, reusable services—such as the **VehicleOncoming Service**, **AmbientLight Service**, **VehicleSpeed Service**, **SteeringAngle Service**, **RoadCurvature Service**, and **BeamAngleController Service**.

This decomposition isolates functionality into loosely coupled services, allowing independent development and easier maintenance. For instance, the cornering beam logic that adjusts beam angle based on steering input can be encapsulated as a separate service, distinct from the main Stateflow component.

4.2.2. Define Services and Interfaces

Once services are identified, the next step is to define clear interfaces for each service. These interfaces form the boundaries through which services communicate with each other, encapsulating functionality and enabling benefits such as reuse, maintainability, version control, and orchestration.
Using tools like System Composer, engineers can configure service ports and interfaces, ensure data consistency and visually represent dependencies and interactions between services. (see Figure 12) [1][4][9]

4.2.3. Define Service Contracts

Establishing explicit service contracts is critical to delineate each service's inputs, outputs, and expected behaviour. Well-defined contracts allow services to be developed, tested, and deployed independently without tight coupling to other system components. Service contracts also facilitate backward-compatible versioning, enabling new service versions to be released without disrupting existing clients. *(see figure 13, 14)* [1][4][9]

4.2.4. Implement and Deploy Services

Finally, services are implemented using Model-Based Design's client-server interfaces. Applications like the Adaptive Light Beam controller's discrete control modules are realized as separate service components within Simulink. These modular services can then be independently deployed within an SOA environment.
Additionally, Embedded Coder can be used to generate C++ code from these models for deployment in generic SOA applications, facilitating integration with existing middleware and runtime environments. [1][4][9]

This Model-Based Design workflow provides a systematic approach to transforming monolithic automotive software, including complex controllers like the Adaptive Light Beam system, into modular, service-oriented applications. This supports scalability, maintainability, and efficient integration in modern SDVs.

4.3 SOA Model Implementation in Simulink

4.3.1 How to define service and interfaces in Simulink

In System Composer:

1)Create a software architecture model

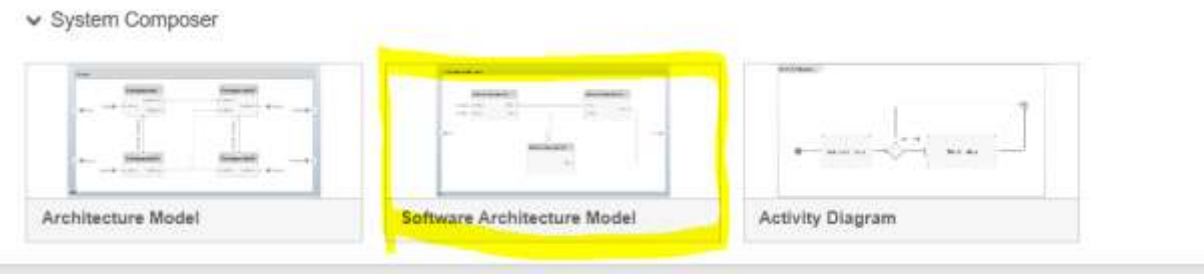


Figure 5: System composer software architecture model

2) Create a software component box for our main application i.e. AdaptiveLightController

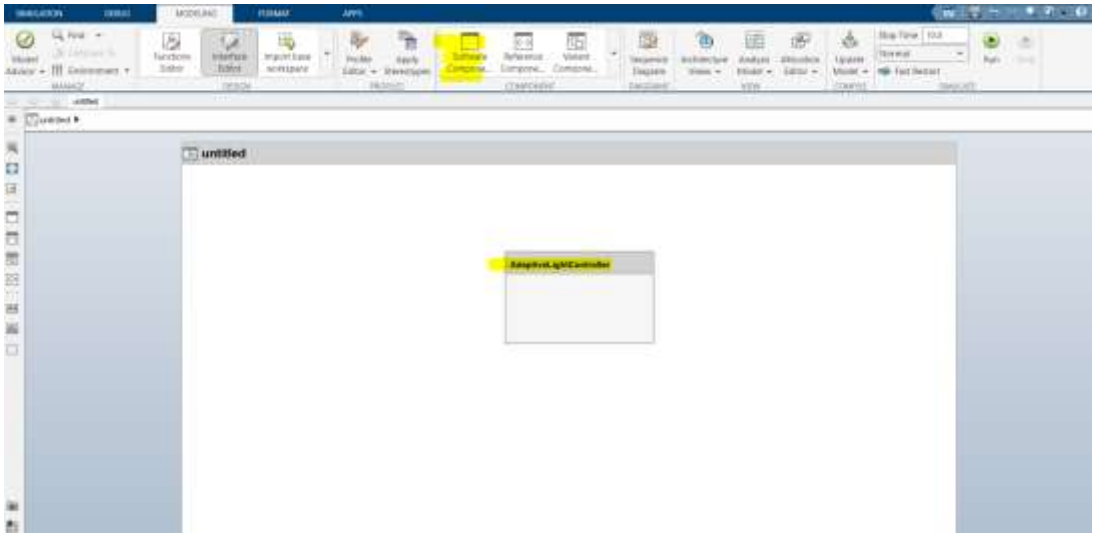


Figure 6: Simulink software component

3) Define all I/O in the main application and connect to Interface boundary of composition(as in below figure 7). The I/O of the application are: VehicleSpeed_km_h, RoadCurvature_per_m, SteeringAngle_degree, AmbientLight, VehicleOncoming, BeamAngle and BeamRange.

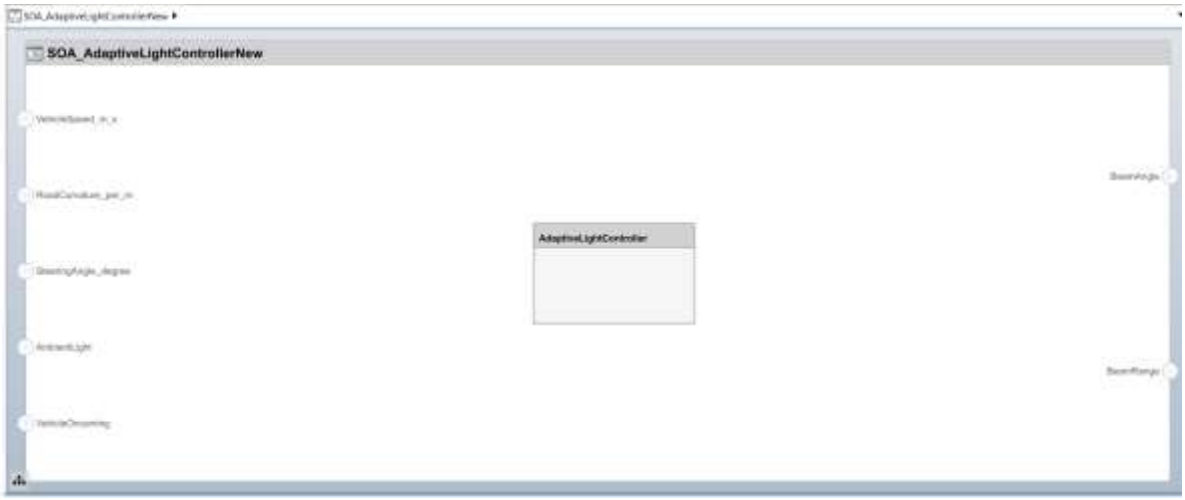


Figure 7: Simulink Adaptive Light Controller(*)

4) Create a software component box for all new service components. Where the services are: VehicleOncomingService, VehicleSpeedService, AmbientLightService, SteeringAngleService, BeamAngleController and RoadCurvatureService. AdaptiveLightController is the main application.

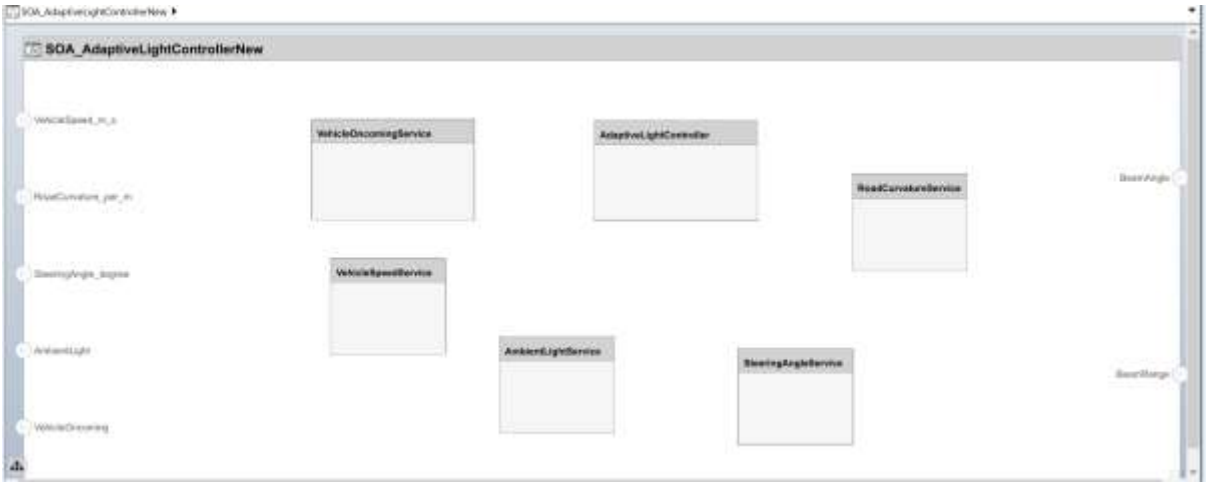


Figure 8: Simulink Services software components(*)

5) Connect all service components to main application (AdaptiveLightController) with client server connectors

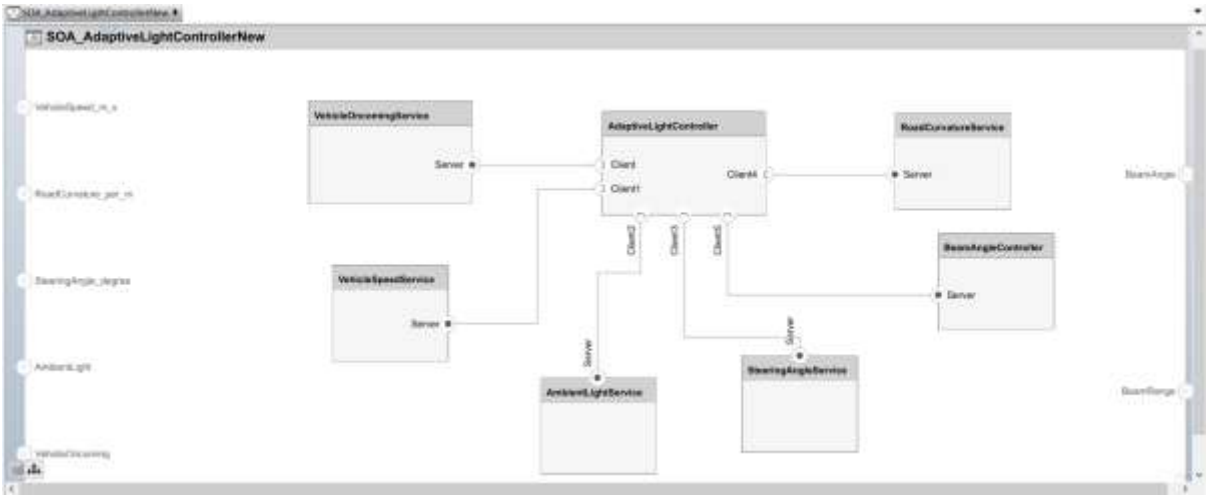


Figure 9: Services connected to main application with client server connectors(*)

6) Defining the Client and Server Interfaces using the Interface Editor

Open the Interface Editor, and the Interfaces window will appear below. Select “Add data interface”, then choose Service Interface from the options. This will create a new service interface. You can now name it according to your application requirements (e.g., AmbientLight_servif).

Figure 10: Client server interface definition

Once the service interface is created, add elements that define the service behavior — such as the input it receives, the function it executes, and the output it produces. To do this, select "Add element to the selected interface" from the Interfaces window (located next to Add data interface). This will generate a default function prototype in the form y = f(u), which you can modify as needed.

For example, in AmbientLight_servif, I created a function called computeAmbientLight. It takes AmbientLight as input and returns outAmbientLight as output. The final function looks like:

outAmbientLight = computeAmbientLight(AmbientLight)

In the same way, additional service interfaces can be created for other services, as illustrated in the figure.

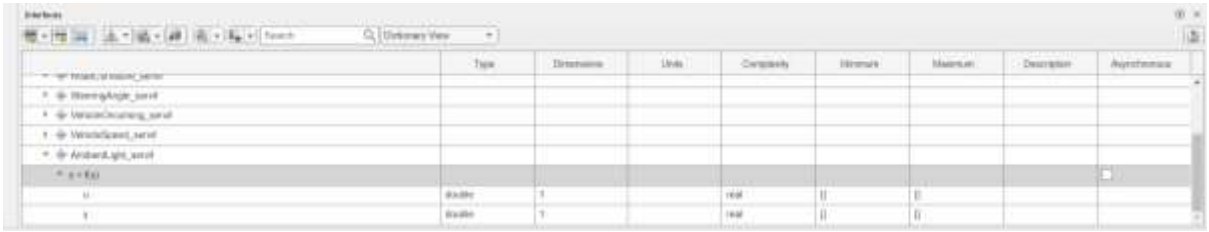


Figure 11: Client service interface function definition

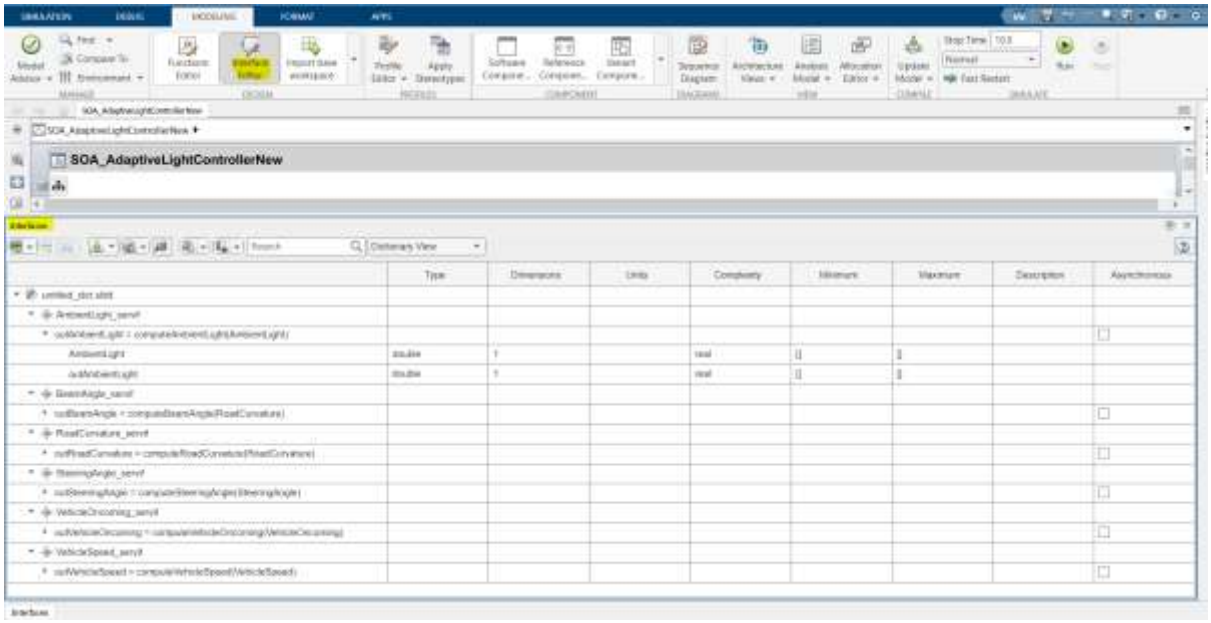


Figure 12: Client service all functions definition

7) After having created the service interface, then link them to the client-server connector. For example:

1. Select the service interface (e.g., AmbientLight_servif).
2. Right-click on the client and server ports.
3. Choose **"Apply selected interface: AmbientLight_servif"**.

To verify that the service interface is correctly assigned to both ports, select the interface. If the client and server ports are highlighted in **purple**, it confirms that the interface has been successfully linked.

8) To create the Simulink behaviour model for a service component:

1. **Right-click** on the service software component.
2. Select **"Create Simulink Behaviour"**, then click **OK**.

Repeat this process for **each service component** to generate their respective Simulink behaviour models.

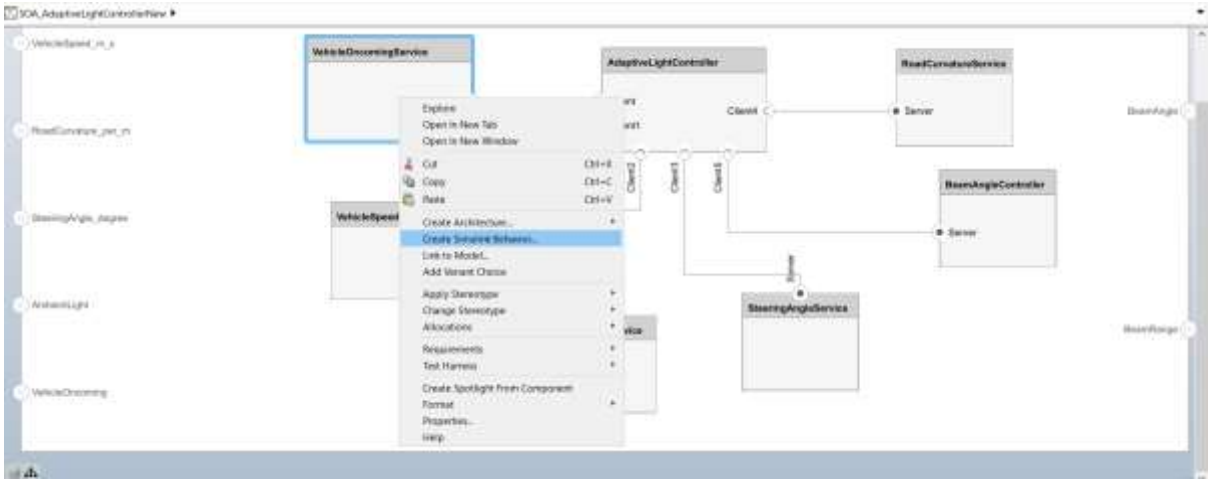


Figure 13: Simulink behaviour for a service(*)

8) Create the Simulink behavior model for the **Main Application** (i.e., AdaptiveLightController) by following these steps:

1. **Right-click** on the AdaptiveLightController software component.
2. Select **"Create Simulink Behavior"** and click **OK**.

Ensure that all **input/output (I/O)** connections are properly attached to the **Main Application component**, while the service components should already be connected via their respective **client-server interfaces**.

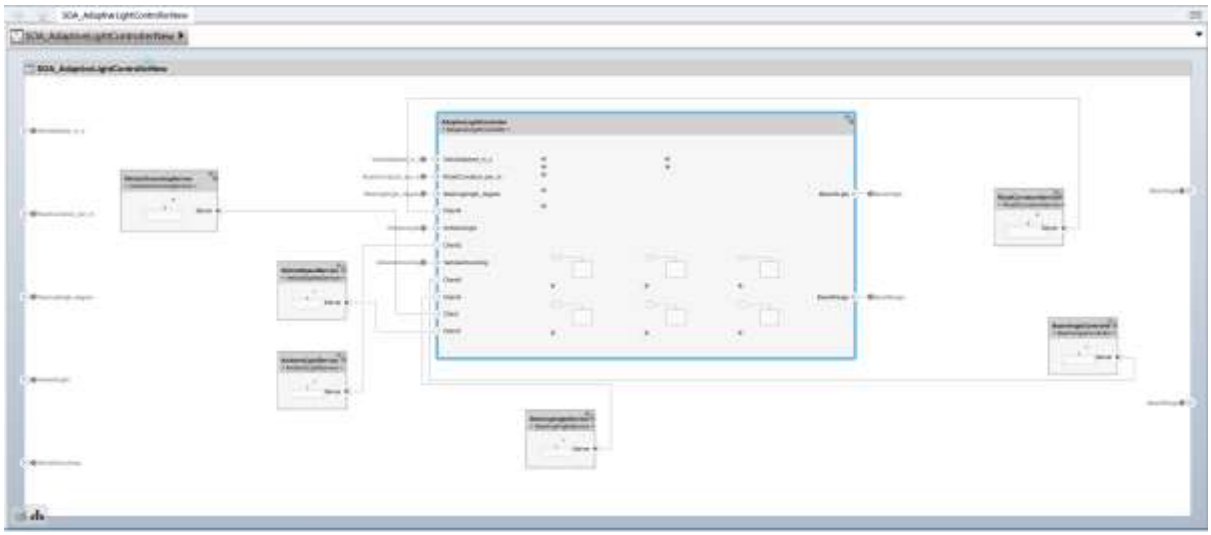
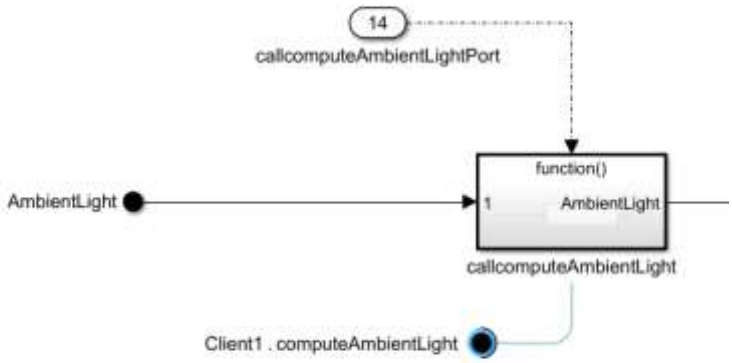


Figure 14: Simulink behaviour for a main application(*)

9) Design of the Main Application (AdaptiveLightController)

In the previous step, Simulink automatically generated the input and output bus signals and created a Function-Call Subsystem for each connected service interface. If the service function is defined in a format such as **outAmbientLight = computeAmbientLight(AmbientLight)**, the

corresponding input (**AmbientLight**) and output (**outAmbientLight**) signals are added to the Function-Call Subsystem



.

Figure 15: Add input and output signal to Function call subsystem

Figure 16 illustrates the internal structure of the computeAmbientLight client function call. It includes a Function Caller block that receives the input signal and invokes the server counterpart. The server performs the necessary computation and returns the output signal, which is then utilized by the main application. This setup exemplifies how the Service-Oriented Architecture (SOA) promotes modularity by decoupling the server implementation from the main application. This separation enables independent updates to the server logic over time and supports use cases such as Over-the-Air (OTA) updates.

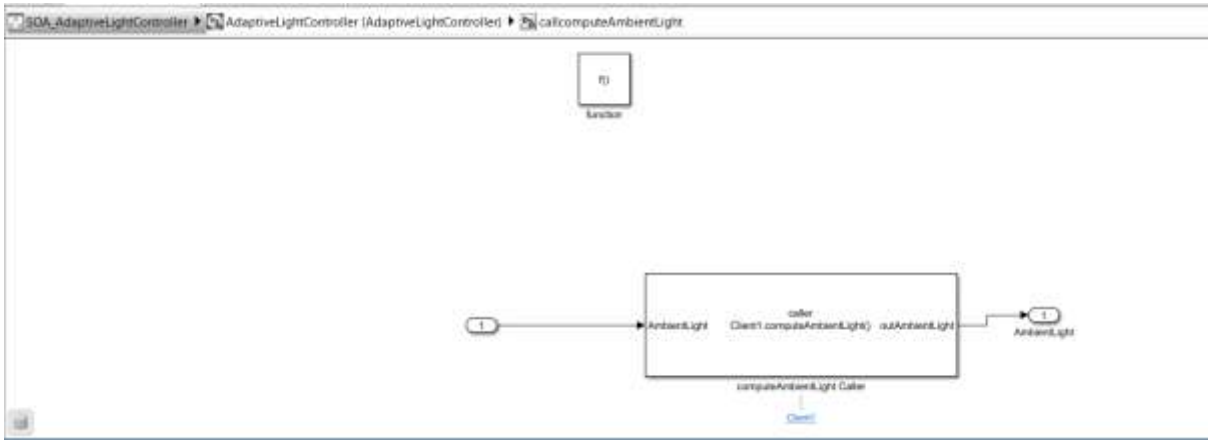


Figure 16: Inside the Function Call Subsystem

Figure 17 shows the server counterpart of a service call, which is triggered when the Function-Call Subsystem in the main application invokes the corresponding function.

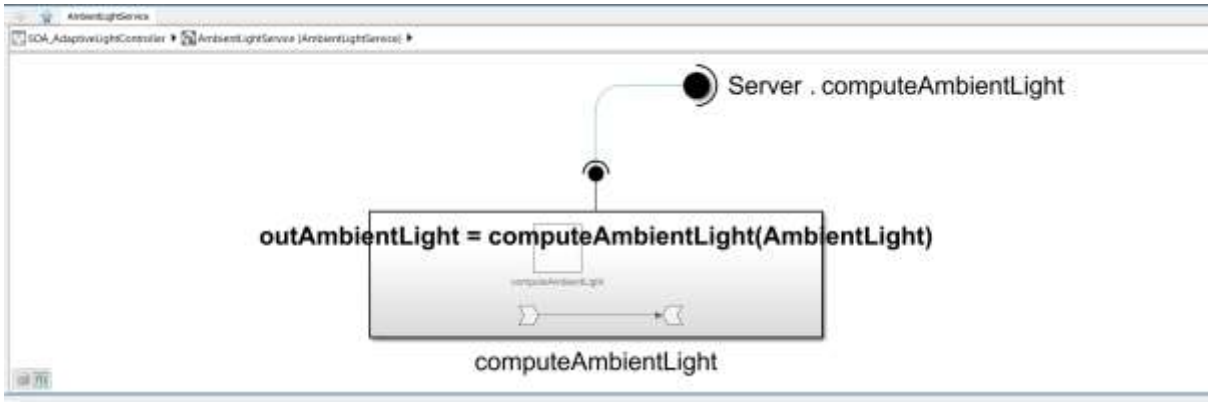


Figure 17: Server counterpart of the Function Call subsystem

Similar client-server service calls are defined for other functionalities, reinforcing the modular and scalable nature of the SOA-based system design.

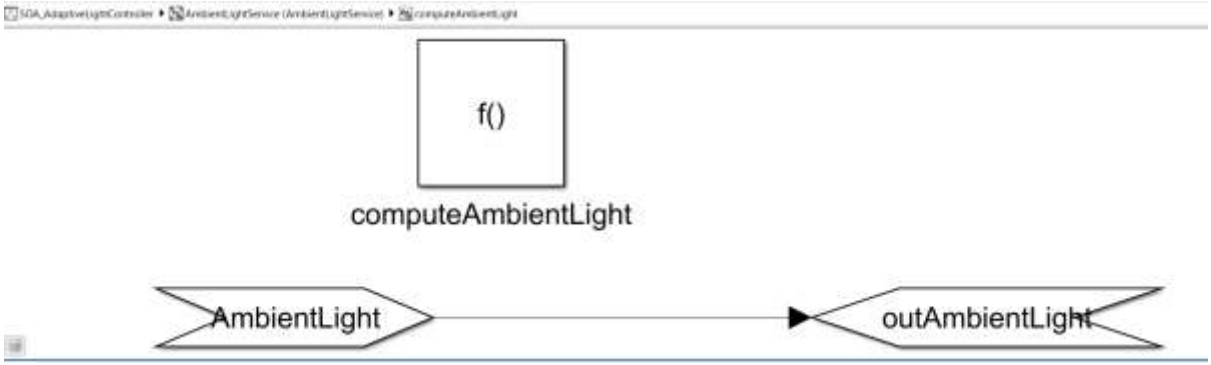


Figure 18: Inside of the Server counterpart of the Function Call Subsystem

The **AdaptiveLightController** is structured in a modular, service-oriented way:

- Each **Function-Call Subsystem** corresponds to a specific service (e.g., AmbientLight, VehicleSpeed, etc.).
- These subsystems take their respective input bus signals—for example, the AmbientLight Function-Call Subsystem receives the Ambient Light input, and so on for the others.

Stateflow Integration in SOA-Based Design

In the previous Monolithic Design of the AdaptiveLightController, a single Stateflow chart was used to implement the internal logic controlling Beam Range and Beam Angle.

In the SOA-based redesign, we continue to use a Stateflow-based approach, but with an important architectural adjustment:

The **Stateflow** chart is now placed inside a **Function-Call Subsystem**.

This design decision is essential in the context of **Service-Oriented Architecture (SOA)** and **AUTOSAR** compatibility:

- Enables event-driven execution of the control logic.
- Aligns with AUTOSAR runnable semantics, where function-call triggers map directly to runnable.
- Promotes modularity, easier testing, and standards-compliant behaviour modelling.

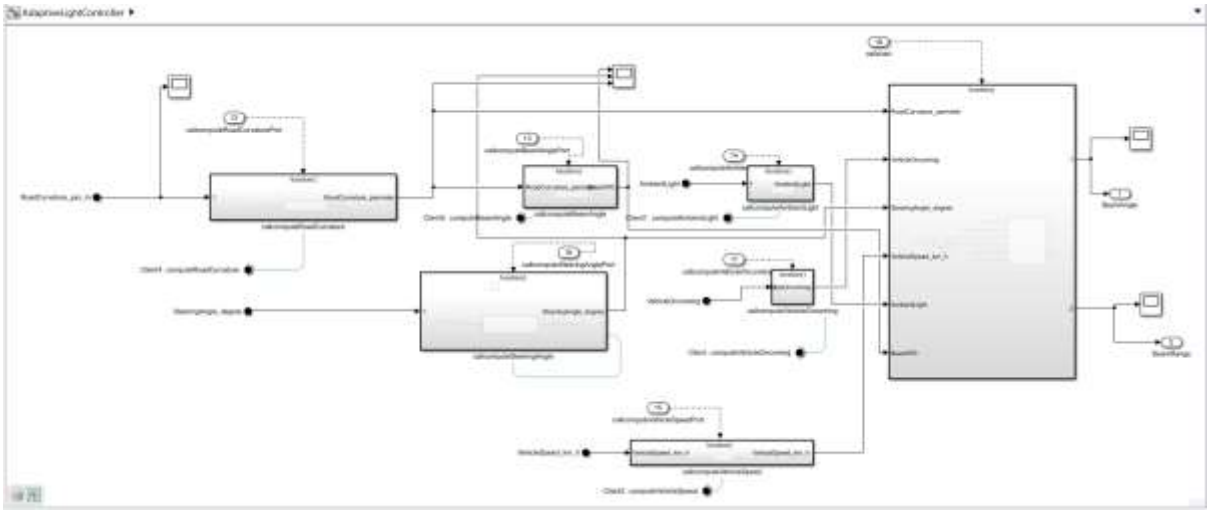


Figure 19: Main Application with State chart inside Function call Subsystem

Final Architecture Overview

The resulting **AdaptiveLightController** is a modular, SOA-compliant system where:

- Each service input is handled by a dedicated Function-Call Subsystem.
- The core decision logic, implemented via Stateflow, is triggered through a function-call, ensuring the system only reacts when relevant inputs or events occur.
- This setup defines the internal logic that determines Beam Range and Beam Angle, based on service-provided environmental and vehicle data.

The figure 19 illustrates the final AdaptiveLightController architecture, showcasing the clean separation of services and the centralized decision logic encapsulated in the Stateflow-driven subsystem.

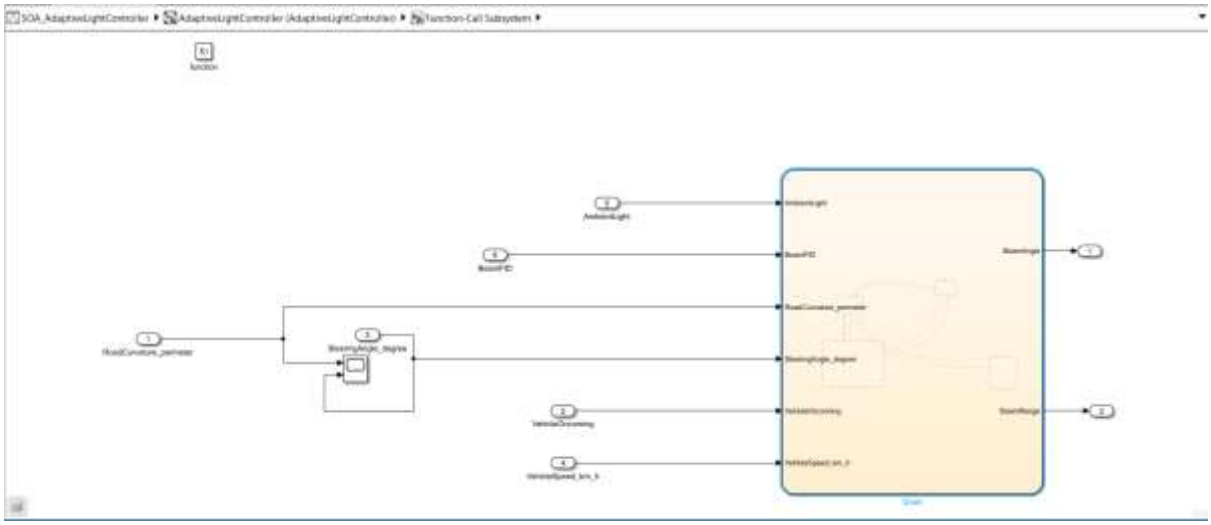


Figure 20: State Flow with input and output signals

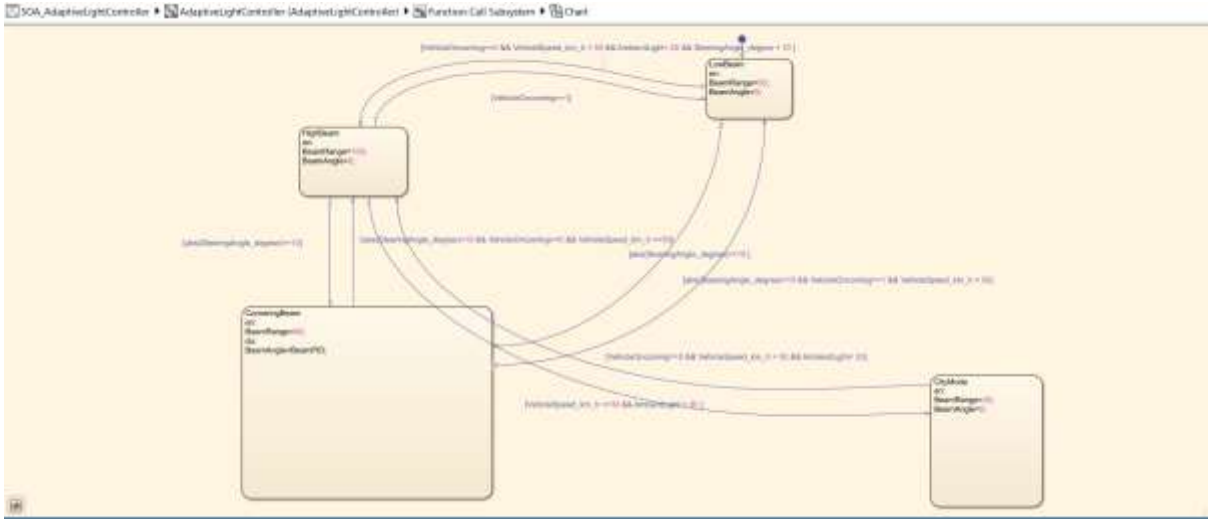


Figure 21: Logic inside Stateflow which determine state of system

4.4 Advantages Observed from SOA-Based Design

The adoption of a **Service-Oriented Architecture (SOA)** in the design of the **Adaptive Light Beam Controller** has introduced several key advantages compared to the traditional monolithic approach:

1. Modularity and Reusability

- Each function (e.g., ambient light sensing, vehicle speed handling) is encapsulated as an **independent service component**.
- These services can be **reused** across different applications or systems without redesign.
- Changes to one service do not impact others, as long as interfaces remain unchanged.

2. Improved Maintainability

- The clear separation of logic into client-server components allows for **easier debugging, updates, and testing**.
- Individual services can be tested in isolation, reducing system complexity during validation.

3. Scalability

- New services or features can be added without major rework.
- For example, adding a new weather or camera-based input only requires integrating another service interface.

4. AUTOSAR Compliance

- SOA aligns with AUTOSAR principles, enabling:
    - Standardized **runnables** and **interface definitions**
    - Compatibility with code generators and embedded platforms
    - Easier deployment in **AUTOSAR Classic or Adaptive platforms**

5. Event-Driven Execution

- Function-Call Subsystems trigger logic (e.g., Stateflow) **only when needed**, saving computational resources.
- This mirrors real-world embedded execution, where runnables execute based on **events or signals**, not continuously.

6. Enhanced Integration and Interoperability

- Clearly defined **interfaces (ports and services)** allow seamless integration with other components or external systems.
- Encourages **team-based parallel development**, since service contracts are defined upfront.

Chapter 5

5. Code Generation and Integration of the code into Android studio

5.1 C++ Code Generation using Embedded Coder from Simulink Models

The process of transforming a Simulink model into deployable **C++ source code** is a critical step in embedding model-based designs into real-time and embedded systems. **MATLAB Embedded Coder** provides specialized tools for generating high-quality, portable C and C++ code directly from Simulink models, which can then be integrated into target environments such as Android Studio for further development and deployment.[16]

5.1.1 Overview of Embedded Coder

Embedded Coder is an add-on to MATLAB/Simulink that:

- Generates **highly optimized C and C++ code** from models and Stateflow charts.
- Supports **hardware-specific optimizations** and integration hooks.
- Offers configuration settings for **data types, naming conventions, and file packaging**.
- Produces **traceable code** with links back to Simulink blocks for debugging.

This capability is widely used in automotive, aerospace, and industrial control applications, where real-time performance and deterministic behaviour are critical. [16]

5.1.2 Preparation of the Simulink Model

Before initiating code generation, the Simulink model must be:

1. **Functionally validated** – Ensure the simulation matches design requirements.
2. **Configured for code generation** – Use *Configuration Parameters → Code Generation* to set:
    - **System target file**: ert.tlc (Embedded Real-Time) for standalone code. [16]
    - **Language**: C++ (instead of default C). [16]
    - **Toolchain**: Select an available C++ compiler compatible with your host system. [16]
3. **I/O interfaces defined** – External inputs/outputs must be represented using Inports and Outports with clearly defined data types. [16]
4. **Sample times fixed** – Embedded code requires deterministic execution rates.

5.1.3 Set Configuration parameters for Code Generation

1. **Open Model Settings**

In the *Modelling* tab, select **Model Settings** (or press Ctrl+E) to open the *Configuration Parameters* dialog. [16]

2. **Configure Solver Options**
    - Navigate to the **Solver** pane.
    - Set *Solver Type* to **Fixed-step**.
    - Select **discrete (no continuous states)** as the solver.
    - Specify the *Fixed-step size* as **Ts(0.01s)**, where Ts is the model's sampling time.
    - This ensures a deterministic execution rate suitable for embedded code deployment. [16]
3. **Select System Target File**
    - Go to the **Code Generation** pane.
    - Set *System target file* to **ert.tlc** (Embedded Real-Time) for high-quality standalone code generation. [16]
4. **Set the Target Language**
    - In the *Language* field, choose **C++** to produce object-oriented, portable code. [16]
5. **Adjust Build Process Settings**
    - Under *Build Process*, enable **Generate code only**. [16]
    - This option produces the source files without compiling them, allowing for external integration in Android Studio.
6. **Choose Toolchain**
    - In the *Toolchain* field, select **MinGW64 | gmake (64-bit Windows)**. [16]
    - This ensures compatibility with Windows-hosted builds targeting cross-platform deployment.
7. **Define Code Interface Packaging**
    - In the *Interface* section of the *Code Generation* pane:
        - Set *Code interface packaging* to **C++ Class**. [16]
        - Configure additional interface parameters (e.g., class naming conventions, file packaging) according to project requirements.
8. **Set Hardware Implementation**

- Go to the **Hardware Implementation** pane.
- Set *Hardware board* to **None**.
- Configure *Device vendor* and *Device type* according to the intended deployment hardware.

With these settings, the generated C++ code will be optimized for embedded integration, modular in structure, and portable to Android Studio or other development environments.

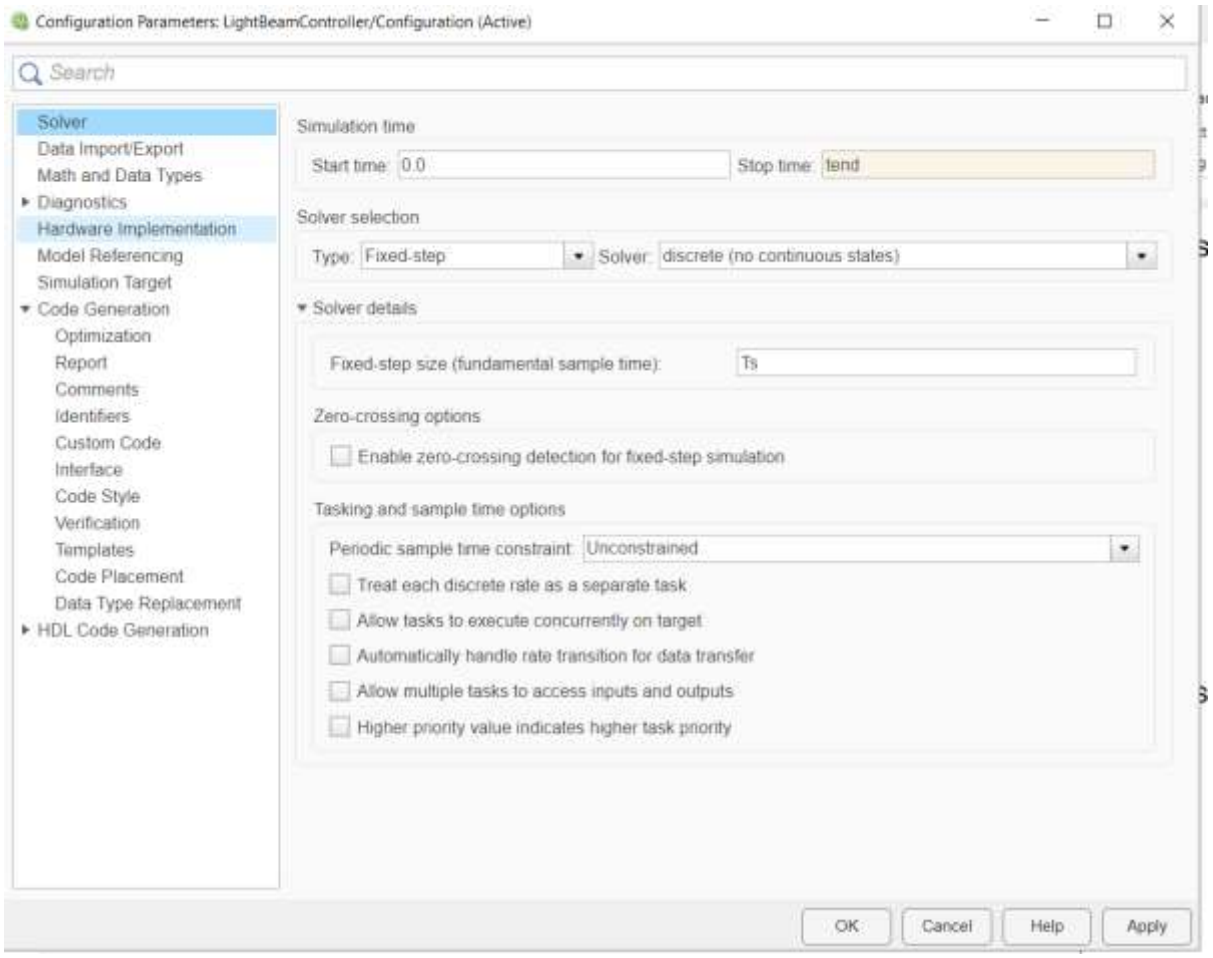


Figure 22: Solver configuration

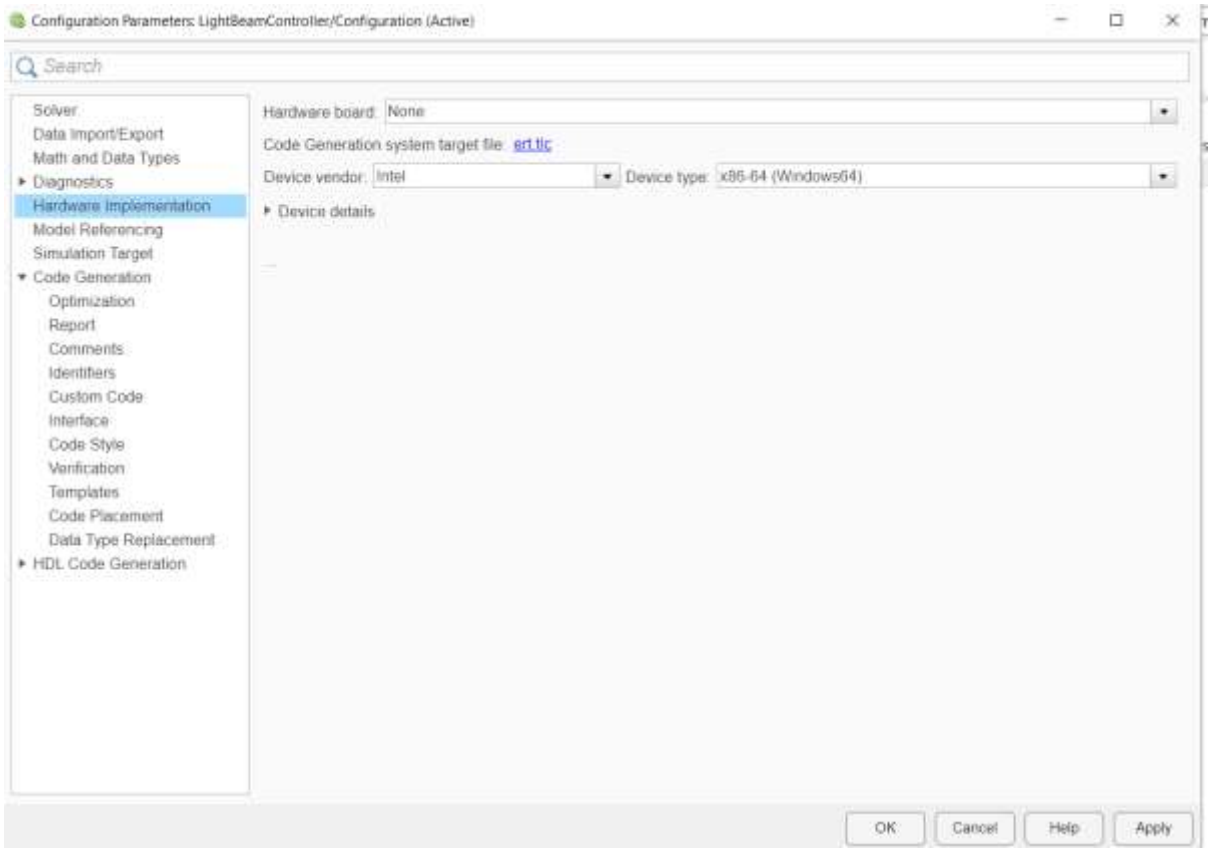


Figure 23: Hardware implementation configuration

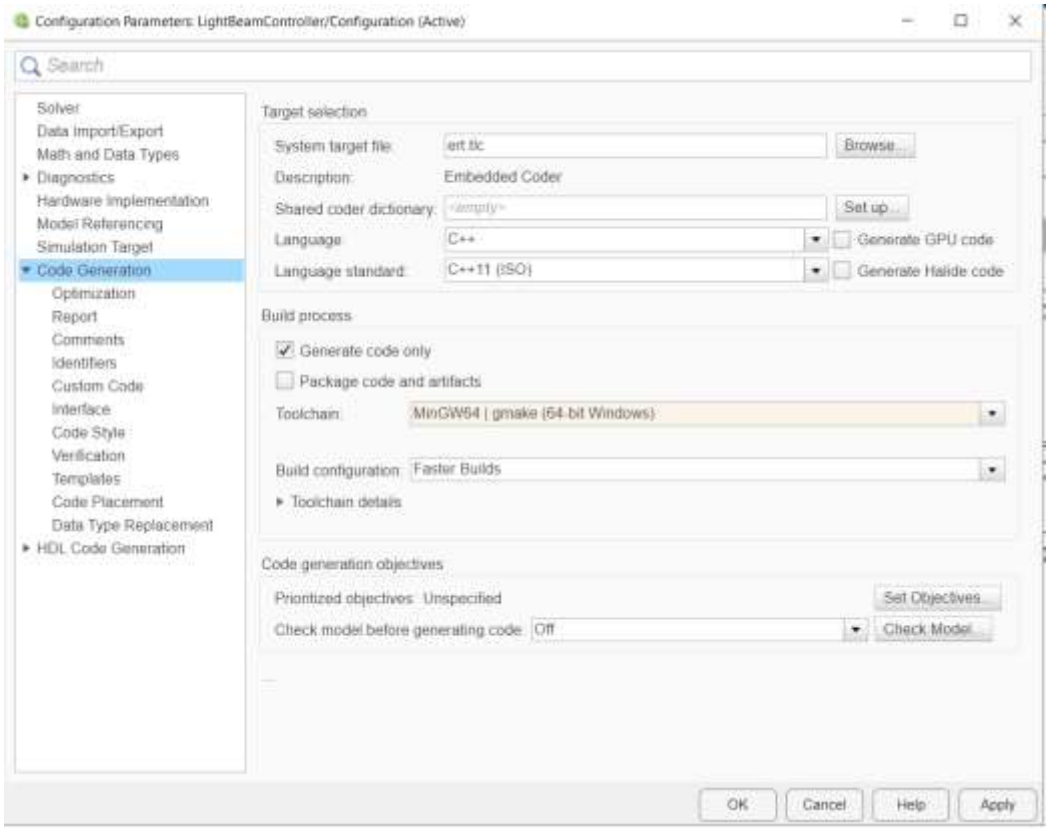


Figure 24: Code generation configuration

5.2 Structure of File Folder after code generation in both Monolithic and SOA Simulink models

5.2.1 File structure and generated files for Monolithic Simulink model

1. After the code has been generated using embedded coder. It has generated two folders named slprj and MonolithicAdaptiveLightBeamController_ert_rtw.

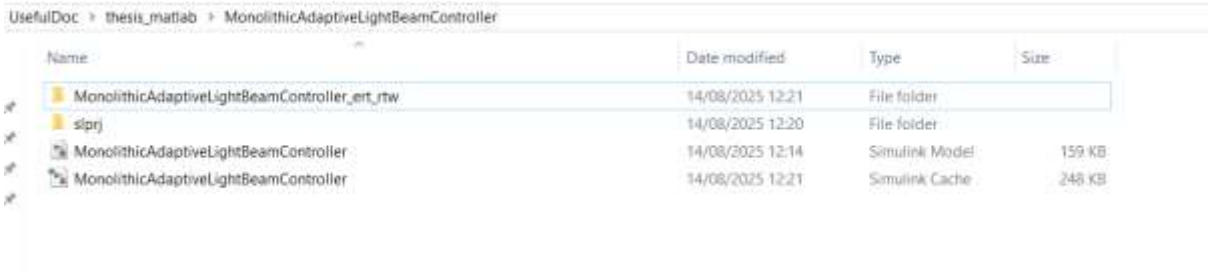


Figure 25: Monolithic Application folder structure

2. Inside the folder MonolithicAdaptiveLightBeamController_ert_rtw it has generated all the source code and header file which will be used while integrating the code into Android Studio.

Figure 26: Files inside folder MonolithicAdaptiveLightBeamController_ert_rtw

3. While file under slprj folder are not relevant for code integration into Android Studio

5.2.2. File structure and generated files for SOA Simulink model

1) File structure before Simulink run.

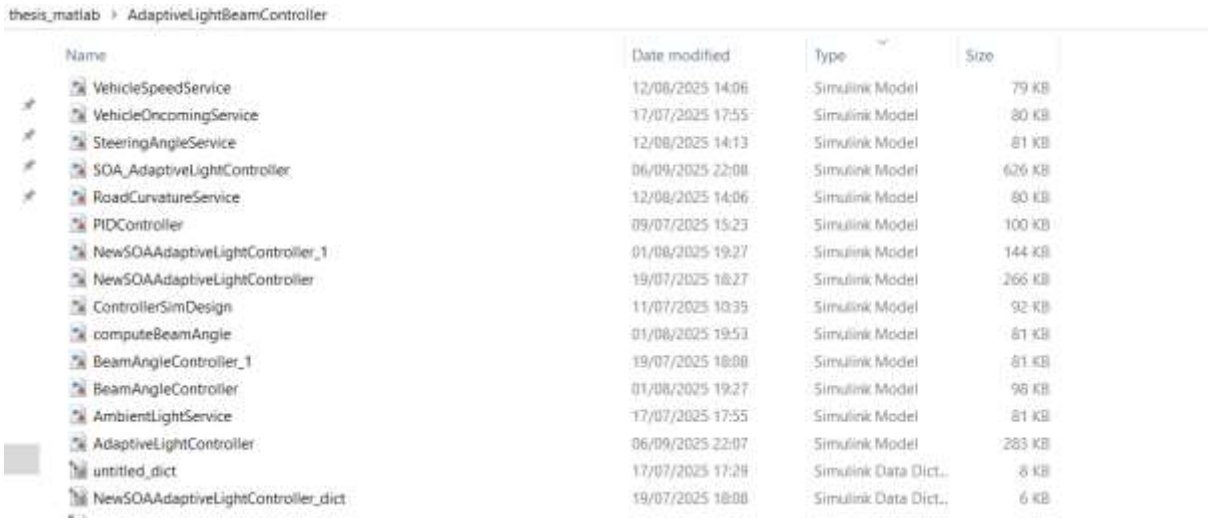


Figure 27: Simulink files before code generation

2) File structure after building the Simulink model and generating the code. The Simulink has created two folders: splrj and SOA_AdaptiveLightController_ert_rtw.

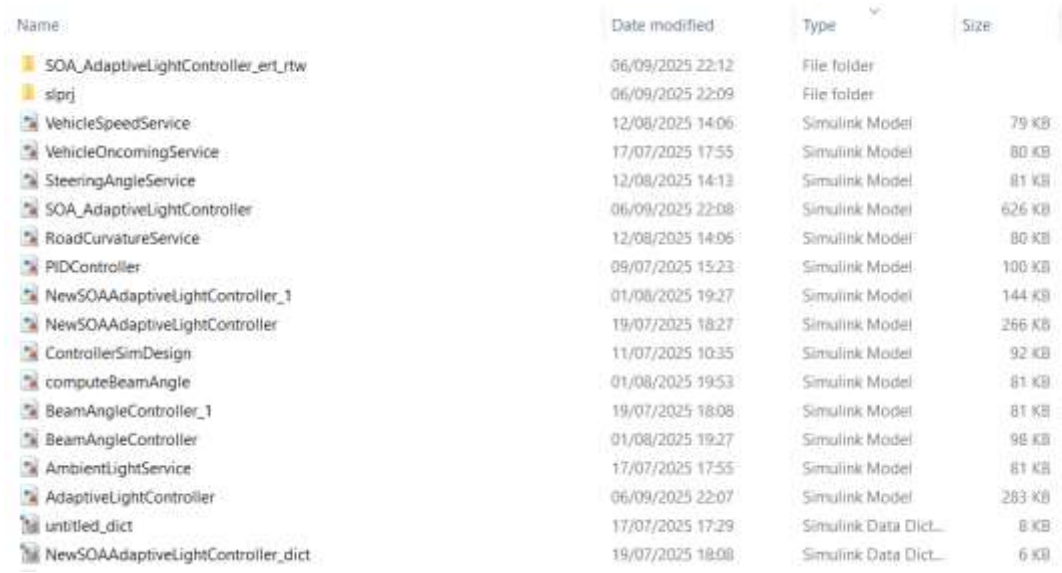


Figure 28: File structure after code generation

3) File inside folder slprj.

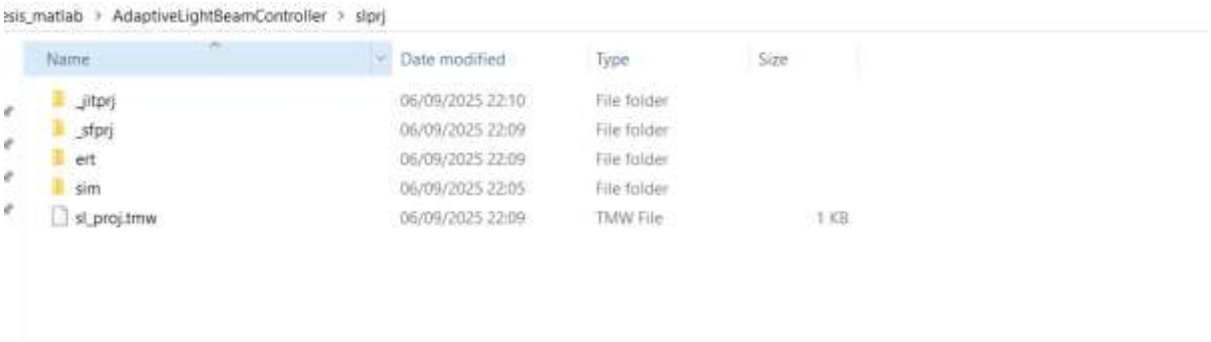


Figure 29: File inside folder slprj

All the relevant code files are under ert folder. The file structure of ert folder.

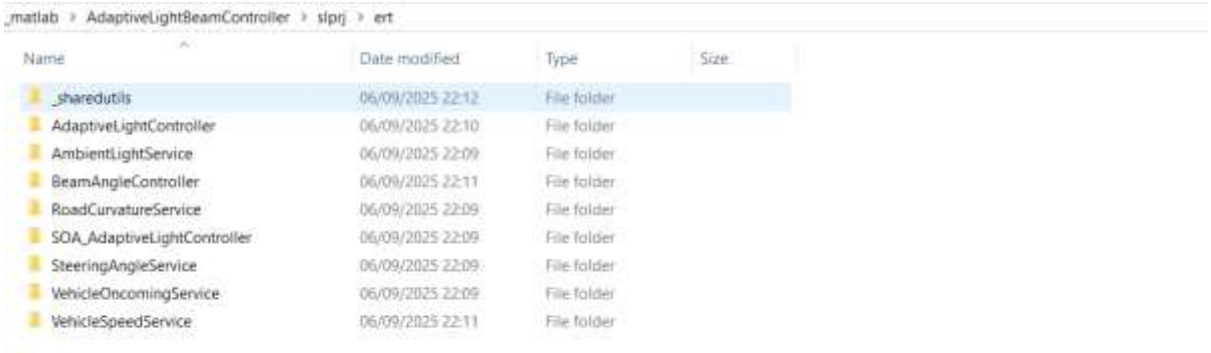


Figure 30: Files inside ert folder

Under the _sharedutils folder, you find code generated corresponding to the service interface.

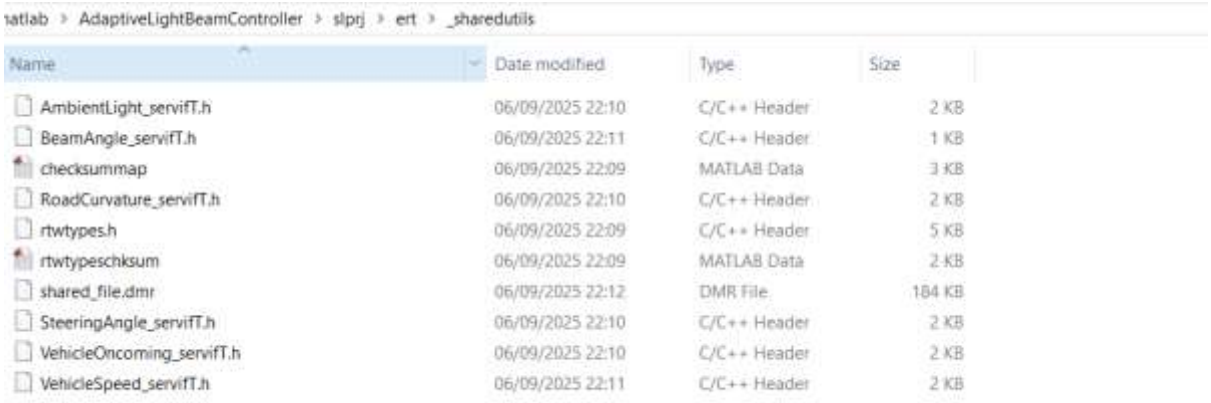


Figure 31: Files inside _sharedutils folder

While in the folder corresponding to the service for instance AmbientLightService inside the folder you find it's header and c++ code files and all relevant source code files. Similarly, rest of the services source code can be found in their respective folder.

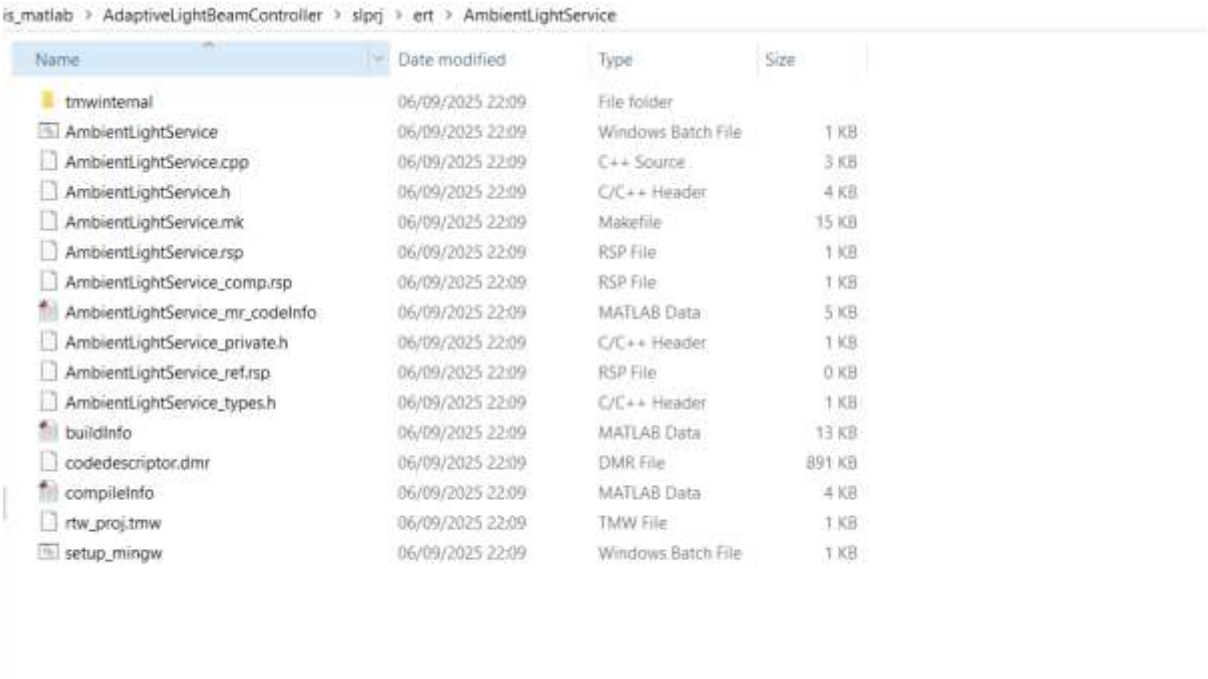


Figure 32: Files inside service folder namely AmbientLightService

4) File inside folder SOA_AdaptiveLightController_ert_rtw.

Inside folder you find the source code files associated to the main architecture model.

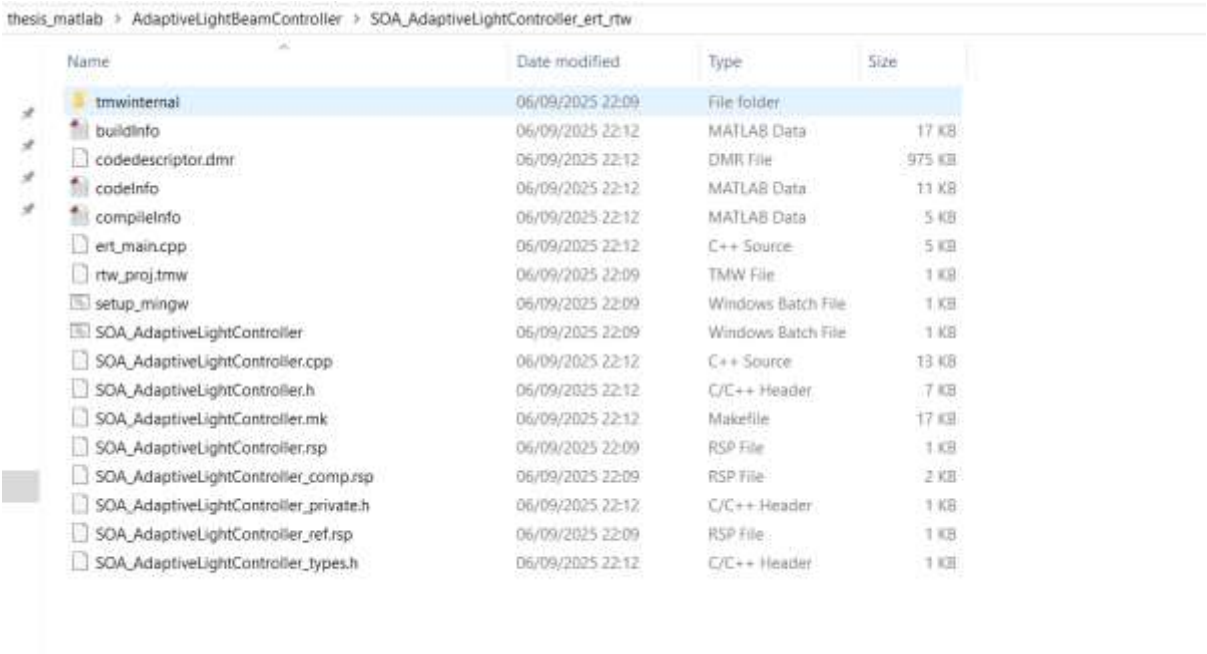


Figure 33: Files inside folder SOA_AdaptiveLightController_ert_rtw

5.3 Code Integration in Android Studio of Monolithic Models

This section provides a complete, step-by-step guide on integrating a monolithic Simulink model into an Android Automotive application. The process covers project setup, native code

integration, UI design, and build configuration, culminating in a runnable application on an automotive emulator.

5.3.1 Project Setup and File Structure

First, an Android Studio project is created using the **Automotive** template with **No Activity**. This provides a clean foundation without a default UI, allowing for a custom implementation. After creation, the project's folder structure is modified to accommodate the C++ model files.

1. **Open Android Studio**, go to **File > New > New Project**, and select the **Automotive** template. [10]
2. Choose **No Activity** when prompted and complete the remaining steps. [10]
3. Create a new folder named **cpp** under automotive/src/main.
4. Copy all the Simulink-generated C++ source files (.cpp and .h) and the native-lib.cpp file into this new cpp folder.

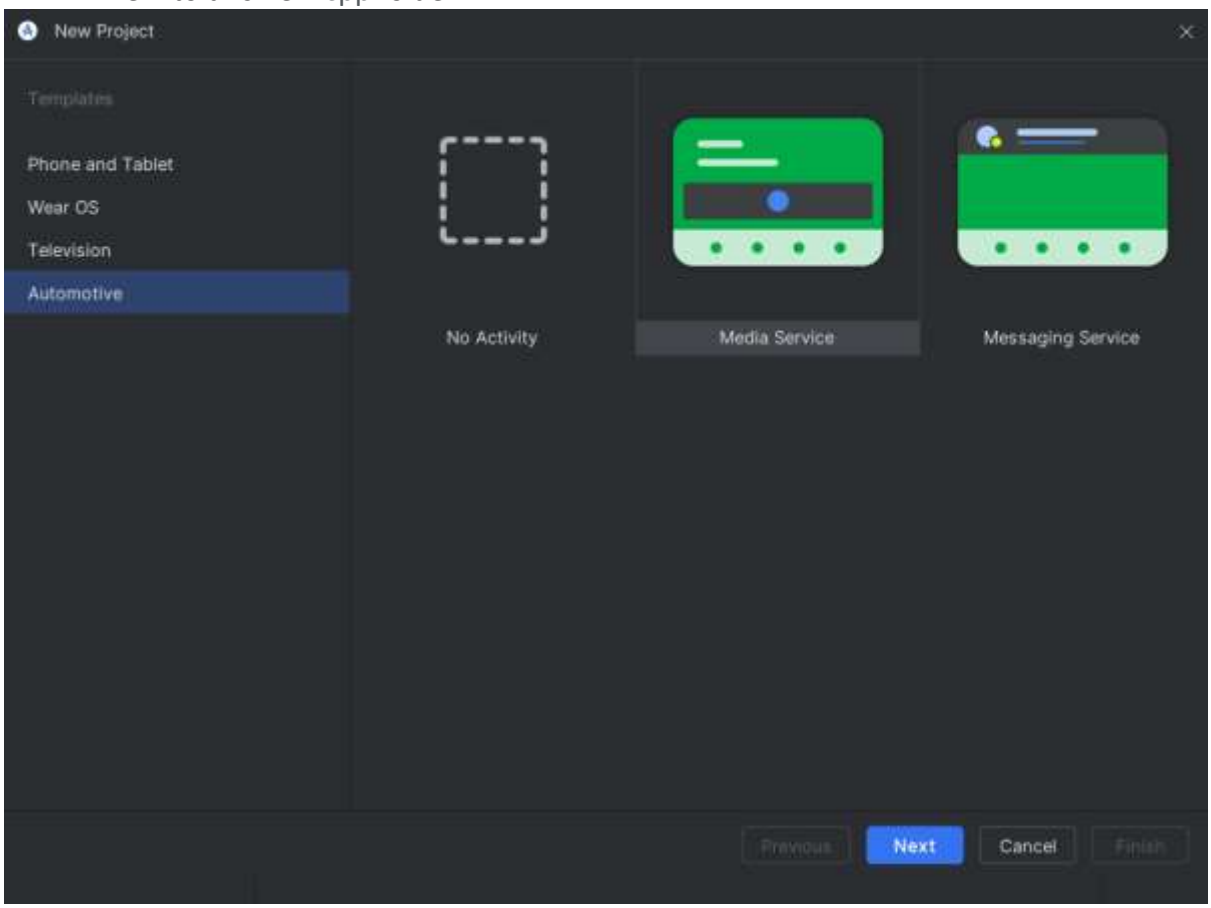


Figure 34: Android Studio New Project

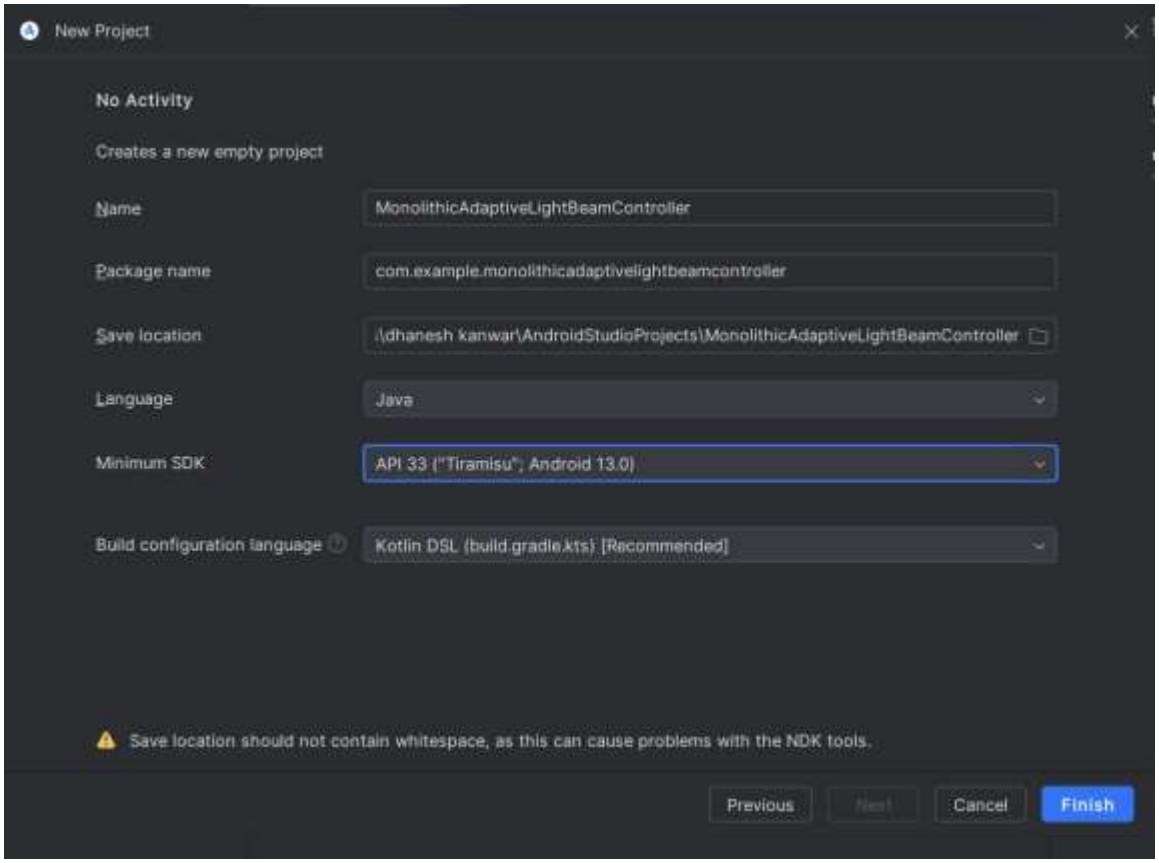


Figure 35: Android Studio New project setup

After Finish android studio generate the following folders

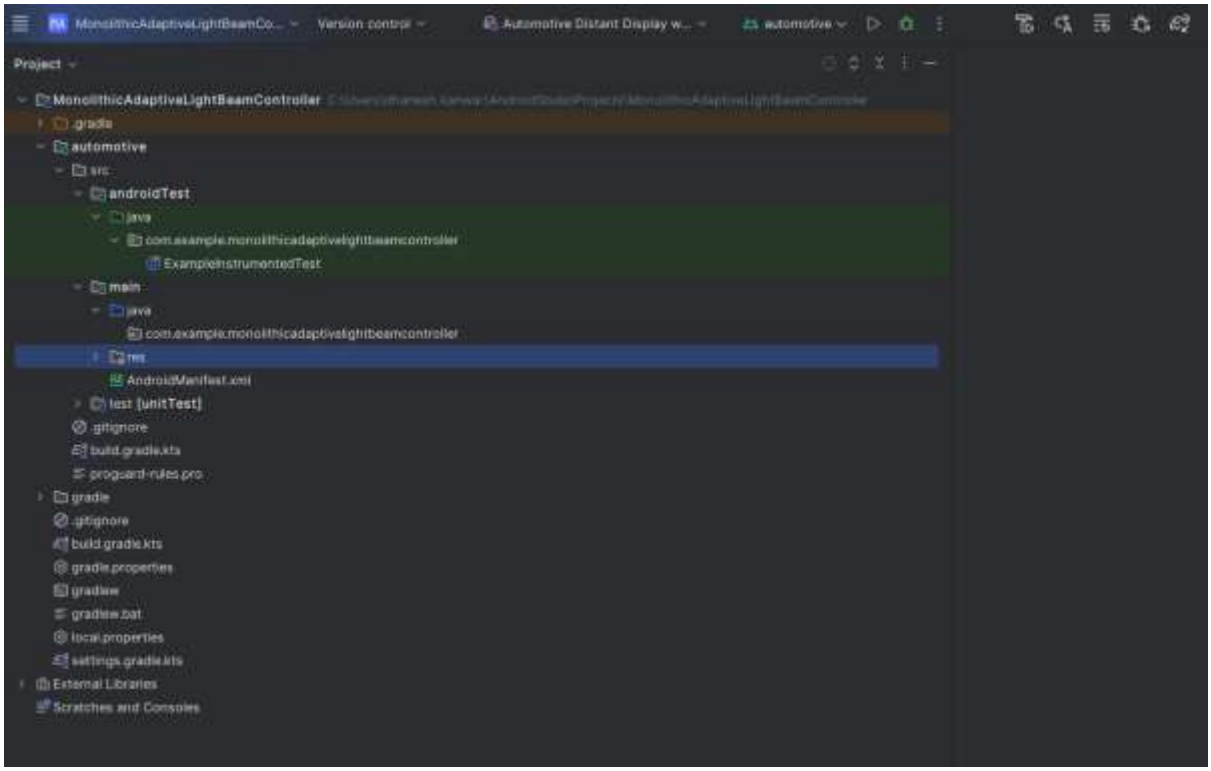


Figure 36: Android Studio folder structure

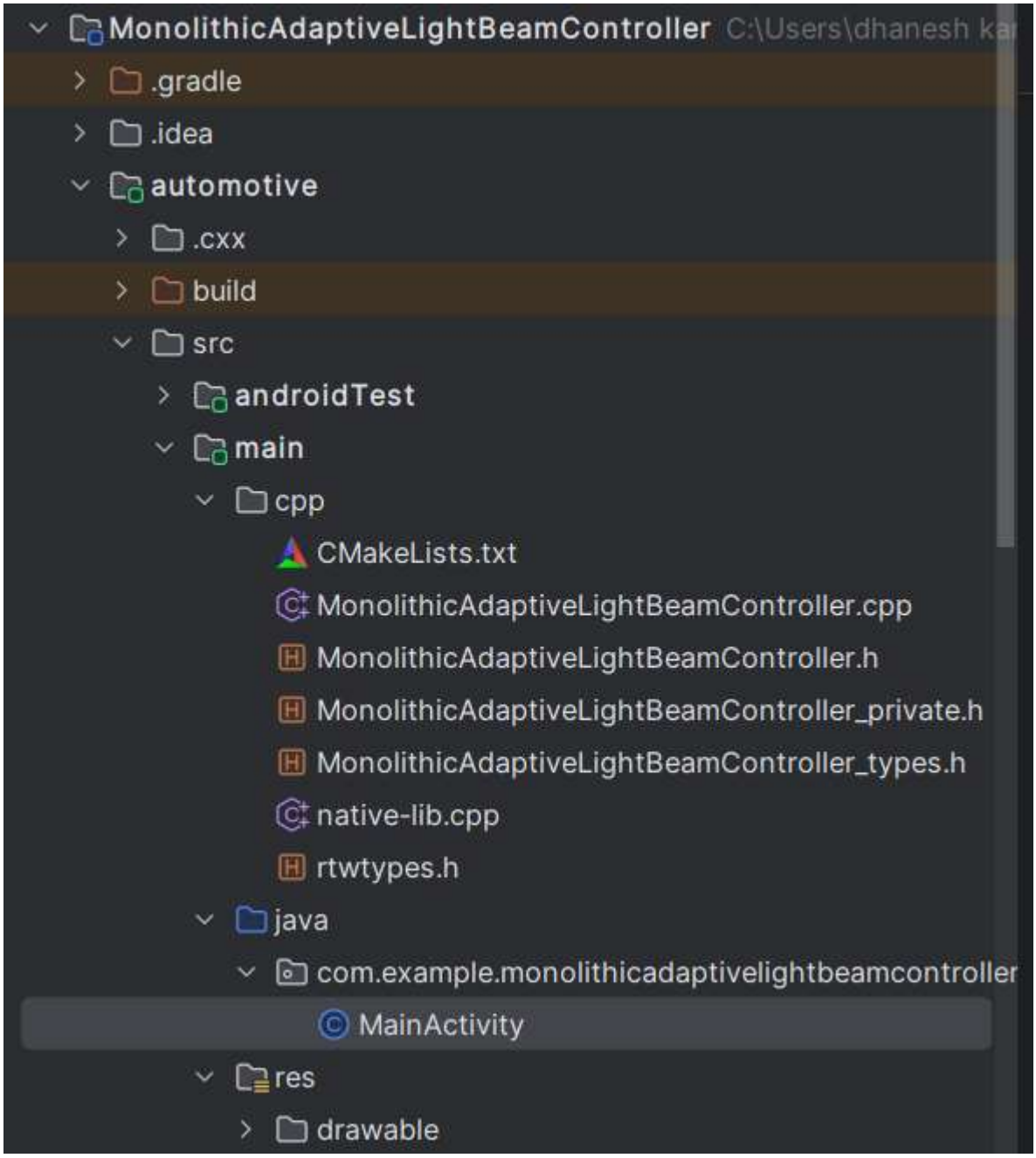


Figure 37: Monolithic ALBC File structure in Android Studio

5.3.2 Native C++ and JNI Bridge (native-lib.cpp)

The **native-lib.cpp** file acts as a **JNI wrapper**, bridging the Java application and the C++ model. It exposes the model's core functions (initialize, step, and terminate) to the Java layer.

C++

```
#include <jni.h>
#include <string>
#include <sstream>
#include "MonolithicAdaptiveLightBeamController.h"

// Global model instance
```

```
static MonolithicAdaptiveLightBeamController model;

extern "C" JNIEXPORT void JNICALL
Java_com_example_monolithicadaptivelightbeamcontroller_MainActivity_initModel(
    JNIEnv* env, jobject /* this */) {
    model.initialize();
}

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_monolithicadaptivelightbeamcontroller_MainActivity_stepModel(
    JNIEnv* env, jobject /* this */,
    jdouble ambientLight,
    jdouble roadCurvature,
    jdouble steeringAngle,
    jboolean vehicleOncoming,
    jdouble vehicleSpeed) {

    // Prepare and set model inputs
    MonolithicAdaptiveLightBeamController::ExtU_MonolithicAdaptiveLightB_T
input{};
    input.AmbientLight = static_cast<real_T>(ambientLight);
    input.RoadCurvature_permeter = static_cast<real_T>(roadCurvature);
    input.SteeringAngle_degree = static_cast<real_T>(steeringAngle);
    input.VehicleOncoming = static_cast<real_T>(vehicleOncoming);
    input.VehicleSpeed_km_h = static_cast<real_T>(vehicleSpeed);

    model.setExternalInputs(&input);
    model.step();

    // Retrieve model outputs
    const auto& output = model.getExternalOutputs();

    // Prepare JSON response
    std::ostringstream oss;
    oss << "{"
        << "\"beamAngle\":" << output.BeamAngle << ","
        << "\"beamRange\":" << output.BeamRange
        << "}";

    return env->NewStringUTF(oss.str().c_str());
}

extern "C" JNIEXPORT void JNICALL
Java_com_example_monolithicadaptivelightbeamcontroller_MainActivity_terminateModel
(
    JNIEnv* env, jobject /* this */) {
    model.terminate();
}
```

Key Functions:

- **initModel()**: Calls the model's initialize() method to set it up.
- **stepModel()**: Receives inputs from Java, converts them to C++ types, executes a single step of the model's logic, and returns the outputs as a **JSON string**.
- **terminateModel()**: Calls the model's terminate() method to clean up resources when the app closes.

5.3.3 Native Build Configuration (CMakeLists.txt)

A **CMakeLists.txt** file is created in the cpp folder to define how the native source files are compiled and linked.

CMake

```
cmake_minimum_required(VERSION 3.22.1)

project("monolithicadaptivelightbeamcontroller")

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -static-libstdc++")
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_library(${CMAKE_PROJECT_NAME} SHARED
    native-lib.cpp
    MonolithicAdaptiveLightBeamController.cpp
)

find_library(log-lib log)

target_link_libraries(${CMAKE_PROJECT_NAME}
    android
    log
    c++
)
```

- add_library(): Combines native-lib.cpp and MonolithicAdaptiveLightBeamController.cpp into a single **shared library** (.so).
- target_link_libraries(): Links the generated library with essential Android system libraries, such as log for logging and c++ for the C++ standard library.

5.3.4 Android Application Controller (MainActivity.java)

The **MainActivity.java** file is the central controller for the application's UI and logic. It handles the complete workflow, from user input to displaying the final output.

Code Excerpts and Explanation:

1. **Load Native Library**: The static block loads the compiled C++ shared library, libmonolithicadaptivelightbeamcontroller.so.

   Java

```
static {
    System.loadLibrary("monolithicadaptivelightbeamcontroller");
}
```

2. **Declare Native Methods**: These declarations inform the Java compiler that the methods are implemented in native C++ code.

   Java

```
public native void initModel();
public native String stepModel(double ambientLight, double roadCurvature, double
steeringAngle, boolean vehicleOncoming, double vehicleSpeed);
public native void terminateModel();
```

3. **UI Initialization**: In the onCreate method, the UI elements from the XML layout are linked to Java variables.

4. **Button Click Handler**: An OnClickListener is set up on the analyzeBtn. When clicked, it reads user inputs, calls the **stepModel()** native method, and processes the JSON output.
    - **Input Validation**: It checks if the input values are within valid ranges (e.g., ambient light from 0 to 100).
    - **JSON Parsing**: The returned JSON string is parsed to extract the beamAngle and beamRange.
    - **UI Update**: The extracted values are used to update the TextViews on the screen.
5. **Lifecycle Management**: The onDestroy() method ensures the native model is properly shut down when the activity is destroyed.

Java

```
@Override
protected void onDestroy() {
    super.onDestroy();
    terminateModel();
}
```

5.3.5 User Interface Layout (main_activity.xml)

This XML file defines the user interface for the app. It includes input fields for various parameters and displays for the calculated outputs.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="24dp">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:gravity="center_horizontal">
        <TextView ... android:text="Adaptive Light Beam Controller" />
        <EditText android:id="@+id/inputAmbientLight" ... />
        <EditText android:id="@+id/inputRoadCurvature" ... />
        <EditText android:id="@+id/inputVehicleSpeed" ... />
        <EditText android:id="@+id/inputSteeringAngle" ... />
        <Switch android:id="@+id/inputVehicleOncoming" ... />
        <Button   android:id="@+id/analyzeBtn"   ...   android:text="Analyze   Beam
Intensity and Angle" />
        <TextView    android:id="@+id/intensityOutput"    ...    android:text="Beam
Intensity: --" />
        <TextView android:id="@+id/BeamAngleOutput" ... android:text="Beam Angle:
" />
    </LinearLayout>
</ScrollView>
```

5.3.6 Build File Configuration (app/build.gradle.kts)

This Gradle file orchestrates the entire build process, linking the Java and C++ components.

Gradle

```
plugins {
    alias(libs.plugins.android.application)
}

android {
    namespace = "com.example.monolithicadaptivelightbeamcontroller"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.example.monolithicadaptivelightbeamcontroller"
        minSdk = 33
        targetSdk = 35
        versionCode = 1
        versionName = "1.0"
        externalNativeBuild {
            cmake {
                cppFlags += "-std=c++17"
                arguments += listOf("-DANDROID_STL=c++_shared")
            }
        }
        ndk {
            abiFilters += listOf("armeabi-v7a", "x86_64", "arm64-v8a")
        }
    }
    packaging {
        resources {
            pickFirsts.add("**/libc++_shared.so")
        }
    }
    buildTypes { ... }
    externalNativeBuild {
        cmake {
            path = file("src/main/cpp/CMakeLists.txt")
            version = "3.22.1"
        }
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_11
        targetCompatibility = JavaVersion.VERSION_11
    }
}
```

dependencies { ... }

- **externalNativeBuild**: Points Gradle to the CMakeLists.txt file and configures the C++ build with c++17 standard and shared runtime.
- **ndk**: Specifies the target CPU architectures (abiFilters) to optimize the final APK size.
- **packaging**: The pickFirsts rule prevents file conflicts with libc++_shared.so.

5.3.7 Android Manifest (AndroidManifest.xml)

The manifest file defines the application's components and settings.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    package="com.example.monolithicadaptivelightbeamcontroller">
    <application
        android:allowBackup="true"
        android:label="Adaptive Light Beam Controller"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AppCompat.Light.NoActionBar">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

- **<application>**: Defines global attributes like the app's name, icon, and theme.
- **<activity>**: Declares the MainActivity as the app's entry point (android.intent.action.MAIN) and makes it visible in the app launcher (android.intent.category.LAUNCHER).

5.3.8 Application Run and Verification

The final step is to build and run the application. After a successful build, the app can be launched on an Android Automotive emulator, where the UI is displayed, and the native C++ model runs the adaptive light beam control logic.(see chapter 5.6 for test cases run result on emulator)

Figure 38: UI of ALBC application on android emulator

5.3.9 Whole workflow of the monolithic adaptive light controller application

The workflow of the Android application for the monolithic model is a **complete, end-to-end process** that starts with user interaction on the Android UI and concludes with the display of the calculated results. This process is a continuous loop that runs each time the user requests a new calculation.

Workflow

1. **User Input on Android UI**

   The user enters various driving and environmental parameters—such as ambient light, road curvature, vehicle speed, steering angle, and whether a vehicle is oncoming—into the text fields and switches on the app's user interface, defined in main_activity.xml.

2. **Input Reading and Validation**

   When the user taps the Analyze Beam Intensity and Degree button, the MainActivity.java file takes control. It reads all the values entered by the user and performs a basic validation to ensure they are within a reasonable range (e.g., ambient light is between 0 and 100). If the inputs are invalid, it displays an error message on the screen.

3. **JNI Call to the Native Model**

   After validation, MainActivity.java calls the native C++ method, stepModel(), passing the user-provided inputs as arguments. This is the crucial step where the control and data are passed from the Java layer to the native C++ layer.

4. **C++ Model Execution**

   The native-lib.cpp file, acting as the JNI bridge, receives the call. It first converts the Java data types (jdouble, jboolean) into C++-compatible types (real_T). It then uses these inputs to set the external inputs of the global model instance. Finally, it executes the model's core logic by calling model.step(). This is where the Simulink-generated monolithic model performs all its calculations to determine the optimal beam angle and range.

5. **Output Generation and Return**

   Once the model has completed its step, the native-lib.cpp file retrieves the calculated outputs (beam angle and beam range). It then packages these two values into a JSON-formatted string and returns this string back to the Java layer. This is a robust way to pass structured data across the JNI bridge.

6. **JSON Parsing and UI Update**

MainActivity.java receives the JSON string from the native code. It parses the string to extract the beamAngle and beamRange values. It then formats these values and uses them to update the TextViews on the UI, displaying the final results to the user.

7. **Loop and Termination**

   The application remains in a loop, ready for the user to change the inputs and repeat the process. When the app is closed, MainActivity.java's onDestroy() method is triggered, which in turn calls the native terminateModel() function to safely shut down the C++ model and free any allocated resources. This ensures a clean exit and prevents memory leaks.

5.4 Code Integration in Android Studio for SOA models

This section outlines the step-by-step process of integrating the Service-Oriented Architecture (SOA) models, generated from Simulink, into an Android Studio project. The methodology ensures a robust, modular, and scalable system by separating the native C++ code from the Java application layer and utilizing the Android Native Development Kit (NDK).

5.4.1 Project Setup in Android Studio

The integration process begins by creating a new Android Studio project.

1. **Open Android Studio**. [10]
2. Navigate to **File → New → New Project**. [10]
3. From the project templates, select the **Automotive** category. [10]
4. When prompted to select an activity, choose **No Activity** to create a foundational project without a default user interface, which is ideal for a back-end service-oriented application. [10]
5. Complete the remaining project setup options and click **Finish**. [10]

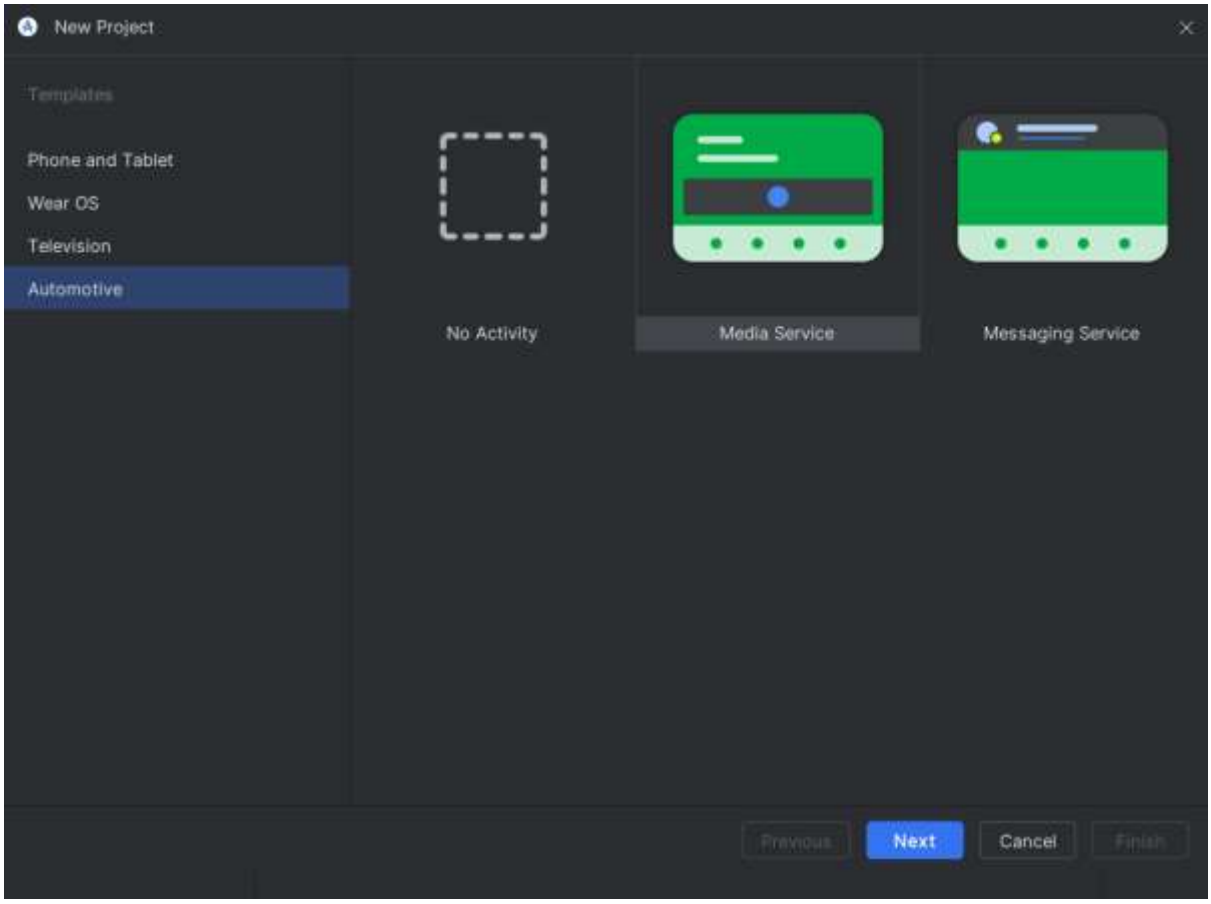


Figure 39: SOA Automotive new project

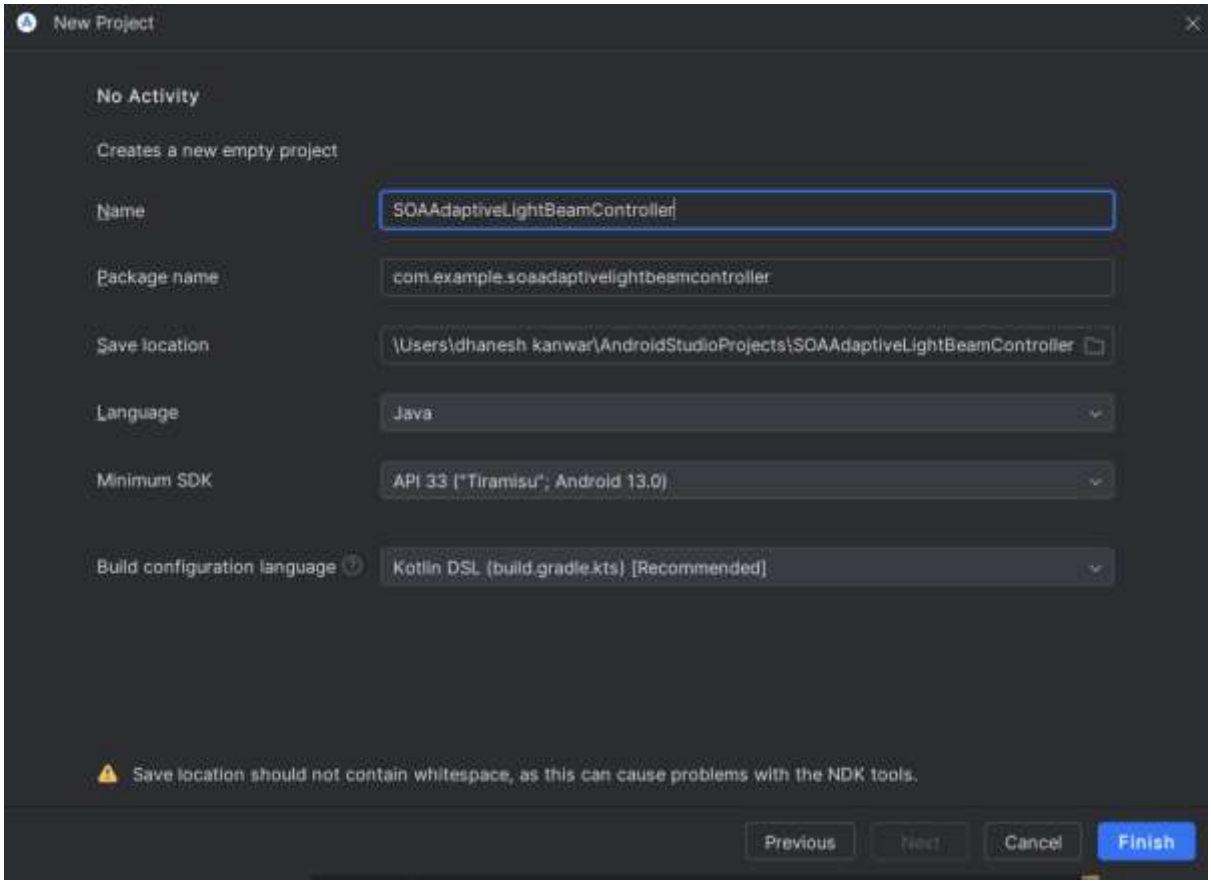


Figure 40: SOA ALBC project creation

After Finish android studio generate the following folders

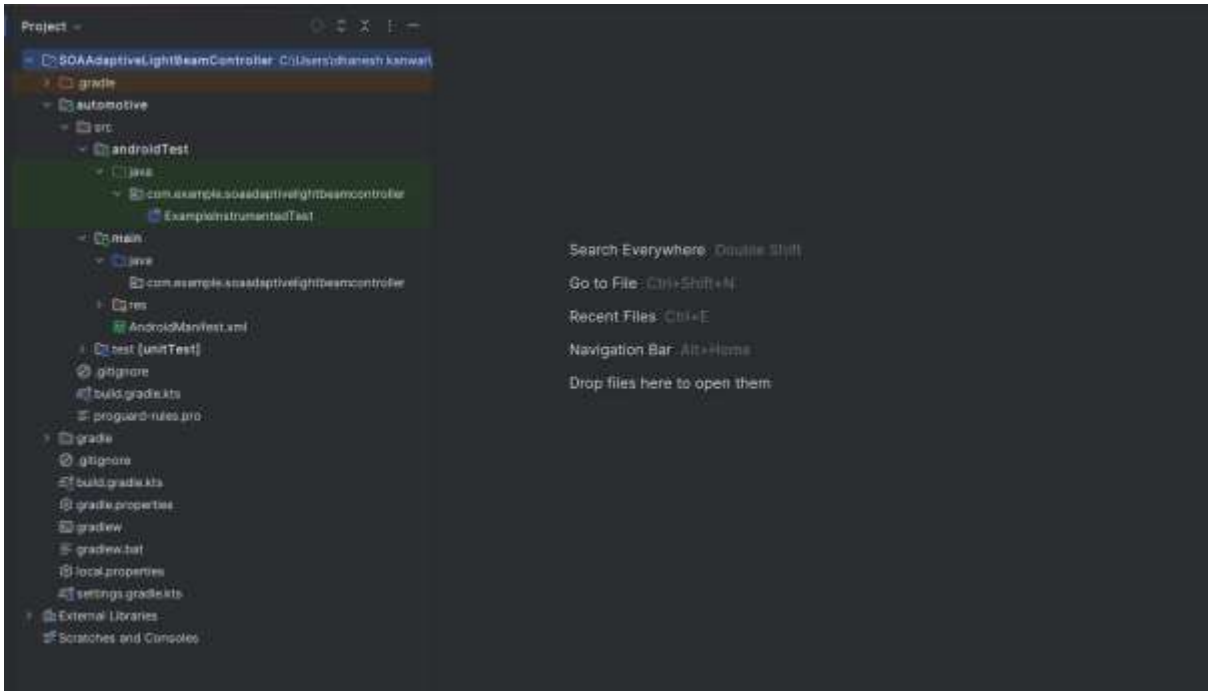


Figure 41: SOA ALBC folders generated after finish

5.4.2. Project Folder Structure

After the project is generated, the file structure is modified to accommodate the SOA models. A new folder named cpp is created under the automotive/src/main directory. This folder will house all the native C++ source code. Inside the cpp folder, a dedicated subdirectory is created for each service: AmbientLightService, BeamAngleController, RoadCurvatureService, SteeringAngleService, VehicleOncomingService, and VehicleSpeedService. AdaptiveLightController main application folder is created for the source code of main application while shared folder is created for common utility files. A top-level CMakeLists.txt file is placed directly in the cpp folder to manage the entire native build process.[2][3][5][7] The final structure of the cpp folder is shown below.

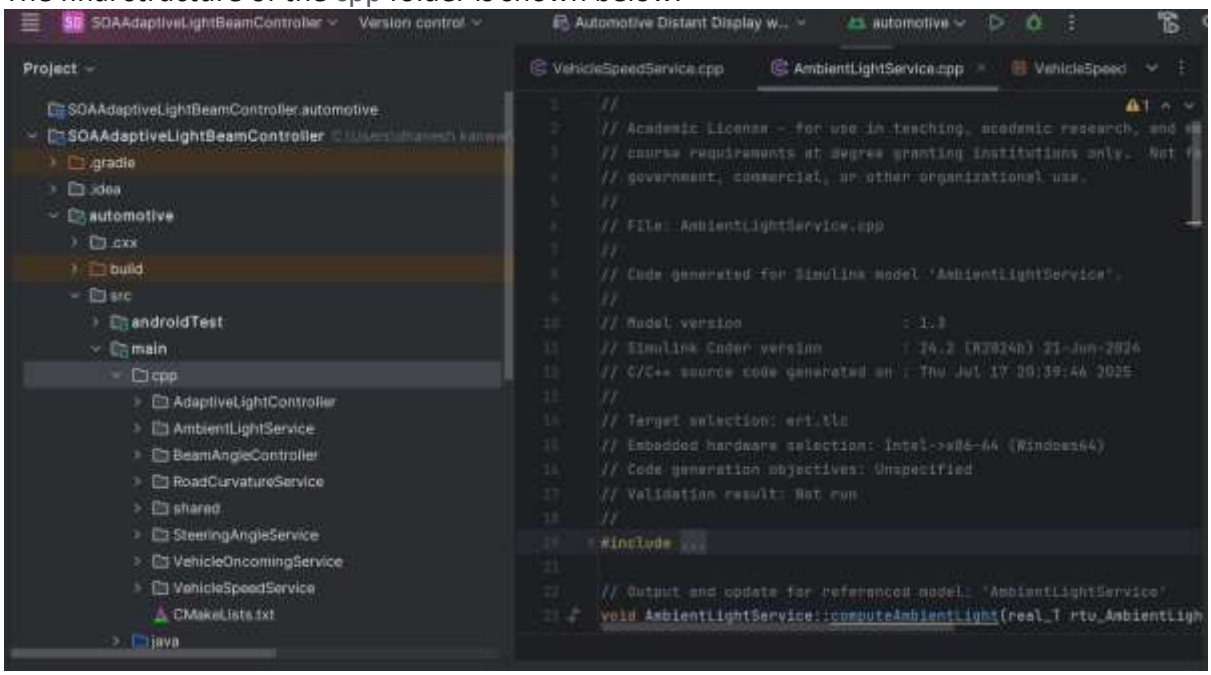


Figure 42: SOA ALBC folders to create

5.4.3. File structure of Single Service

Within each service folder (e.g., AmbientLightService), the Simulink-generated source code (.cpp and .h files) is placed. An additional CMakeLists.txt file is created in each service folder to build a shared library for that service. [2][3][5][7]

An example of the CMakeLists.txt file for the AmbientLightService is as follows:

CMake

```
cmake_minimum_required(VERSION 3.22.1)

project("AmbientLightService")

add_library(AmbientLightService SHARED
        AmbientLightService.cpp
)

# Include local headers + shared headers
target_include_directories(AmbientLightService PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}
```

```
)

# Link against shared utils + Android log
find_library(log-lib log)

target_link_libraries(AmbientLightService PRIVATE
      shared
      ${log-lib}
      c++_shared
)
```

Breakdown of AmbientLightService CMake file:

- cmake_minimum_required(VERSION 3.22.1): Ensures compatibility with the specified CMake version.
- project("AmbientLightService"): Defines the name of the project.
- add_library(AmbientLightService SHARED ...): Creates a **shared library** (.so file) from the source file.
- target_include_directories(...): Specifies the include paths for header files.
- find_library(log-lib log): Locates the Android system logging library.
- target_link_libraries(...): Links this library to its dependencies, including the shared utility library and the Android log library.

This process is repeated for each service, placing its respective source code and a CMakeLists.txt file in its dedicated folder.

The file structure of the AmbientLightService folder:

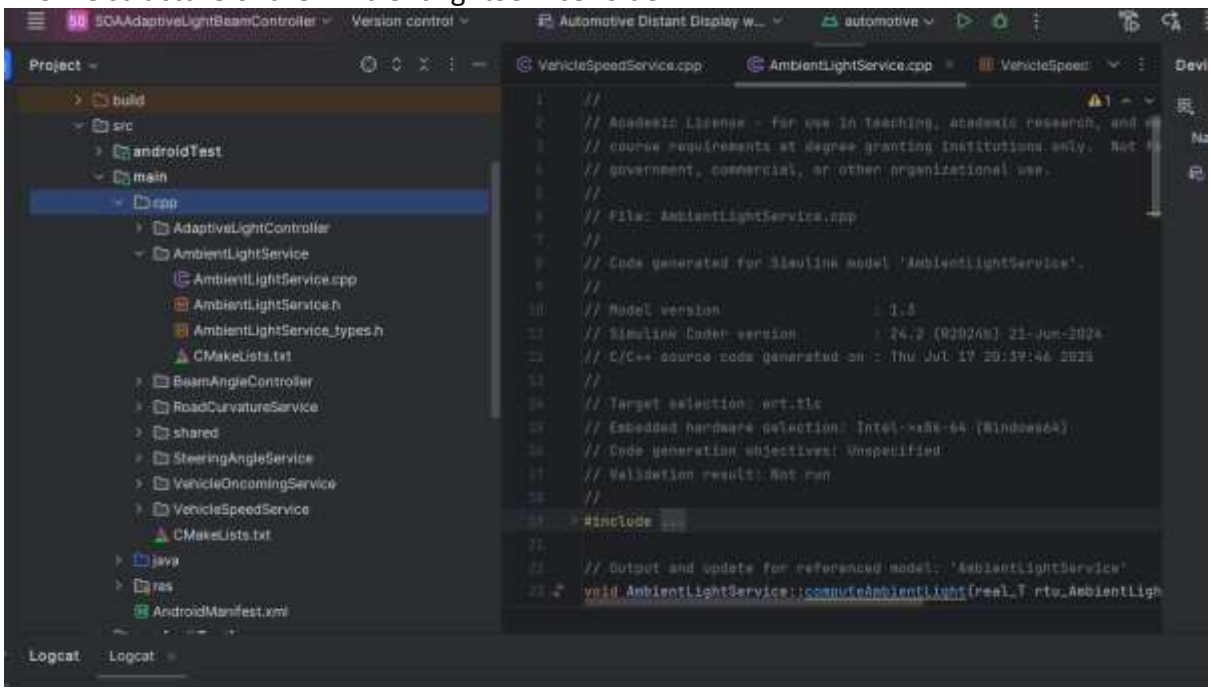


Figure 43: File structure of single service

5.4.4. File structure of the Main Application (AdaptiveLightController)

The main application folder contains all components required to compile and execute the Simulink-generated Adaptive Light Controller within the Android NDK environment. This folder serves as the integration point between the automatically generated C++ models, the

SOA service modules, and the Android application layer. The structure and purpose of the key elements in this directory are described below. [2][3][5]

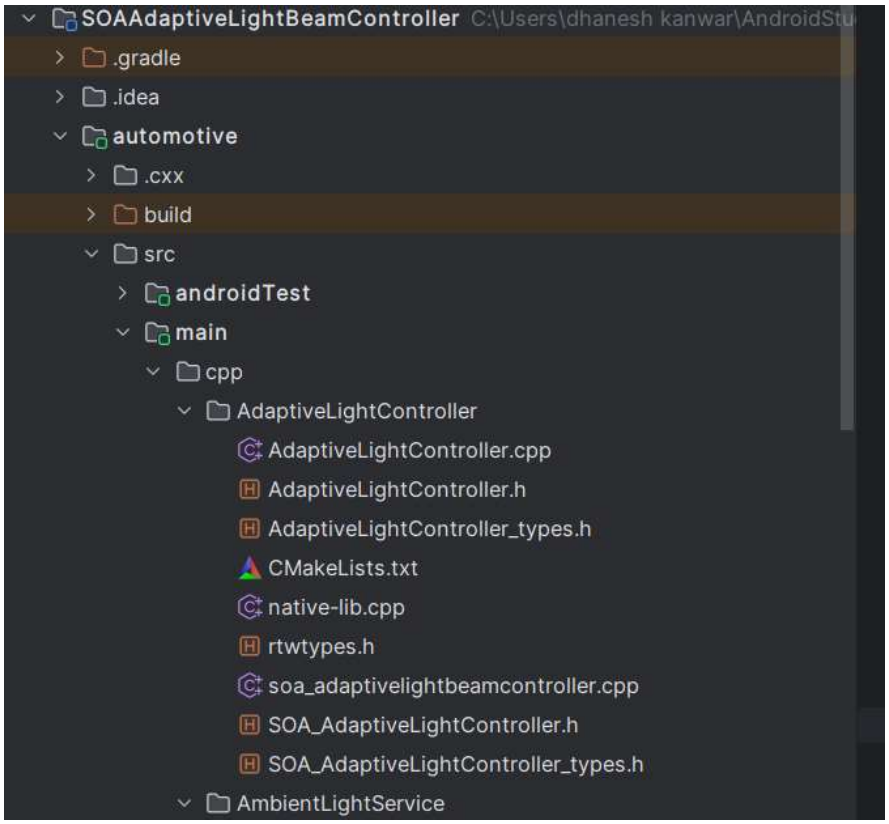


Figure 44: File structure of AdaptiveLightController

5.4.4.1. Integration of Simulink-Generated Source Code

The core functionality of the adaptive lighting system originates from the Simulink model. Two source files are generated during the code generation process:

5.4.4.1.1. Top-Level Wrapper Model (e.g., SOA_AdaptiveLightController)

This file is the top-level wrapper model generated by Simulink. It serves as the integration layer between the Android native runtime and the core adaptive-light controller. Its responsibilities include:

- Receiving raw input signals from the Android system or JNI layer.
- Triggering the appropriate *Function-Call Subsystems* from the core model.
- Managing model initialization, step execution, and termination.
- Orchestrating communication with service-oriented architecture (SOA) client interfaces.
- Providing the external API that is invoked from native-lib.cpp.

This file does not contain control logic itself; instead, it forwards sensor data to the core model and retrieves algorithm outputs.

5.4.4.1.2. Core Control Logic Model (AdaptiveLightController)

This file contains the actual adaptive headlight control algorithm produced from the referenced Simulink model. Its responsibilities include:

- Executing SOA service client calls (e.g., steering angle, ambient light, speed, road curvature).
- Running the Stateflow state machine that governs headlight behaviour:
    - Low Beam
    - High Beam
    - City Mode
    - Cornering Beam
- Computing the final beam angle and beam range, which are returned to the wrapper model.
- Maintaining intermediate block states and signals according to Simulink's generated structure.

This file acts as the "brain" of the system.
The wrapper model invokes the functions in this file during every control cycle.

5.4.4.2. Creation of the CMake Build Configuration

A dedicated **CMakeLists.txt** file is created in the main application folder.
Its purpose is to instruct Android Studio on how to compile the native Simulink-generated code.

An example of CMakeFile.txt file :

```
cmake_minimum_required(VERSION 3.22.1)

project("AdaptiveLightController")

# Create AdaptiveLightController shared library
add_library(AdaptiveLightController SHARED
      AdaptiveLightController.cpp
      soa_adaptivelightbeamcontroller.cpp
      native-lib.cpp
)

# Include headers for this module + all dependent service modules
target_include_directories(AdaptiveLightController PUBLIC
```

```
    ${CMAKE_CURRENT_SOURCE_DIR}                # AdaptiveLightController headers
    ${CMAKE_CURRENT_SOURCE_DIR}/../BeamAngleController
    ${CMAKE_CURRENT_SOURCE_DIR}/../RoadCurvatureService
    ${CMAKE_CURRENT_SOURCE_DIR}/../SteeringAngleService
    ${CMAKE_CURRENT_SOURCE_DIR}/../VehicleOncomingService
    ${CMAKE_CURRENT_SOURCE_DIR}/../VehicleSpeedService
    ${CMAKE_CURRENT_SOURCE_DIR}/../AmbientLightService
    ${CMAKE_CURRENT_SOURCE_DIR}/../shared
)

# Find Android log library
find_library(log-lib log REQUIRED)

# Link against Android system libs, C++ runtime, and all service libraries
target_link_libraries(AdaptiveLightController PRIVATE
    android
    ${log-lib}
    c++_shared
    BeamAngleController
    RoadCurvatureService
    SteeringAngleService
    VehicleOncomingService
    VehicleSpeedService
    AmbientLightService
    shared
)
```

Explanation of the CMake file:

(1) Defining the CMake and Project Version

```
cmake_minimum_required(VERSION 3.22.1)
project("AdaptiveLightController")
```

This sets the minimum version of CMake required by Android Studio and names the project. It ensures compatibility with Android's NDK toolchain.

(2) Creating the Shared Native Library

```
add_library(AdaptiveLightController SHARED
        AdaptiveLightController.cpp
        soa_adaptivelightbeamcontroller.cpp
        native-lib.cpp
)
```

- Creates a **shared library** named AdaptiveLightController.
- Includes:
    - **AdaptiveLightController.cpp** → *core control logic model*

- **soa_adaptivelightbeamcontroller.cpp** → *wrapper/top-level model*
- **native-lib.cpp** → *JNI interface* connecting C++ to Android Java/Kotlin layer

CMake compiles these into libAdaptiveLightController.so, which is loaded by the Android application at runtime.

(3) Including Header Files for All Modules

```
target_include_directories(AdaptiveLightController PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}
        ${CMAKE_CURRENT_SOURCE_DIR}/../BeamAngleController
        ${CMAKE_CURRENT_SOURCE_DIR}/../RoadCurvatureService
        ${CMAKE_CURRENT_SOURCE_DIR}/../SteeringAngleService
        ${CMAKE_CURRENT_SOURCE_DIR}/../VehicleOncomingService
        ${CMAKE_CURRENT_SOURCE_DIR}/../VehicleSpeedService
        ${CMAKE_CURRENT_SOURCE_DIR}/../AmbientLightService
        ${CMAKE_CURRENT_SOURCE_DIR}/../shared
)
```

This section makes the header files available during compilation. It includes:

- Headers for the main adaptive lighting model
- All SOA-dependent service modules (Beam Angle, Steering Angle, Road Curvature, etc.)
- Shared utility headers

This is necessary because the wrapper model and core model both make calls to these service modules.

(4) Locating Android Logging Library

find_library(log-lib log REQUIRED)

Android requires this to support native logging via __android_log_print().

(5) Linking All Required Libraries

```
target_link_libraries(AdaptiveLightController PRIVATE
        android
        ${log-lib}
        c++_shared
        BeamAngleController
        RoadCurvatureService
        SteeringAngleService
        VehicleOncomingService
        VehicleSpeedService
        AmbientLightService
        shared
)
```

This links:

Android system libraries

- android
- log
- c++_shared (C++ runtime)

All service module libraries

These are external libraries compiled elsewhere in the project:

- BeamAngleController
- RoadCurvatureService
- SteeringAngleService
- VehicleOncomingService
- VehicleSpeedService
- AmbientLightService
- shared (common utilities)

This ensures that the adaptive lighting controller can call all SOA service functions.

5.4.4.3 Implementation of the JNI Interface (native-lib.cpp)

The JNI bridge, implemented in the native-lib.cpp file, acts as the communication layer between the Java application and the C++ SOA models. It exposes C++ functions that can be called directly from Java and handles the data type conversion between the two languages.

C++

```
#include <jni.h>
#include <string>
#include <sstream>
#include "BeamAngleController.h"
#include "RoadCurvatureService.h"
#include "SteeringAngleService.h"
#include "VehicleOncomingService.h"
#include "VehicleSpeedService.h"
#include "AdaptiveLightController.h"
#include "SOA_AdaptiveLightController.h"

// Instantiate global service objects
static BeamAngleController beamAngleController;
static RoadCurvatureService roadCurvatureService;
static SteeringAngleService steeringAngleService;
static VehicleOncomingService vehicleOncomingService;
static VehicleSpeedService vehicleSpeedService;
static AdaptiveLightController adaptiveLightController;
static SOA_AdaptiveLightController soaController;

// ... (JNIEXPORT functions for individual services as provided in the prompt)

// === AdaptiveLightController step ===
extern "C" JNIEXPORT jstring JNICALL
```

```
Java_com_example_soa_1adaptivelightbeamcontroller_AdaptiveLightController_nativeSt
epAdaptiveLightController(
    JNIEnv* env, jobject /* this */,
    jdouble ambientLight,
    jdouble roadCurvature,
    jdouble steeringAngle,
    jboolean vehicleOncoming,
    jdouble vehicleSpeed) {
    // ... (code as provided in the prompt)
}

// === SOAAdaptiveLightController.initModel ===
extern "C" JNIEXPORT void JNICALL
Java_com_example_soa_1adaptivelightbeamcontroller_SOAAdaptiveLightController_initM
odel(
    JNIEnv* env, jobject /* this */) {
    soaController.initialize();
}

// === SOAAdaptiveLightController.stepModel ===
extern "C" JNIEXPORT jstring JNICALL
Java_com_example_soa_1adaptivelightbeamcontroller_SOAAdaptiveLightController_stepM
odel(
    JNIEnv* env, jobject /* this */,
    jdouble ambientLight,
    jdouble roadCurvature,
    jdouble steeringAngle,
    jboolean vehicleOncoming,
    jdouble vehicleSpeed) {
    // ... (code as provided in the prompt)
}

// === SOAAdaptiveLightController.terminateModel ===
extern "C" JNIEXPORT void JNICALL
Java_com_example_soa_1adaptivelightbeamcontroller_SOAAdaptiveLightController_termi
nateModel(
    JNIEnv* env, jobject /* this */) {
    soaController.terminate();
}
```

Key Parts:

- **Global Service Objects**: static instances of each C++ service are created, ensuring they persist across JNI calls.
- **JNI-Exported Functions**: Functions are declared with the JNIEXPORT macro, following a specific naming convention to allow them to be called from the Java layer.
- **Data Marshalling**: The code demonstrates how to convert Java data types (jdouble, jboolean) to their C++ equivalents (real_T).
- **JSON String Output**: The C++ code compiles the output values (e.g., beamAngle, beamRange) into a JSON string, which is a flexible format for passing complex data back to the Java layer.

5.4.5. Root-Level CMakeLists.txt

The CMakeLists.txt file placed at the root of the cpp folder is the primary build script for the entire native component. It orchestrates the build process by including all service, shared and main application subdirectories.

CMake

```
cmake_minimum_required(VERSION 3.22.1)
project("soaadaptivelightbeamcontroller")

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Android log library
find_library(log-lib log REQUIRED)

# Add subdirectories for each module
add_subdirectory(shared)
add_subdirectory(VehicleSpeedService)
add_subdirectory(VehicleOncomingService)
add_subdirectory(AmbientLightService)
add_subdirectory(SteeringAngleService)
add_subdirectory(RoadCurvatureService)
add_subdirectory(BeamAngleController)
add_subdirectory(AdaptiveLightController)
```

Explanation:

- set(CMAKE_CXX_STANDARD 17): Enforces the C++17 standard for all native code compilation.
- add_subdirectory(...): This command is used to include and build all the subprojects defined in the respective subfolders. This ensures all services are built and their libraries are available for linking to the main application.

5.4.6. Java Application Files

The Android application is built around two primary Java files: MainActivity.java and SOAAdaptiveLightController.java.

MainActivity.java

This file contains the UI logic. It reads user input from the XML layout, calls the native C++ model via the Java wrapper, and updates the UI with the results.

Java

```
package com.example.soaadaptivelightbeamcontroller;

import android.os.Bundle;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Switch;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
import org.json.JSONException;
import org.json.JSONObject;

public class MainActivity extends AppCompatActivity {
    private SOAAdaptiveLightController soaController;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        soaController = new SOAAdaptiveLightController();
        soaController.initModel();

        // ... (UI element bindings as provided)

        analyzeBtn.setOnClickListener(v -> {
            try {
                // ... (input reading and parsing logic as provided)

                // Call native model
                String jsonResult = soaController.stepModel(
                    ambientLight,
                    roadCurvature,
                    steeringAngle,
                    vehicleOncoming,
                    vehicleSpeed
                );

                // Parse JSON result and update UI
                // ... (parsing and UI update logic as provided)

            } catch (NumberFormatException e) {
                // ... (error handling)
            } catch (JSONException e) {
                // ... (error handling)
            }
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        soaController.terminateModel();
    }
}
```

Explanation:

- **Initialization**: An instance of SOAAdaptiveLightController is created, and the native initModel() function is called to prepare the C++ environment.
- **Event Handling**: An OnClickListener is set up on the analyzeBtn to capture user inputs, pass them to the native stepModel(), and parse the returned JSON string to update the UI.
- **Lifecycle Management**: The native terminateModel() is called in the onDestroy() lifecycle method to ensure proper cleanup of C++ resources when the activity is closed.

SOAAdaptiveLightController.java

This is a simple Java wrapper class that loads the native library and declares the native methods.

```
Java
package com.example.soaadaptivelightbeamcontroller;

public class SOAAdaptiveLightController {
    static {
        System.loadLibrary("AdaptiveLightController");
```

```
    }

    public native void initModel();
    public native String stepModel(
        double ambientLight,
        double roadCurvature,
        double steeringAngle,
        boolean vehicleOncoming,
        double vehicleSpeed);
    public native void terminateModel();
}
```

Explanation:

- System.loadLibrary(...): This static block loads the shared native library named **libAdaptiveLightController.so**. This library is the final product of the CMake build process, which links all the individual service libraries together.
- **native keyword**: Declares the methods that are implemented in the C++ native-lib.cpp file.

5.4.7. User Interface Layout (activity_main.xml)

The XML layout file defines the visual components of the Android application's main screen.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="24dp">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:gravity="center_horizontal">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Adaptive Light Beam Controller"
            android:textSize="24sp"
            android:textStyle="bold"
            android:layout_marginBottom="24dp"
/>

        <EditText
            android:id="@+id/inputAmbientLight"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="AmbientLight(0.0 to 100.0)"
            android:inputType="numberDecimal"
            android:layout_marginTop="8dp"
/>
        <Button
            android:id="@+id/analyzeBtn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
```

```
            android:text="Analyze Beam Intensity and degree"
            android:layout_marginTop="20dp"
/>
        </LinearLayout>
</ScrollView>
```

Explanation:

- The layout uses a ScrollView and a LinearLayout to organize the UI elements vertically.
- **EditText and Switch**: These components allow the user to input sensor values manually for demonstration purposes.
- **Button**: The analyzeBtn is the trigger for the computation logic.
- **TextView**: Two text views display the final output values (intensityOutput and BeamAngleOutput) returned from the native model.

5.4.8. Gradle Build Files

Gradle manages the build system for both the Java and native components.

Root-Level build.gradle.kts
This is the top-level build script for the project, where plugins are declared.

Gradle

```
plugins {
    alias(libs.plugins.android.application) apply false
}
```

Explanation:

- apply false: This ensures the Android plugin is available to all modules but is not applied at the root level, which is a standard practice for multi-module projects.

Module-Level build.gradle.kts
This file contains the core build configuration for the application module.

Gradle

```
plugins {
    alias(libs.plugins.android.application)
}

android {
    namespace = "com.example.soaadaptivelightbeamcontroller"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.example.soaadaptivelightbeamcontroller"
        minSdk = 33
        targetSdk = 35
        // ... (versioning and other configs)
        externalNativeBuild {
            cmake {
                cppFlags += "-std=c++17"
                arguments += listOf("-DANDROID_STL=c++_shared")
            }
        }
        ndk {
            abiFilters += listOf("armeabi-v7a", "x86_64", "arm64-v8a")
        }
```

```
    }
    packaging {
        resources {
            pickFirsts.add("**/libc++_shared.so")
        }
    }
    // ... (buildTypes and compileOptions)
    externalNativeBuild {
        cmake {
            path = file("src/main/cpp/CMakeLists.txt")
            version = "3.22.1"
        }
    }
}
dependencies {
    // ... (dependencies)
}
```

Explanation:

- **externalNativeBuild**: This block is crucial for the NDK integration. It points Gradle to the top-level CMakeLists.txt file and sets the C++ standard and shared runtime library.
- **ndk**: The abiFilters are configured to specify the target CPU architectures, which helps reduce the size of the final APK.
- **packaging**: The pickFirsts rule prevents conflicts when multiple libraries might include the same shared C++ runtime library.

5.4.9. Android Manifest

Your provided AndroidManifest.xml file is a well-structured configuration for an Android Automotive app. It declares the app's essential components and requirements for the Android system.

Code Explanation

Here is the breakdown of your AndroidManifest.xml file, which is placed in the root of the app/src/main/ directory.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-feature
        android:name="android.hardware.type.automotive"
        android:required="true" />

    <application
        android:allowBackup="true"
        android:appCategory="audio"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.SOAAdaptiveLightBeamController">

        <activity
            android:name=".MainActivity"
            android:exported="true">
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

    </application>

</manifest>
```

Manifest and Feature Declaration

- <manifest ...>: This is the root tag of the manifest file. It defines the package name for the application and holds all the app's components.
- <uses-feature android:name="android.hardware.type.automotive" android:required="true" />: This is a crucial line for Automotive apps. It declares that the application is designed **exclusively for in-car systems**. The android:required="true" attribute ensures that the Google Play Store and other package managers will only allow this app to be installed on devices with the specified automotive hardware. This prevents it from being installed on phones or tablets.

Application and Activity

- <application ...>: This tag contains global settings that apply to all components within the app.
    - android:allowBackup="true": Permits the system to back up the app's data.
    - android:appCategory="audio": Specifies that this app belongs to the "audio" category, which helps the system understand its primary function.
    - android:icon and android:label: These attributes set the app's main icon and name as they will appear to the user.
    - android:supportsRtl="true": Enables support for right-to-left languages like Arabic and Hebrew.
    - android:theme="@style/...": Defines the overall visual style for the app.
- <activity android:name=".MainActivity" android:exported="true">: This tag declares the app's main screen, or **Activity**.
    - android:name=".MainActivity": Points to the MainActivity.java file as the class that handles this activity.
    - android:exported="true": Marks the activity as accessible to other apps. For a launcher activity (an app that can be launched from the home screen), this is mandatory on modern Android versions.
- <intent-filter>: This block is what makes the MainActivity the entry point of the application.
    - <action android:name="android.intent.action.MAIN"/>: This action indicates that the activity is the main entry point for the application, not just a supporting screen.
    - <category android:name="android.intent.category.LAUNCHER"/>: This category tells the Android system to display an icon for this activity in the app launcher, allowing the user to start the app from the home screen.

5.4.10 Whole Workflow

The complete workflow from user input to final output can be summarized as follows:

1. **User Input on Android UI**: The user enters values into the UI fields in activity_main.xml.
2. **MainActivity.java**: Reads the input values and calls soaController.stepModel(...).
3. **SOAAdaptiveLightController.java**: This wrapper class loads the libAdaptiveLightController.so library and routes the call to the native C++ method.
4. **JNI Bridge (native-lib.cpp)**: Receives the Java call, converts the data types, and orchestrates the C++ services. It calls the appropriate functions in each service to compute the final beamAngle and beamRange.
5. **Return to Java**: The JNI bridge returns a JSON string containing the computed results.
6. **MainActivity.java**: The MainActivity receives the JSON string, parses it, and updates the text views in the UI.
7. **User Output**: The user sees the calculated Beam Angle and Beam Range on the screen.

This comprehensive workflow ensures a clear separation of concerns, with the Android application handling the UI and the native C++ code performing the complex model computations. The JNI bridge serves as a robust and efficient connection between these two distinct layers.

Flow Diagram:

```
        Android UI
   (MainActivity layout views)
   • Ambient Light (lux)
   • Road Curvature (1/m)
   • Steering Angle (deg)
   • Vehicle Speed (km/h)
   • Vehicle Oncoming (switch)
```

| Reads user input

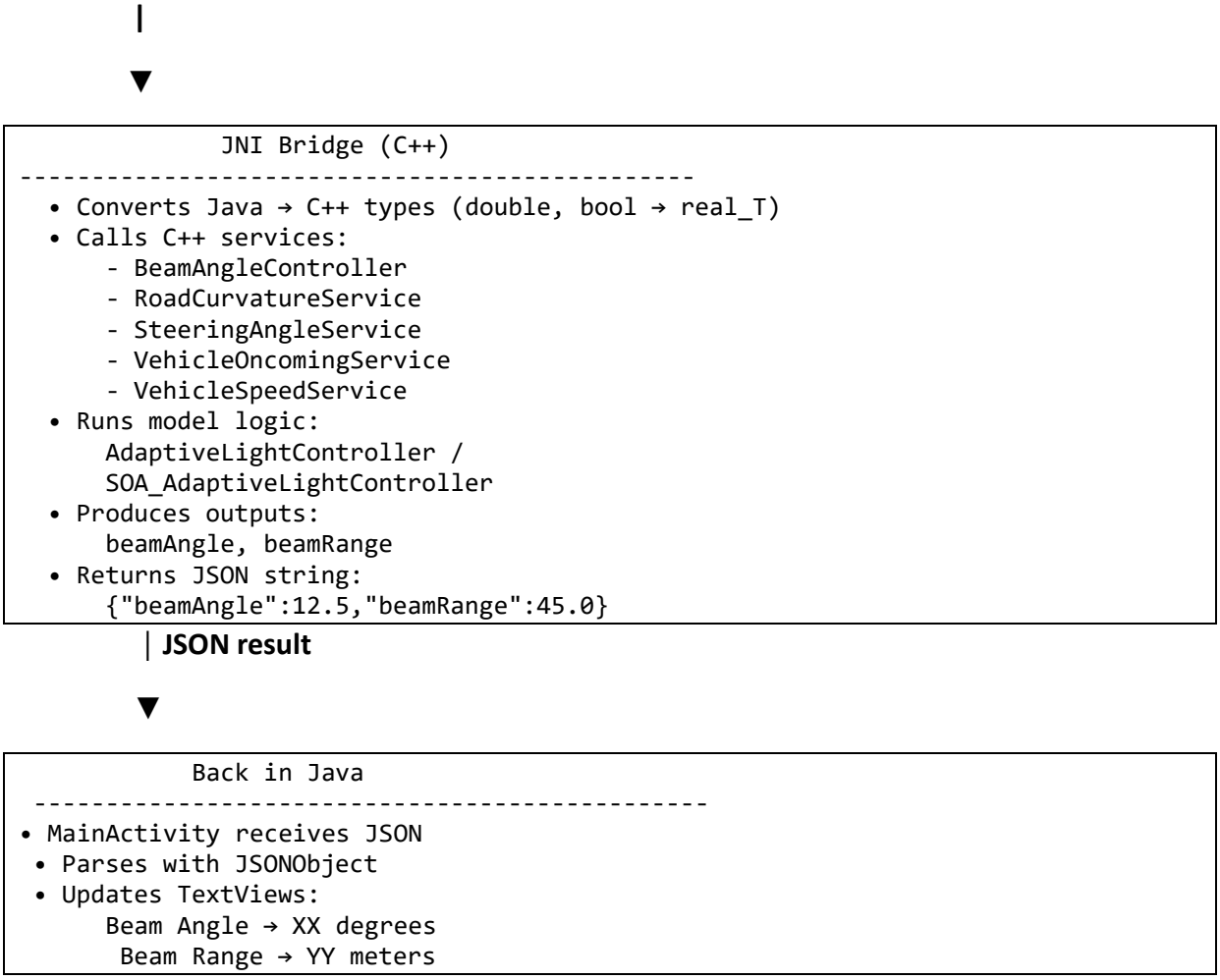
▼

```
              MainActivity.java
 ------------------------------------------
  • Collects UI values from EditText/Switch
  • Calls:
      soaController.stepModel(…)
```

| JNI call

▼

```
        SOAAdaptiveLightController.java
 ---------------------------------------------
  • Loads native library:
        System.loadLibrary("AdaptiveLightController")
  • Declares native methods:
       initModel()
       stepModel(...)
       terminateModel()
  • stepModel(...) → enters JNI
                  Native bridge
```



5.5 Configuration of Automotive Emulator

Step 1: Open AVD Manager

1. Open Android Studio.
2. Go to Tools → Device Manager (or AVD Manager in older versions).[6]
3. Click Create Device. [6]

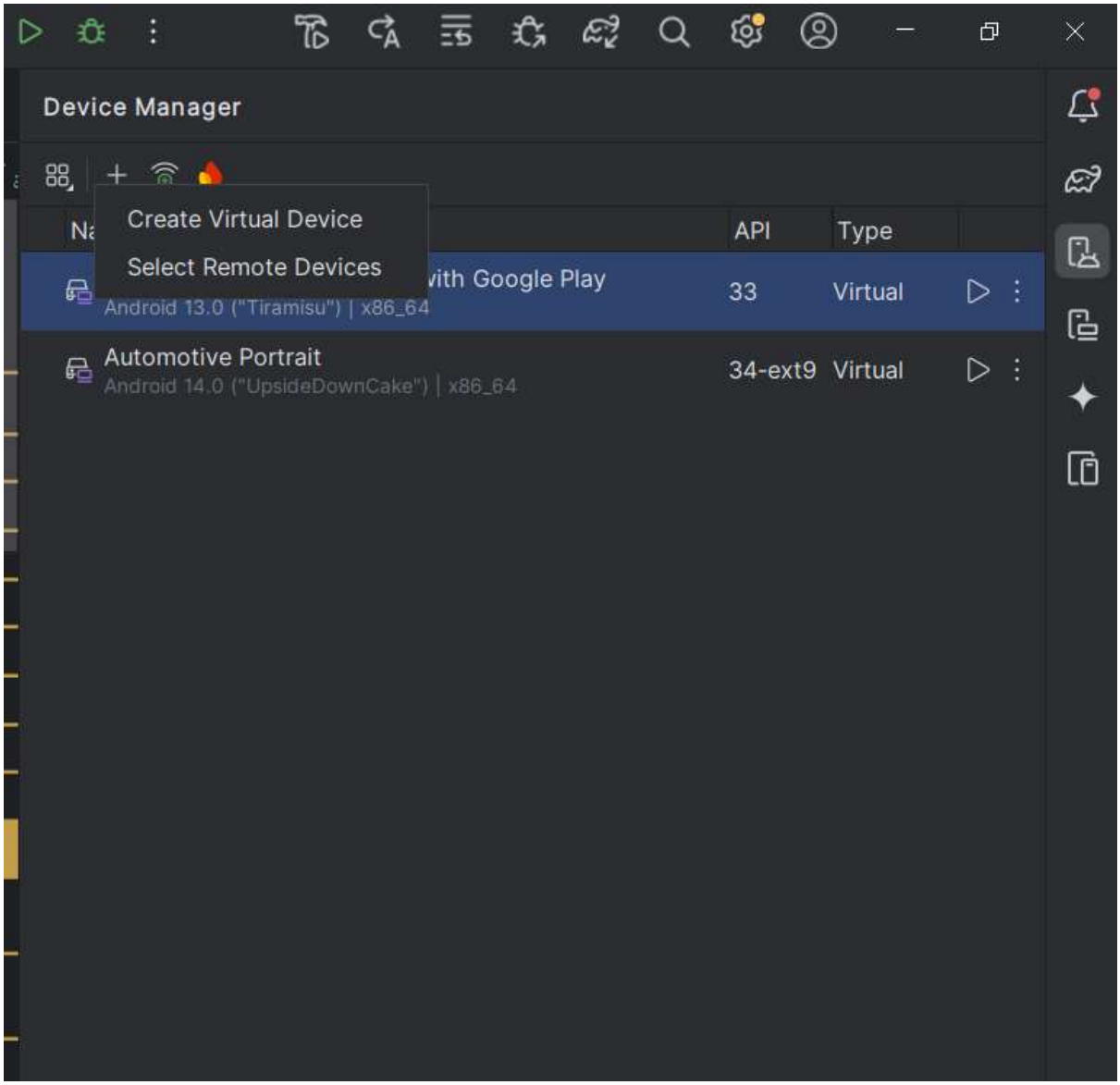


Figure 45: Create new Automotive emulator

Step 2: Select Automotive Hardware Profile

1. In the Select Hardware window, scroll down and select Automotive. [6]
2. Choose a profile, e.g., Automotive 1024x600 (or other recommended screen size). [6]
3. Click Next. [6]

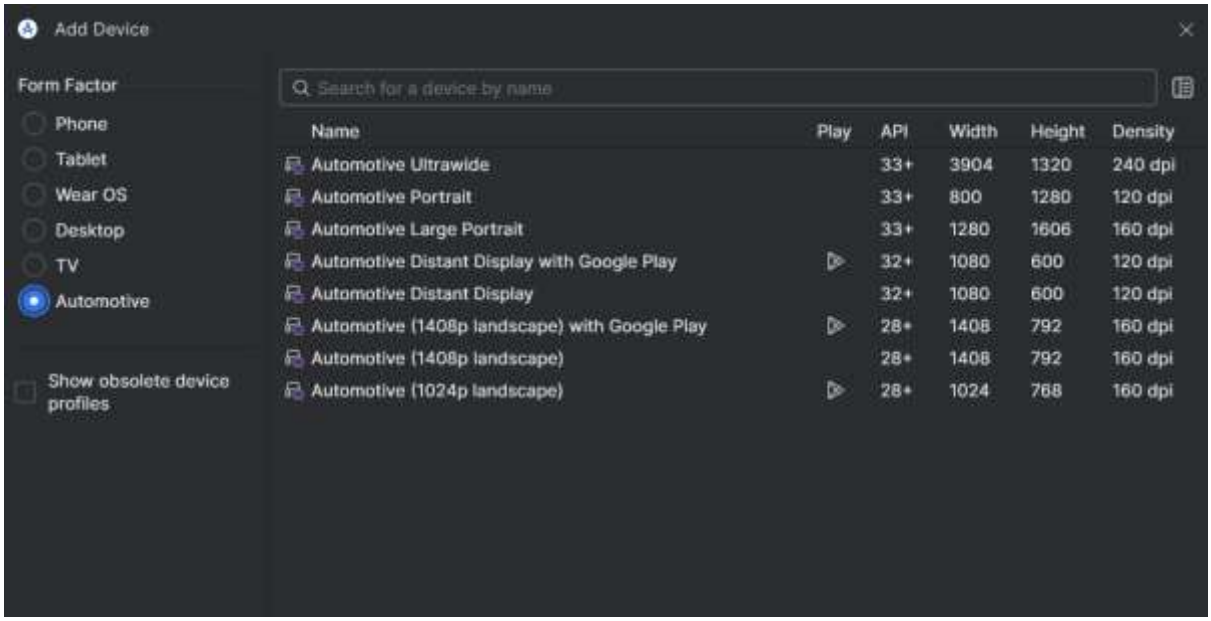


Figure 46: Selection of Automotive emulator from the list

Step 3: Choose a System Image

1. Switch to the "Recommended" or "Other Images" tab.
2. Look for Android Automotive OS images (for example, Android 12 or 13 Automotive). [6]
3. If no image is installed:
    - Click Download next to the desired image. [6]
    - Wait for the download to complete. [6]
4. Once downloaded, select the image and click Next.

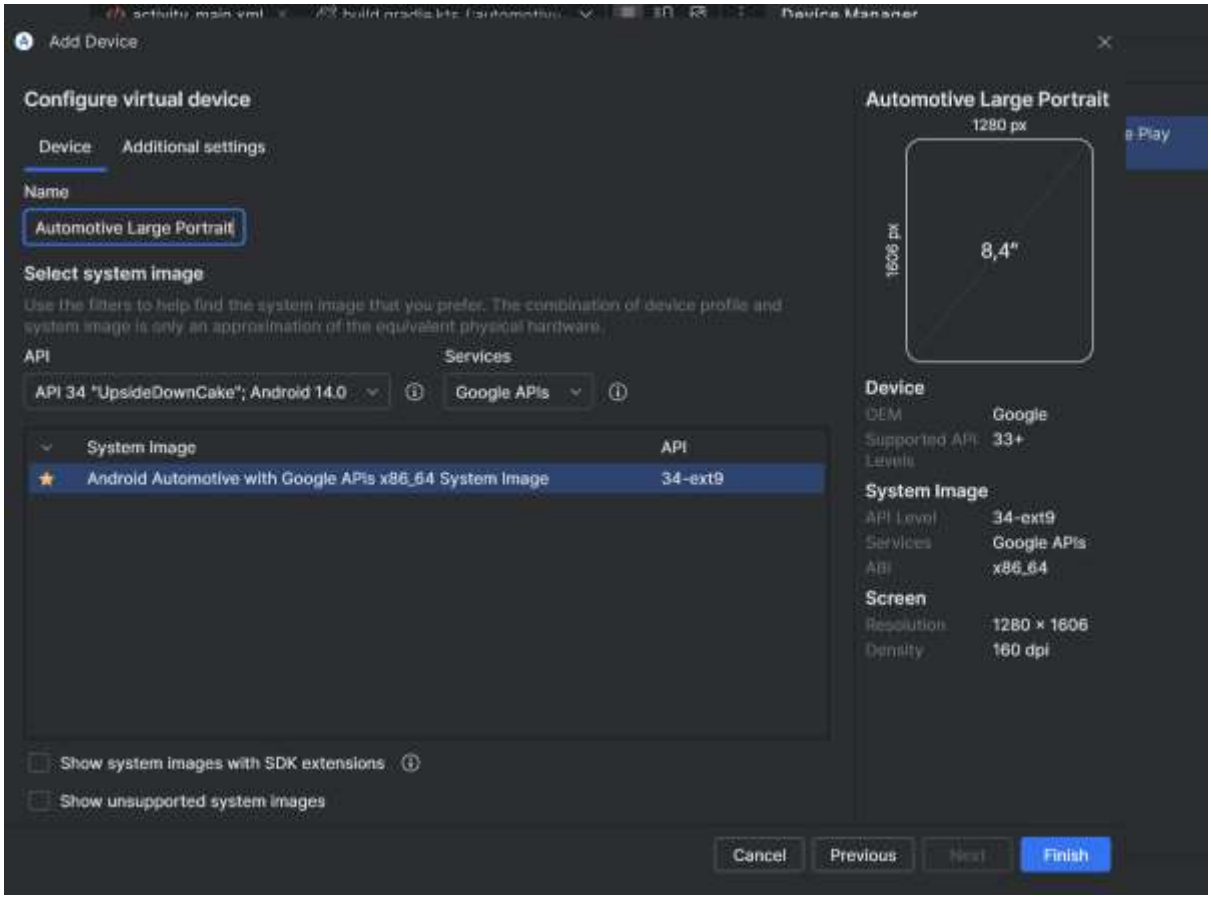


Figure 47: Selection of system image for Automotive emulator

Step 4: Configure AVD

1. Set a Name for your emulator, e.g., Automotive_Emulator_12. [6]
2. Adjust settings if needed:
    - Orientation: Landscape (most automotive screens are landscape).
    - Scale: Keep default or adjust to your monitor size.
3. Click Finish to create the AVD.

Step 5: Launch the Automotive Emulator

1. In the AVD Manager, click the Play button next to your new automotive emulator. [6]
2. Wait for it to boot—this may take a few minutes. [6]
3. Once running, you can deploy and test your automotive app just like a regular Android app.

5.6 Testing Result on emulator of monolithic and SOA applications

Test Cases:

Test cases	Vehicle Oncoming)	Vehiclespeed(km/h)	Ambient light	Steering Angle(degree)	Road curvature(degree)	Beam Range (m)	Beam Angle(degree)	Active mode
1.1	0	60	10	0	-	100	0	High Beam
1.2	1	60	10	0	-	50	0	Low Beam
1.3	0	40	10	0	-	50	0	Low Beam
1.4	0	60	25	0	-	50	0	Low Beam
2.1	0	60	10	15	4	60	Close to 4	Cornering Beam
2.2	0	60	10	5	4	100	0	High Beam
2.3	0	60	10	20	10	60	Close to 10	Cornering Beam
3.1	0	20	40	0	5	30	0	City Beam
3.2	0	60	10	0	5	100	0	High Beam
4.1	1	60	10	0	5	50	0	Low Beam

Result of Run of test cases on automotive emulator:

Figure 48: Result of Test Case 1.1

Figure 49: Result of Test Case 1.2

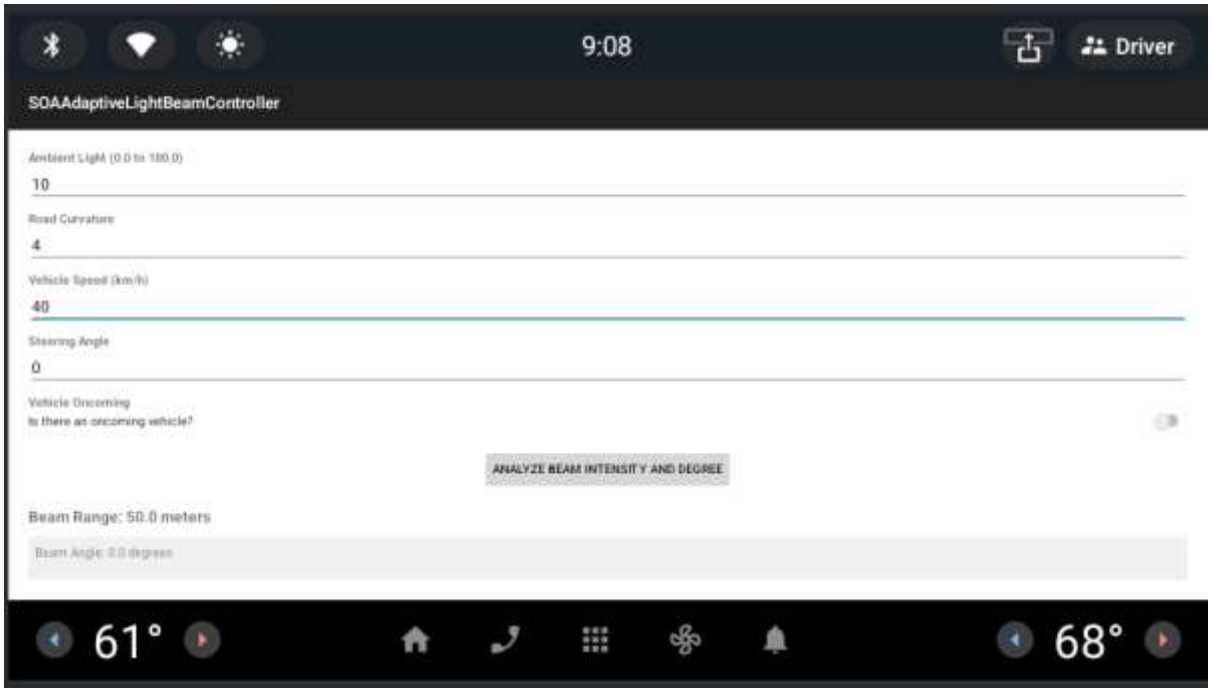


Figure 50: Result of Test Case 1.3

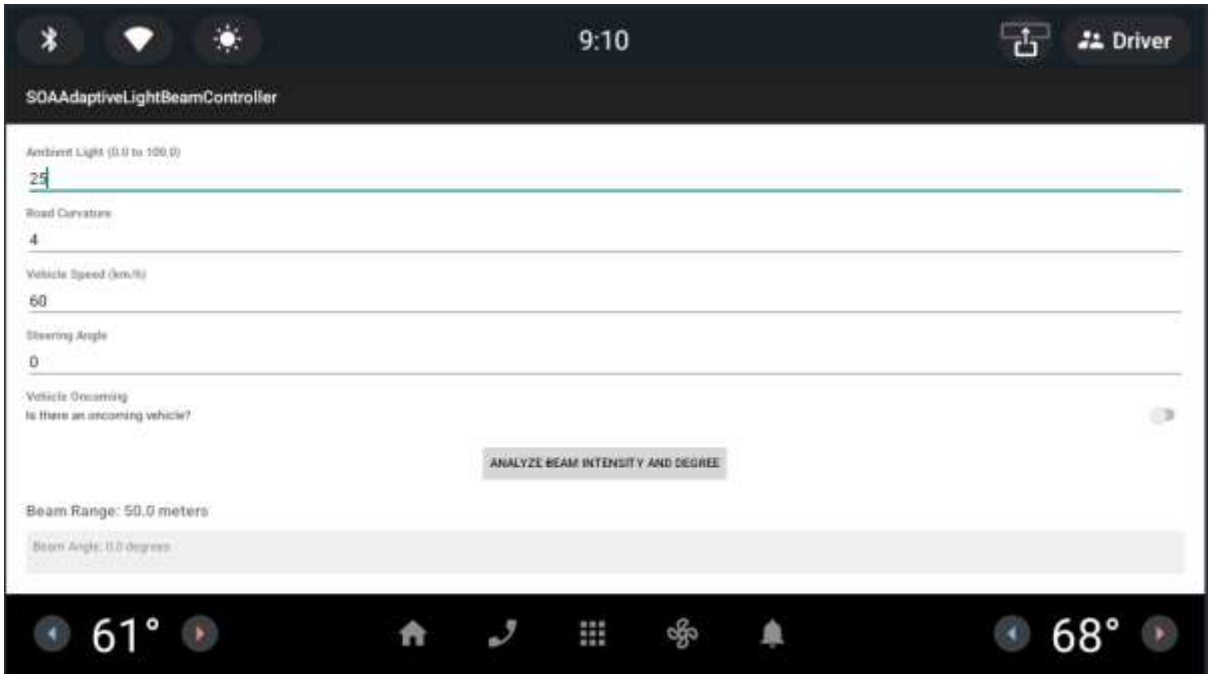


Figure 51: Result of Test Case 1.4

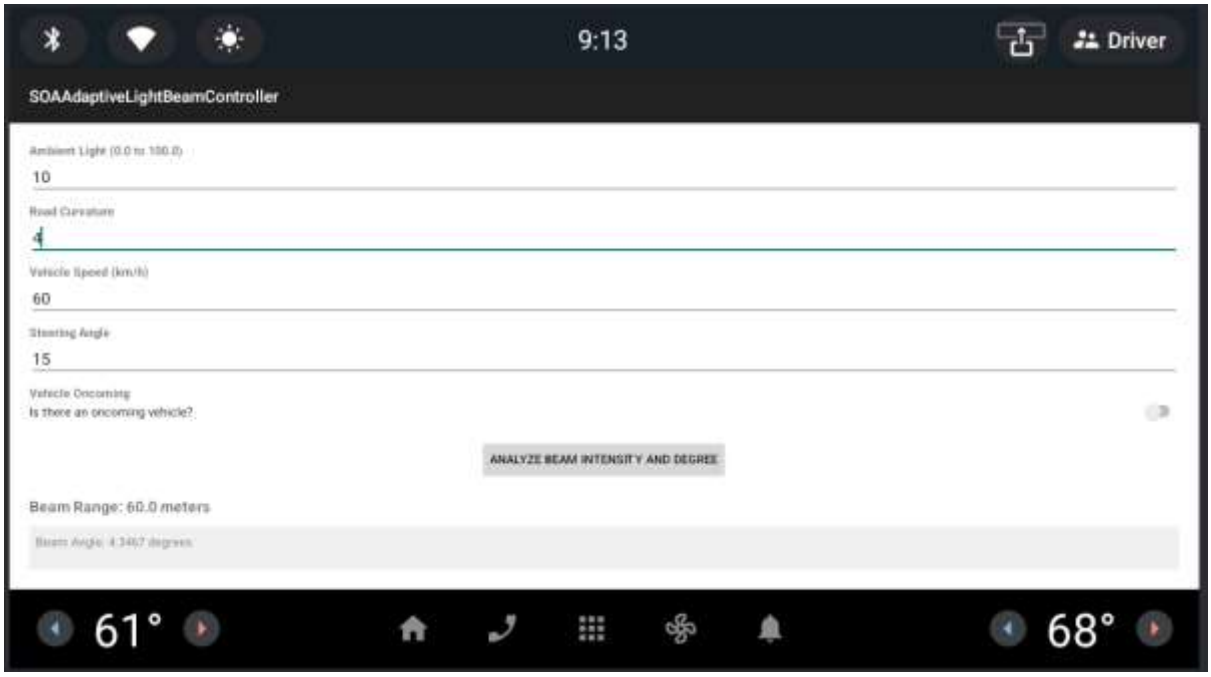


Figure 52: Result of Test Case 2.1

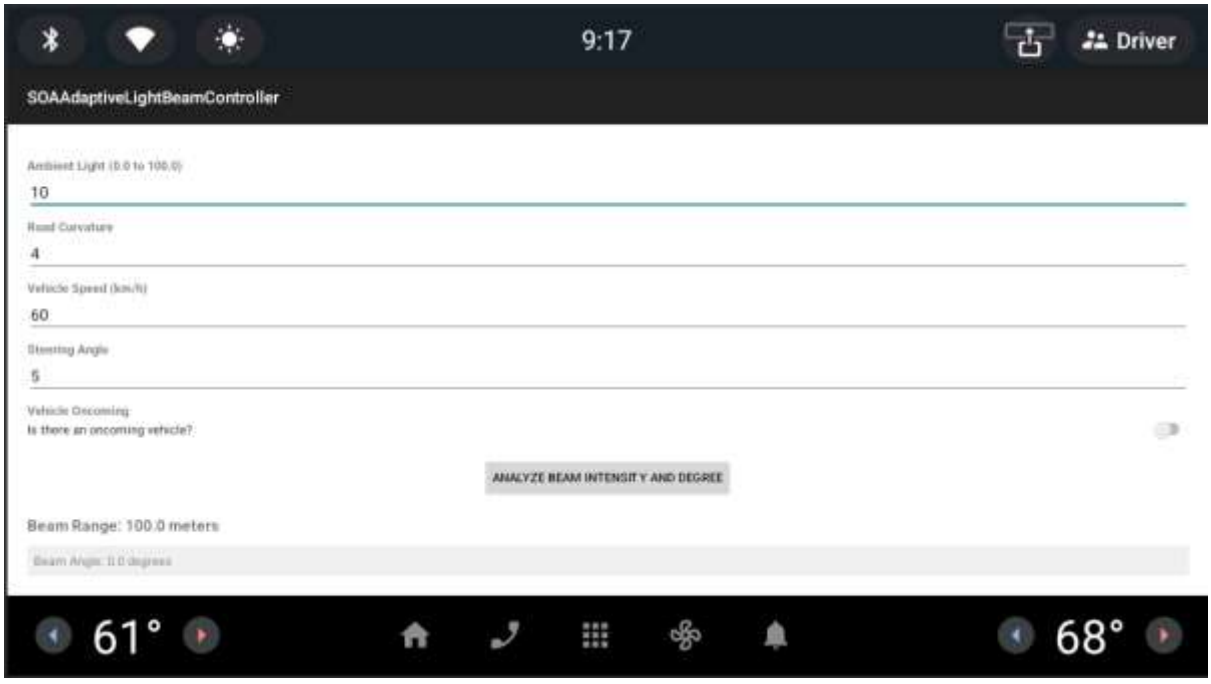


Figure 53: Result of Test Case 2.2

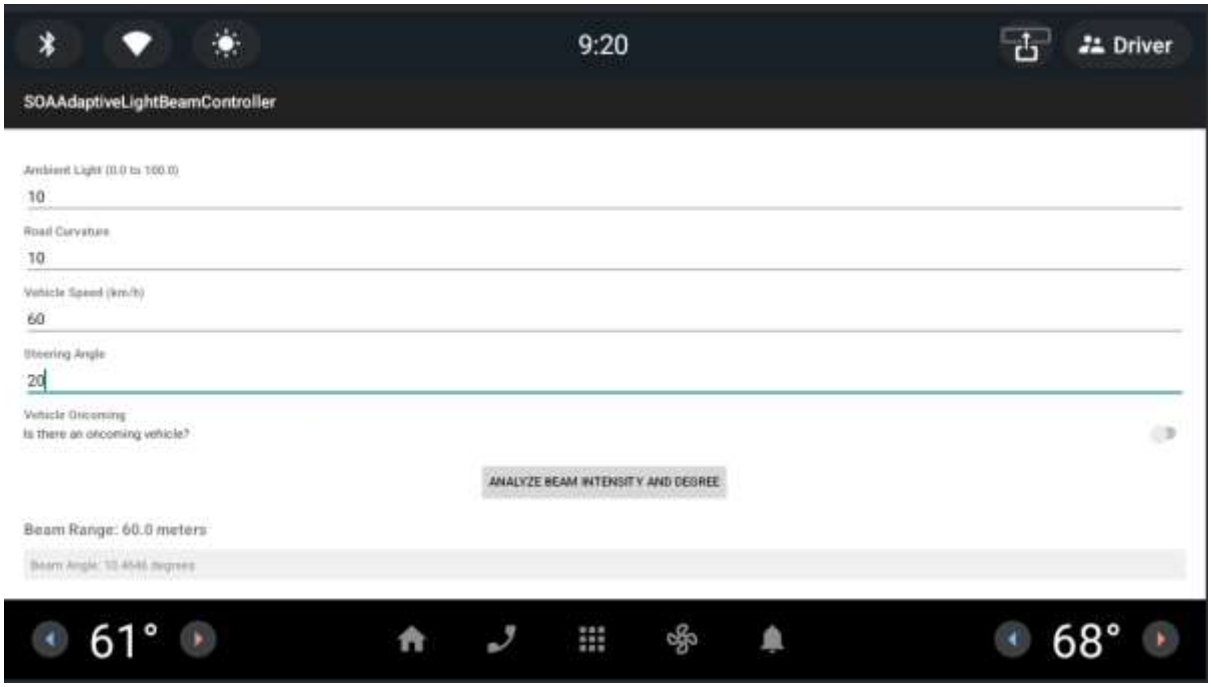


Figure 54: Result of Test Case 2.3

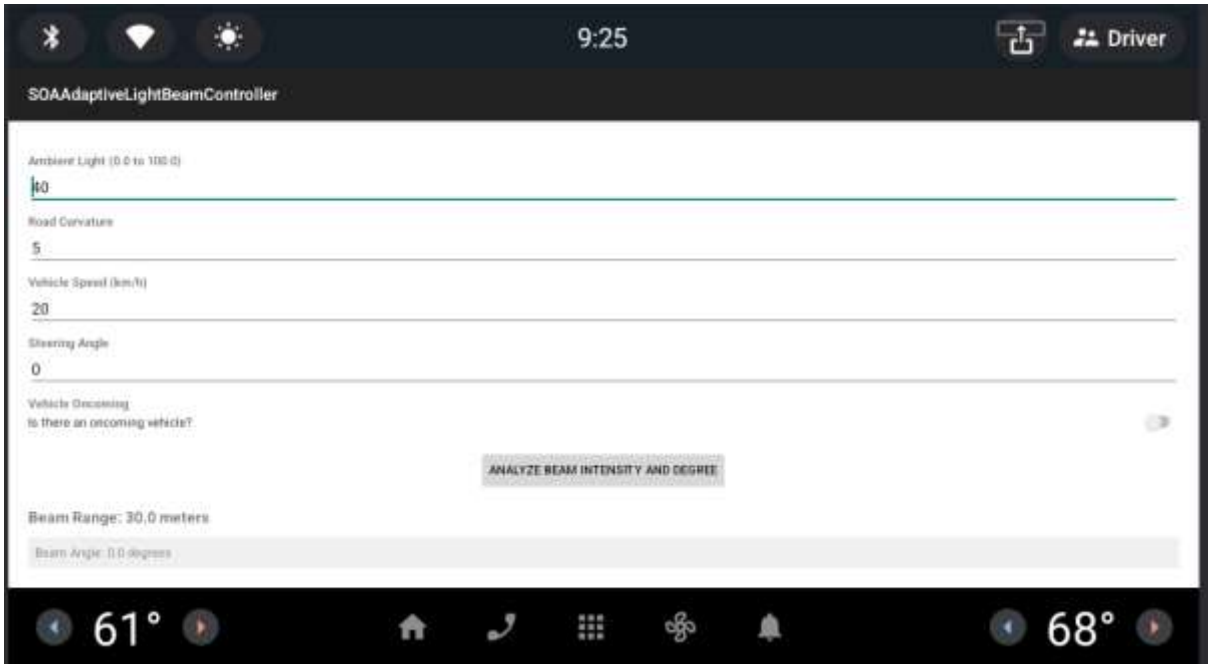


Figure 55: Result of Test Case 3.1

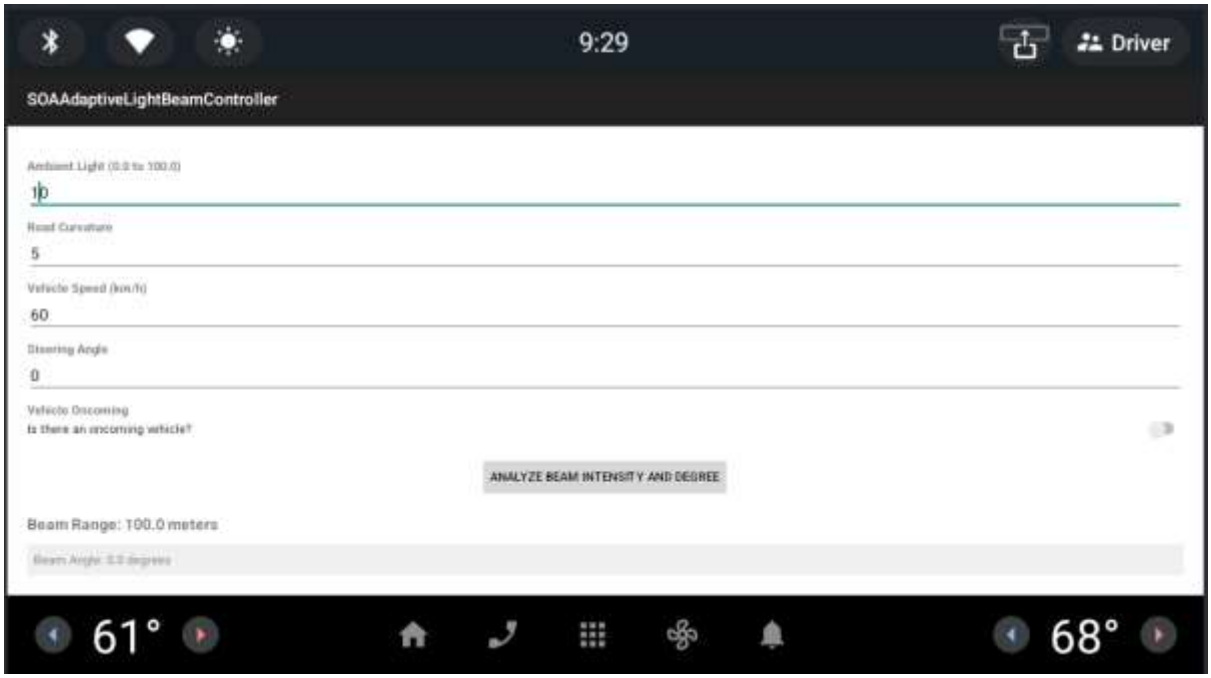


Figure 56: Result of Test Case 3.2

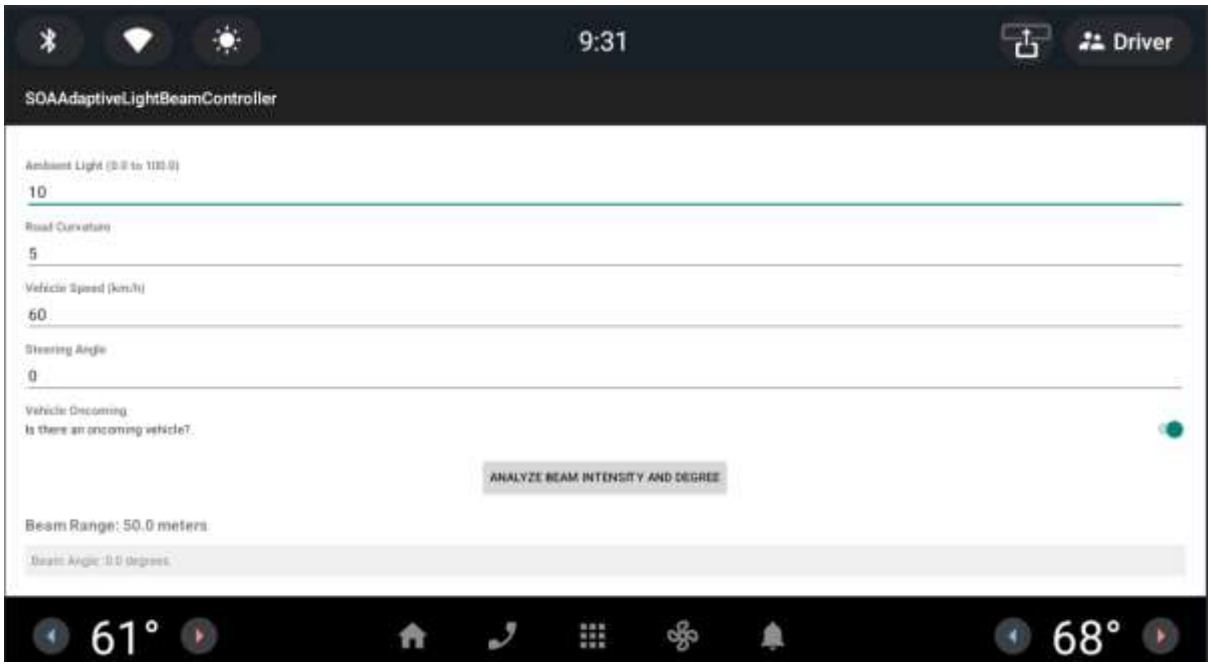


Figure 57: Result of Test Case 4.1

Chapter 6

6. How to shift the Android Studio project to AOSP

6.1 Shifting of MonolithicAdaptiveLightController

Integration Methods

There are two main ways to integrate an app with Android Automotive:

- Adding the Complete Source Code: Embed your app's code directly into the AOSP build system. [17]
- Adding the Compiled APK File: Build your app separately, create an APK, and integrate it into the AOSP folder. [17]

Method 1: Integrating Source Code (Detailed Steps)

6.1.1. Create a Sample Android Project:

- See chapter 3 and 5. In which I created the application named "MonolithicAdaptiveLightController" and integrated into the Android Studio.

6.1.2. Prepare a Folder in AOSP:

- In the AOSP source code, navigate to */packages/apps/Car*. [17]
- Duplicate an existing app folder (e.g calendar app) and rename it to "MonolithicAdaptiveLightController".
- Delete all files and folders except *src, res* and *Android.bp*. Delete the existing contents inside src and res folders and keep it empty. [17]

6.1.3. Copy App Code and Resources:

- Copy the source code from the android studio folder automotive/src/main/cpp to MonolithicAdaptiveLightController/src/main/cpp folder. Remove the CMakeLists.txt file from the cpp, c++ source code will be built using Android.bp file. [17]
- Copy the java file from the android studio folder automotive/src/main/java to MonolithicAdaptiveLightController/src/main/java folder. [17]
- Do the same for the *res* folder shift it from automotive/src/main/res to MonolithicAdaptiveLightController /res. [17]

- Finally, copy *AndroidManifest.xml* to the AOSP project's MonolithicAdaptiveLightController folder.

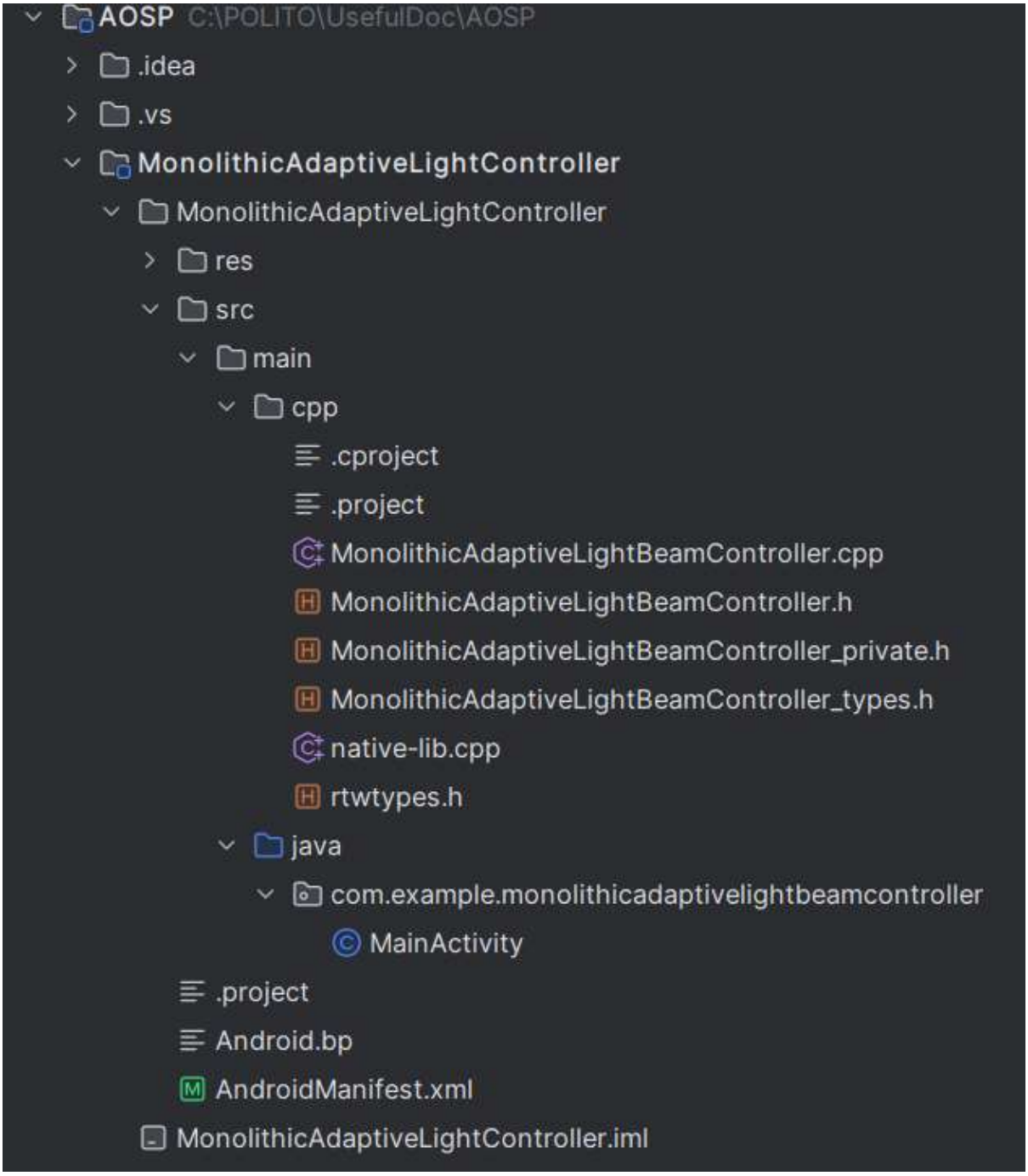


Figure 58: AOSP MonolithicAdaptiveLight Controller folder structure

6.1.4. Modify Android.bp:

The Android Studio project uses gradle as build system and the build configurations are defined in the build.gradle file. The AOSP project uses the Soong build system and the build configurations are defined using blueprint file(.bp). [17]

- Open Android.bp in the AOSP project's MonolithicAdaptiveLightController folder.

- Update the project name, remove unnecessary dependencies, and add required ones from your *build.gradle* file, ensuring syntax compatibility.
- In order to build the C++ source code use cc_library_shared inside the android.bp file it tells Soong to build a shared library .so from C++ sources and then using shared_libs inside android_app you can link your native library to the android app APK.

Here is an example of *Android.bp* file

```
cc_library_shared {
    name: "monolithicadaptivelightbeamcontroller",
    srcs: [
        "src/main/cpp/native-lib.cpp",
        "src/main/cpp/MonolithicAdaptiveLightBeamController.cpp",
    ],
    shared_libs: [
        "android",
        "log",
    ],
    stl: "libc++",
    cppflags: ["-std=c++17"],
    cflags: ["-std=c++17"],
    export_include_dirs: [
        "src/main/cpp",
    ],
    sdk_version: "current",
}

android_app {
    name: " monolithicadaptivelightbeamcontroller ",

    srcs: [
        "src/main/java//*.java",
    ],

    manifest: "AndroidManifest.xml",

    static_libs: [
        "androidx_appcompat",
        "androidx_core",
    ],

    jni_libs: [
        " monolithicadaptivelightbeamcontroller ",
    ],

    sdk_version: "current",

    aaptflags: ["--auto-add-overlay"],
}
```

6.1.5 Replace UI-based input signal acquisition with retrieval from the Vehicle HAL, as is done in real-world applications

Here's a **fully integrated summary** of all required changes, combining the project-level notes with the MainActivity.java, activity_main.xml, and AndroidManifest.xml changes: [17]

6.1.5.1 Change summary

Here's a concise **before vs after** comparison:

Component	UI-based Project (Before)	HAL-based Project (Now)
UI (activity_main.xml)	Input fields (EditText, Switch, Button) for manual user input	Only shows **results** & **logs**. All inputs come from **HAL/mocks**
MainActivity.java	Reads values from UI, validates them, passes to stepModel()	Subscribes to **Vehicle HAL** via CarPropertyManager, mocks missing signals (roadCurvature, vehicleOncoming), sends values automatically to stepModel()
JNI Bridge (C++ file)	Method Data passed from UI	Unchanged. Still gets 5 parameters, now from HAL
AndroidManifest.xml	Standard Android app permissions	Needs **Automotive HAL permissions** (e.g., android.permission.CAR_SPEED, CAR_STEERING)
Gradle Config	Standard Android app settings	Ensure minSdk ≥ 29, targetSdk matches Automotive; no major changes
Testing	User types input manually	Signals auto-updated from HAL; non-standard signals mocked (e.g., roadCurvature)

6.1.5.2 MainActivity.java (HAL-based)

```
package com.example.monolithicadaptivelightbeamcontroller;

import android.car.Car;
import android.car.hardware.CarPropertyValue;
import android.car.hardware.property.CarPropertyManager;
import android.car.hardware.property.VehiclePropertyIds;
import android.os.Bundle;
import android.util.Log;
```

```
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

import org.json.JSONException;
import org.json.JSONObject;

public class MainActivity extends AppCompatActivity {
    static {
        System.loadLibrary("monolithicadaptivelightbeamcontroller");
    }

    // Native methods
    public native void initModel();
    public native String stepModel(double ambientLight, double roadCurvature,
                                   double steeringAngle, boolean vehicleOncoming,
                                   double vehicleSpeed);
    public native void terminateModel();

    // Vehicle-related
    private Car car;
    private CarPropertyManager carPropertyManager;

    // Latest signal values
    private double vehicleSpeed = 0.0;
    private double steeringAngle = 0.0;
    private double ambientLight = 50.0;
    private boolean vehicleOncoming = false; // MOCKED
    private double roadCurvature = 0.0;      // MOCKED (degrees)

    private TextView intensityOutput;
    private TextView logOutput;

    private volatile boolean isRunning = true;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        initModel();

        intensityOutput = findViewById(R.id.intensityOutput);
        logOutput = findViewById(R.id.BeamAngleOutput);

        // Connect to CarService
        try {
            car = Car.createCar(this);
            carPropertyManager                =                (CarPropertyManager)
car.getCarManager(Car.PROPERTY_SERVICE);

            // Subscribe to standard HAL signals
            registerCarSignal(VehiclePropertyIds.PERF_VEHICLE_SPEED);
            registerCarSignal(VehiclePropertyIds.STEERING_ANGLE);
            registerCarSignal(VehiclePropertyIds.AMBIENT_LIGHT_LEVEL);

        } catch (Exception e) {
            Log.e("MainActivity", "CarService init failed: " + e.getMessage());
        }
```

```
        //  🔹 Mock road curvature (in degrees) & vehicle oncoming
        new Thread(() -> {
            int counter = 0;
            while (isRunning) {
                // Fake curvature between -60° and +60°
                roadCurvature = -60.0 + Math.random() * 120.0;

                // Toggle oncoming vehicle every 5 updates (~10 sec)
                if (counter % 5 == 0) {
                    vehicleOncoming = !vehicleOncoming;
                }
                counter++;

                runModelAndUpdateUI();

                try {
                    Thread.sleep(2000); // update every 2s
                } catch (InterruptedException ignored) {}
            }
        }).start();
    }

    private void registerCarSignal(int propertyId) {
        carPropertyManager.registerCallback(callback, propertyId,
                CarPropertyManager.SENSOR_RATE_ONCHANGE);
    }

    private final CarPropertyManager.CarPropertyEventCallback callback =
            new CarPropertyManager.CarPropertyEventCallback() {
        @Override
        public void onChangeEvent(CarPropertyValue value) {
            try {
                switch (value.getPropertyId()) {
                    case VehiclePropertyIds.PERF_VEHICLE_SPEED:
                        vehicleSpeed = ((Float) value.getValue()).doubleValue();
                        break;
                    case VehiclePropertyIds.STEERING_ANGLE:
                        steeringAngle = ((Float) value.getValue()).doubleValue();
                        break;
                    case VehiclePropertyIds.AMBIENT_LIGHT_LEVEL:
                        ambientLight = ((Float) value.getValue()).doubleValue();
                        break;
                }
                runModelAndUpdateUI();
            } catch (Exception e) {
                Log.e("MainActivity", "Signal parse error: " + e.getMessage());
            }
        }

        @Override
        public void onErrorEvent(int propId, int zone) {
            Log.e("MainActivity", "CarProperty error for " + propId);
        }
    };

    private void runModelAndUpdateUI() {
        try {
```

```
            String resultJson = stepModel(
                    ambientLight,
                    roadCurvature,
                    steeringAngle,
                    vehicleOncoming,
                    vehicleSpeed
            );

            JSONObject output = new JSONObject(resultJson);
            double beamRange = output.getDouble("beamRange");
            double beamAngle = output.getDouble("beamAngle");

            runOnUiThread(() -> {
                intensityOutput.setText(String.format("Beam Range: %.1fm | Angle:
%.1f°",
                        beamRange, beamAngle));

                logOutput.append(String.format(
                        "\nSpeed: %.1f km/h | Ambient: %.1f lx | Curve: %.1f° |
Angle: %.1f° | Oncoming: %b",
                        vehicleSpeed, ambientLight, roadCurvature, steeringAngle,
vehicleOncoming));
            });

        } catch (JSONException e) {
            Log.e("MainActivity", "Model output parse error: " + e.getMessage());
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        isRunning = false;
        try {
            if (carPropertyManager != null) {
                carPropertyManager.unregisterCallback(callback);
            }
            if (car != null) car.disconnect();
        } catch (Exception ignored) {}
        terminateModel();
    }
}
```

Explanation

6.1.5.2.1. Native Model Integration

- initModel() initializes the C++ model.
- stepModel(...) runs the adaptive light beam algorithm with five inputs:
    - ambientLight, roadCurvature, steeringAngle, vehicleOncoming, vehicleSpeed.
- terminateModel() shuts down the model safely.

6.1.5.2.2. Vehicle HAL Integration

- Car and CarPropertyManager connect to the vehicle's HAL.
- Subscribed signals:

    - PERF_VEHICLE_SPEED
    - STEERING_ANGLE
    - AMBIENT_LIGHT_LEVEL
- Callback (CarPropertyEventCallback) automatically updates local variables and triggers the model whenever a property changes.

6.1.5.2.3. Mocked Signals

- roadCurvature:
    - Randomly varies between **−60° to +60°** to simulate curved roads.
- vehicleOncoming:
    - Boolean toggled every 5 updates (~10 seconds) to simulate oncoming traffic.

6.1.5.2.4. Periodic Updates

- A background thread updates **mocked signals** every 2 seconds.
- Calls runModelAndUpdateUI() to combine HAL and mocked signals for processing.

6.1.5.2.5. UI Updates

- intensityOutput shows **beam range and angle**.
- logOutput appends detailed logs: speed, ambient light, road curvature, steering angle, and oncoming vehicle status.
- runOnUiThread() ensures UI updates are thread-safe.

6.1.5.2.6. Lifecycle Management

- onDestroy() stops the background thread (isRunning = false), unregisters HAL callbacks, disconnects from CarService, and terminates the model.
- Ensures clean shutdown and prevents resource leaks.

Summary

This implementation **automates input signal acquisition** using the Vehicle HAL, while **mocking missing signals**. It continuously feeds all signals into the native model, producing **live outputs** in the UI, making it a realistic simulation of **adaptive light beam control in real automotive systems**.

6.1.5.3 activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="24dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:orientation="vertical"
        android:gravity="center_horizontal">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Adaptive Light Beam Controller"
            android:textSize="24sp"
            android:textStyle="bold"
            android:layout_marginBottom="24dp" />

        <!-- Output for final computed values -->
        <TextView
            android:id="@+id/intensityOutput"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Beam Range: -- m | Beam Angle: -- °"
            android:textSize="18sp"
            android:textStyle="bold"
            android:layout_marginTop="20dp" />

        <!-- Log/debug area -->
        <TextView
            android:id="@+id/BeamAngleOutput"
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:text="Logs will appear here..."
            android:background="#f0f0f0"
            android:padding="8dp"
            android:layout_marginTop="12dp"
            android:scrollbars="vertical" />

    </LinearLayout>
</ScrollView>
```

Explanation:

- Removed all **manual input widgets**.
- Added **TextViews** for live beam output (intensityOutput) and logs (BeamAngleOutput).
- UI now reflects **automatic updates from HAL signals**

6.1.5.4 AndroidManifest.xml (HAL Permissions)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.monolithicadaptivelightbeamcontroller">

    <uses-permission android:name="android.permission.CAR_SPEED"/>
    <uses-permission android:name="android.permission.CAR_STEERING"/>
    <uses-permission android:name="android.permission.CAR_AMBIENT_LIGHT"/>

    <application
        android:allowBackup="true"
        android:label="Adaptive Light Beam Controller"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
```

```
        android:supportsRtl="true"
        android:theme="@style/Theme.AppCompat.Light.NoActionBar">

        <activity android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

    </application>

</manifest>
```

Explanation:

- Added permissions for accessing **automotive vehicle signals** like speed, steering, and ambient light.
- Keeps standard activity declaration unchanged.

This gives a **complete view of all changes**: project-level, UI, activity code, manifest, and testing considerations.

6.1.6. Add the Project to the Build:

- Navigate to the appropriate file based on your Android version. [17]
- Android 14: *packages/services/Car/car_product/build/car_system.mk.* [17]
- Android13 or Android 12: *packages/services/Car/car_product/build/car.mk.* [17]
- Add your new app to the list. [17]

```
PRODUCT_PACKAGES                                +=                                  \
    CarFrameworkPackageStubs                                                        \
    CarService                                                                      \
    CarShell                                                                        \
    CarDialerApp                                                                    \
    CarRadioApp                                                                     \
    OverviewApp                                                                     \
    CarLauncher                                                                     \
    CarSystemUI                                                                     \
    LocalMediaPlayer                                                                \
    CarMediaApp                                                                     \
    CarMessengerApp                                                                 \
    CarHTMLViewer                                                                   \
    CarHvacApp                                                                      \
    CarMapsPlaceholder                                                              \
    CarLatinIME                                                                     \
    CarSettings                                                                     \
    CarUsbHandler                                                                   \
    android.car                                                                     \
    car-frameworks-service                                                          \
    com.android.car.procfsinspector                                                 \
    libcar-framework-service-jni                                                    \
    ScriptExecutor                                                                  \
    MonolithicAdaptiveLightBeamController \
```

6.1.7. Build AOSP and Run the Emulator:

- Source the environment using . build/envsetup.sh . [17]
- Choose your target using lunch sdk_car_x86_64-userdebug.
- Build the source code with make -j$(nproc).
- Launch the emulator with emulator & to see your integrated app.

6.2. Shifting of SOAAdaptiveLightController to AOSP

6.2.1 Create a Sample Android Project

Before integrating the system into AOSP, a prototype of the SOA-based Adaptive Light Controller was implemented in Android Studio. (Refer to Chapters 3 and 5 for the implementation of the prototype.)

6.2.2 Prepare an application Folder in AOSP

To integrate the application within the AOSP environment:

6.2.2.1 Navigate to:

/packages/apps/Car

6.6.2.2 Duplicate any existing application folder (e.g., Calendar) and rename it to:

SOAAdaptiveLightController

6.6.2.3 Clean the folder by removing unnecessary subfolders. Keep:

src/
res/
Android.bp

6.6.2.4 Inside src :

Inside src folder place all Java, JNI (C++) source files from the original Android Studio project. [17]

6.6.2.5 Create an aidl directory

Create AIDL files for the AIDL interface files corresponding to each car service. (see figure 59)

6.6.2.6 Add AndroidManifest.xml

Add the application's AndroidManifest.xml to define permissions, package, and activities. [17]

The final file structure of the application:

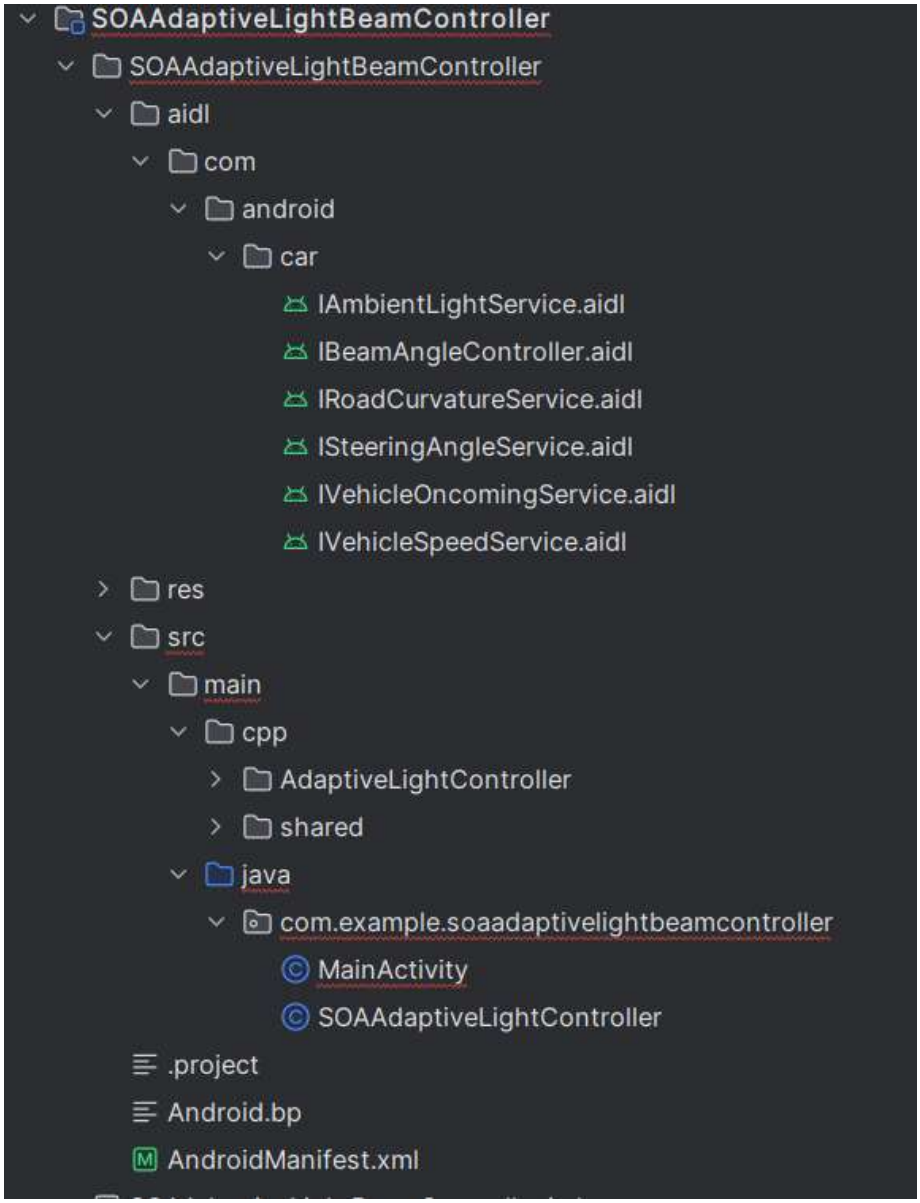


Figure 59: SOAAdaptiveLightController application file structure

6.2.3 Prepare folder structure associated with service implementation

6.2.3.1 Navigate to:

packages/services/car/service/

6.2.3.2 Create folders for each service:

ambientlight/
beamanglecontroller/
roadcurvature/
vehicleoncoming/
vehiclespeed/

6.2.3.3 For each service:

- Place the Simulink-generated .cpp and .h files.

- Add a corresponding Binder wrapper file (e.g., AmbientLightServiceBinder.cpp & AmbientLightServiceBinder.h).
- Create a minimal Android.bp file to compile each service as a static/shared library.

6.2.3.4 Create an aidl folder for service-side AIDL interfaces:

services/car/aidl/

6.2.3.5 Create a shared folder for reusable Simulink or utility code.

services/shared

6.2.3.6 Implement a CarServiceMain.cpp file which registers all services to CarService at startup.

*6.2.3.7 Finally, create a **top-level Android.bp** to include and link all service libraries.*

Final structure of the service folder in AOSP looks like:

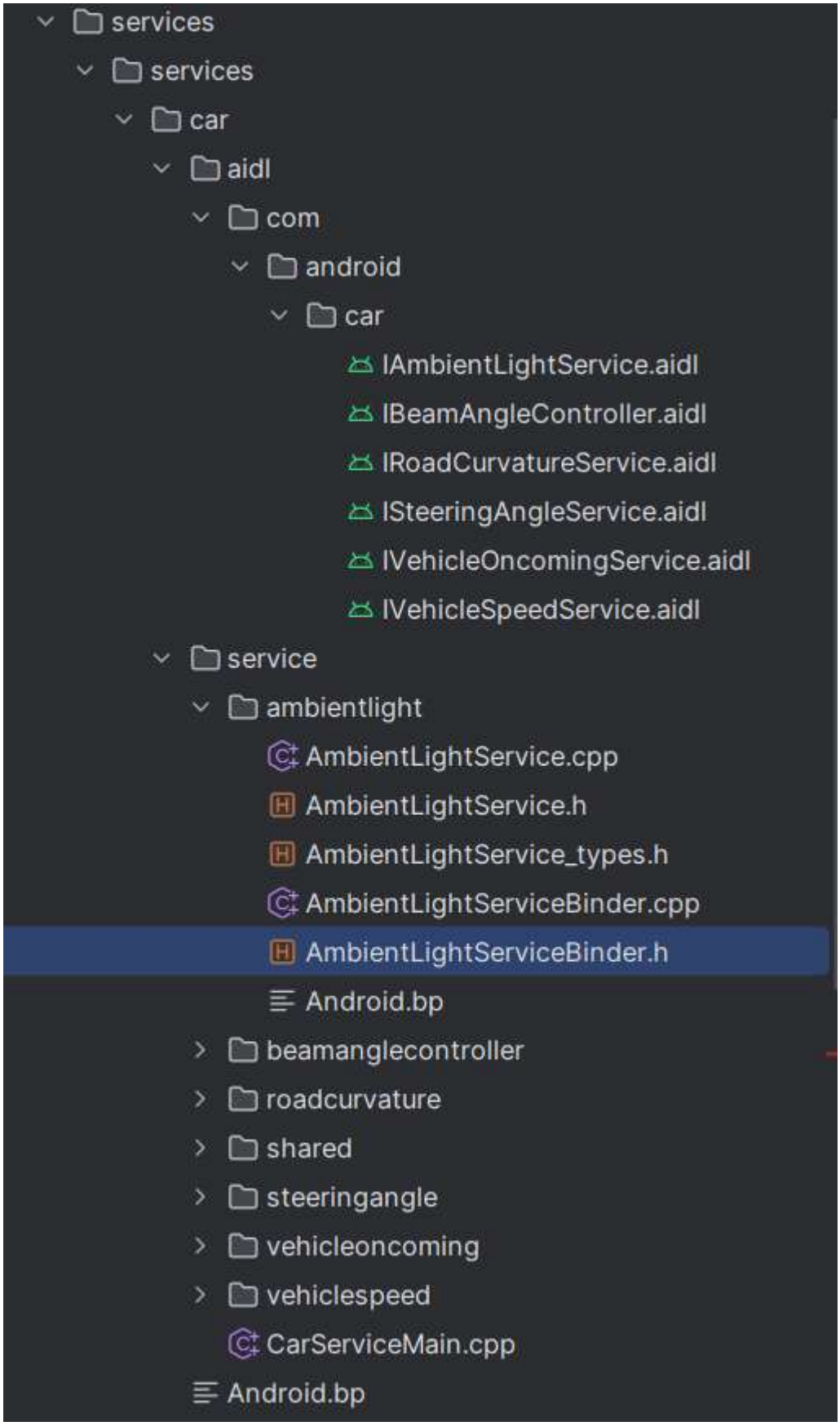


Figure 60: File structure of the service implementation in AOSP

6.2.4 Explanation of Important Files and Their Roles

6.2.4.1 AIDL Files

Each AIDL file defines a Binder interface through which the Java or C++ client (the app) communicates with the native services.

```
Example:
package com.android.car;

interface IAmbientLightService {
    double computeAmbientLight(double inputAmbientLight);
}
```

When compiled, AOSP generates Binder stub and proxy classes (using NDK AIDL), enabling inter-process communication between the app (client) and car service (server).

6.2.4.2 Service Implementation (e.g., AmbientLightService.cpp)

This file contains the **core computation logic** (auto-generated from Simulink). For instance, AmbientLightService::computeAmbientLight() computes or simulates an environmental light response value.

6.2.4.3 Binder Wrapper (AmbientLightServiceBinder.h / .cpp)

These files act as the **bridge** between Android's Binder system and the native C++ service logic.

Header:

```
#pragma once
#include <aidl/com/android/car/BnAmbientLightService.h>
#include "AmbientLightService.h"

// Binder wrapper around AmbientLightService
class               AmbientLightServiceBinder              :               public
aidl::com::android::car::BnAmbientLightService {
public:
    AmbientLightServiceBinder();
    ndk::ScopedAStatus     computeAmbientLight(double     ambientLight,     double*
_aidl_return) override;

private:
    AmbientLightService mService;  // Simulink-generated implementation
};
```

Implementation:

```
#include "AmbientLightServiceBinder.h"
```

```
AmbientLightServiceBinder::AmbientLightServiceBinder() : mService() {}

ndk::ScopedAStatus AmbientLightServiceBinder::computeAmbientLight(
        double ambientLight, double* _aidl_return) {
    double result = 0.0;
    mService.computeAmbientLight(ambientLight, &result);
    *_aidl_return = result;
    return ndk::ScopedAStatus::ok();
}
```

Explanation:

- The class inherits from the **NDK Binder Stub** (BnAmbientLightService).
- It overrides the AIDL-defined method (computeAmbientLight).
- Internally, it calls the real algorithm implemented in AmbientLightService.cpp.
- It returns the result to the caller through the Binder IPC channel.

6.2.4.4 Per-Service Build File (Android.bp)

Each service has its own Android.bp file that defines how it is compiled.

```
cc_library {
    name: "ambientlight",
    srcs: [
        "AmbientLightService.cpp",
        "AmbientLightServiceBinder.cpp",
    ],
    cflags: ["-std=c++17"],
    include_dirs: [".", "../../shared"],
    stl: "c++_shared",
    visibility: ["//visibility:public"],
    ndk: { enabled: true },
}
```

Explanation:

- Builds the service as a **shared C++ library**.
- Includes headers from both its folder and the shared Simulink headers.
- Makes the service publicly visible to other AOSP modules.
- Enables use in the **NDK environment**, which allows Binder IPC with Java apps.

6.2.4.5 CarServiceMain.cpp (Service Registration)

This is the **entry point** of the Car Service process. It registers all service binders into Android's **Service Manager**, making them discoverable by clients.

```
#include <android/binder_manager.h>
#include <android/binder_process.h>
#include <android/log.h>
#include "ambientlight/AmbientLightServiceBinder.h"
#include "vehiclespeed/VehicleSpeedServiceBinder.h"
```

```
#include "roadcurvature/RoadCurvatureServiceBinder.h"
#include "steeringangle/SteeringAngleServiceBinder.h"
#include "vehicleoncoming/VehicleOncomingServiceBinder.h"
#include "beamangle/BeamAngleControllerBinder.h"

int main() {
    ABinderProcess_setThreadPoolMaxThreadCount(0);
    ALOGI("CarServiceMain starting...");

    auto                            ambientLightService                            =
ndk::SharedRefBase::make<AmbientLightServiceBinder>();
    AServiceManager_addService(ambientLightService->asBinder().get(),
                               "com.android.car.IAmbientLightService/default");

    // ... repeat for all other services

    ALOGI("All services registered successfully.");
    ABinderProcess_joinThreadPool();
    return 0;
}
```

Explanation of Flow:

1. The **Binder thread pool** is initialized to handle IPC requests.
2. Each service Binder object (e.g., AmbientLightServiceBinder) is instantiated.
3. Each service is registered with a **unique name** in the Android Service Manager (e.g., "com.android.car.IAmbientLightService/default").
4. The process joins the Binder thread pool to continuously listen for incoming IPC calls from applications.
5. When the app calls the corresponding AIDL interface, the request is routed to this process and handled by the corresponding Binder wrapper.

6.2.4.6 Master Build Script (Android.bp)

At the controller/ or car/ level, a main Android.bp file builds the executable carservice that includes all service modules.

```
cc_binary {
    name: "carservice",
    srcs: ["CarServiceMain.cpp"],
    cflags: ["-std=c++17"],
    stl: "c++_shared",
    shared_libs: [
        "ambientlight",
        "vehiclespeed",
        "beamanglecontroller",
        "roadcurvature",
        "steeringangle",
        "vehicleoncoming",
    ],
    visibility: ["//visibility:public"],
    ndk: { enabled: true },
}
```

Explanation:

- cc_binary defines a **native binary** executable called carservice.
- Links all per-service libraries into one running process.
- Registers all services automatically during boot.

6.2.5 Workflow Summary

1. **App side** calls the service method via AIDL (e.g., computeAmbientLight()).
2. **Binder IPC** transmits the request to the corresponding native service process (carservice).
3. The **Binder wrapper** (AmbientLightServiceBinder) receives the IPC call.
4. It delegates computation to the **Simulink-generated C++ logic** (AmbientLightService.cpp).
5. The computed result is returned through Binder IPC to the application.
6. The **UI updates** accordingly to display the processed data.

6.2.6 Conclusion

Integrating the **SOAAdaptiveLightController** directly into AOSP transforms it from a standalone Android app into a **system-level automotive service framework**. This design leverages the AOSP Binder IPC mechanism to achieve modularity, fault isolation, and scalability — all key characteristics of a **Service-Oriented Architecture (SOA)** within embedded automotive platforms.

Chapter 7

7. OTA Implementation for SOAAdaptiveLightController

7.1 Introduction

The **Over-The-Air (OTA)** update mechanism is an essential feature in modern automotive software architectures. It allows the vehicle's software components to be updated remotely, improving maintainability, safety, and feature expansion over time.[1][4][11][15]

In the context of the **SOAAdaptiveLightController** developed in this research, the OTA mechanism leverages the **Service-Oriented Architecture (SOA)** design to enable **individual services**, such as AmbientLightService, VehicleSpeedService, or BeamAngleController, to be **independently updated** without requiring a complete system rebuild or reflashing of the Android image.

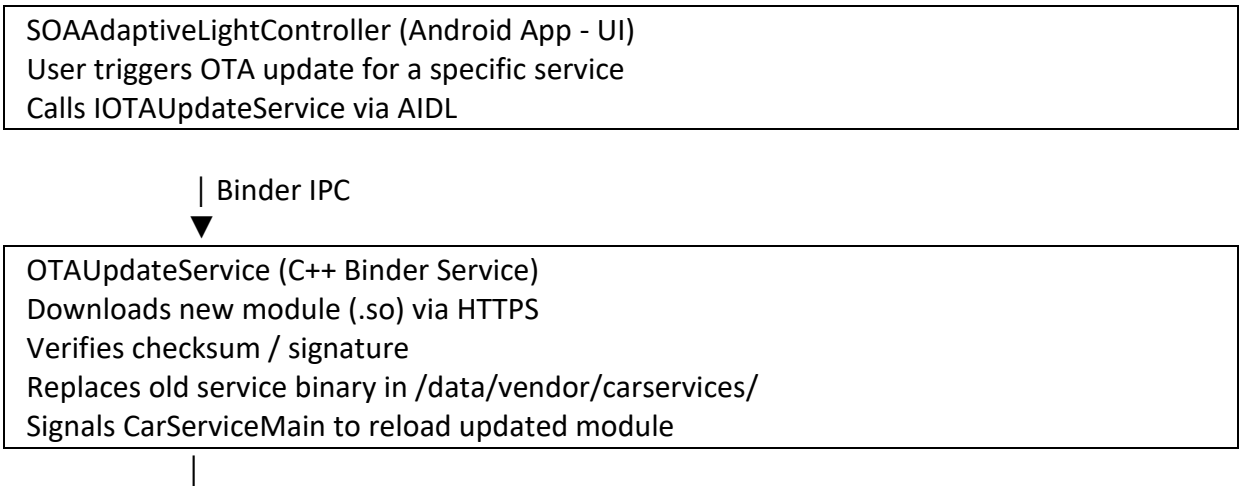
This approach aligns with the SOA principle of *"loose coupling and dynamic service management"*, making the system modular, upgradeable, and more resilient.

7.2 Architecture Overview

In the implemented AOSP structure, each vehicle-related functionality—ambient light detection, road curvature, vehicle speed, etc.—is developed as an independent Binder-based service module.
These services are dynamically registered through the main orchestrator (CarServiceMain.cpp), and their compiled outputs (.so shared objects) are linked within the car service binary.[2][3][12]

The OTA update mechanism introduces a new Binder service called **OTAUpdateService**, which handles downloading, verifying, and updating these service modules.[8]

The following diagram illustrates the OTA workflow within the AOSP-based SOA architecture:[8][2][3][12]



▼

CarServiceMain.cpp
Dynamically loads service libraries via dlopen()
Registers each updated service to AServiceManager
Provides runtime modularity and dynamic service loading

7.3 Implementation Steps

Step 1 – OTAUpdateService AIDL Definition

The IOTAUpdateService.aidl interface is defined to expose OTA update functionalities to the client layer. [8][2][3][12]

```
package com.android.car;

interface IOTAUpdateService {
    boolean downloadAndUpdateModule(String moduleName, String downloadUrl);
    String getUpdateStatus(String moduleName);
}
```

This interface allows the Android application to trigger OTA updates for specific modules (e.g., “ambientlight”) and to query their update status.

Step 2 – C++ Implementation of OTAUpdateService

The OTA service is implemented as a Binder-based C++ class that downloads and replaces service modules: [8][2][3][12]

```
#include "IOTAUpdateService.h"
#include <curl/curl.h>
#include <android/binder_manager.h>
#include <android/log.h>

#define MODULE_PATH "/data/vendor/carservices/modules/"
#define LOG_TAG "OTAUpdateService"

bool downloadFile(const std::string& url, const std::string& outPath);

class OTAUpdateService : public aidl::com::android::car::BnOTAUpdateService {
public:
    ndk::ScopedAStatus downloadAndUpdateModule(
            const std::string& moduleName,
            const std::string& downloadUrl,
            bool* _aidl_return) override {

        std::string tempPath = std::string(MODULE_PATH) + moduleName + "_new.so";
        std::string finalPath = std::string(MODULE_PATH) + moduleName + ".so";

        if (!downloadFile(downloadUrl, tempPath)) {
            ALOGE("Download failed for %s", moduleName.c_str());
            *_aidl_return = false;
            return ndk::ScopedAStatus::ok();
```

```
        }

        // Replace old module
        rename(tempPath.c_str(), finalPath.c_str());

        // Restart carservice to reload updated module
        system("stop carservice && start carservice");

        ALOGI("Updated module %s successfully", moduleName.c_str());
        *_aidl_return = true;
        return ndk::ScopedAStatus::ok();
    }
};
```

The service downloads a new .so binary, replaces the existing one, and triggers a lightweight restart of the carservice process, which then reloads all services including the updated one.

Step 3 – Dynamic Service Loading in CarServiceMain.cpp

To support OTA-updatable modules, CarServiceMain.cpp is modified to **dynamically load** service libraries using dlopen() and dlsym() instead of static linking. [8][2][3][12]

```
#include <dlfcn.h>
#include <android/binder_manager.h>

void loadAndRegisterService(const std::string& libPath, const std::string&
serviceName) {
    void* handle = dlopen(libPath.c_str(), RTLD_NOW);
    if (!handle) {
        ALOGE("Failed to load %s: %s", serviceName.c_str(), dlerror());
        return;
    }

    using CreateBinderFn = ndk::SpAIBinder (*)();
    auto createFn = (CreateBinderFn)dlsym(handle, "createBinder");
    if (!createFn) {
        ALOGE("Symbol not found in %s", serviceName.c_str());
        return;
    }

    auto binder = createFn();
    AServiceManager_addService(binder->asBinder().get(), serviceName.c_str());
}
Each service (such as AmbientLightService) exports a createBinder() method that
returns a Binder instance of the service.
extern "C" ndk::SpAIBinder createBinder() {
    return ndk::SharedRefBase::make<AmbientLightServiceBinder>()->asBinder();
}
```

This mechanism allows the carservice process to load a **newly updated** module without requiring recompilation or flashing.

7.4 OTA Update Example: AmbientLightService

To demonstrate the OTA functionality, the AmbientLightService was selected as a use case.

Initial Setup:

- Original library: /data/vendor/carservices/modules/ambientlight.so
- Registered Binder service: com.android.car.IAmbientLightService/default

Update Scenario:

Suppose an improvement is made in the Simulink model of the ambient light computation (for example, enhanced filtering of sensor noise). A new version of the service is built and published on a remote server as ambientlight_v2.so. [8][2][3][12]

Update Procedure:

1. **Trigger Update from App:**
    - The user (or system) calls:
    - otaService.downloadAndUpdateModule("ambientlight", "https://server.com/updates/ambientlight_v2.so");
2. **Download & Replacement:**
    - OTAUpdateService downloads the new binary and stores it as /data/vendor/carservices/modules/ambientlight_new.so.
    - It verifies integrity and replaces the old module:
    - mv ambientlight_new.so ambientlight.so
3. **Reload Service:**
    - The carservice process is restarted or signalled to reload.
    - CarServiceMain dynamically loads the new ambientlight.so.
    - The updated AmbientLightService is re-registered with the same AIDL interface.
4. **Result:**
    - Clients (like the Adaptive Light Controller app) continue to use IAmbientLightService seamlessly.
    - The updated logic is now active **without reflashing the firmware** or rebuilding the AOSP image.

7.5 Advantages of the OTA Mechanism

Feature	Description
Service-Level Modularity	Each vehicle function can be updated independently.
Reduced Downtime	No need for full system reflashing or reboots.
Improved Maintainability	Easier integration of new features or bug fixes.
Security	Update packages can be signed and verified.
Demonstrates SOA Principles	Dynamic service binding and independent deployment.

7.6 Summary

The integration of OTA update functionality into the **SOAAdaptiveLightController** project transforms it into a **truly service-oriented automotive software platform**. By enabling independent updates for services such as AmbientLightService, the system achieves:

- runtime flexibility,
- reduced maintenance overhead, and
- enhanced scalability.

This approach demonstrates how modern automotive systems can adopt cloud-driven service delivery models while remaining compliant with the modularity principles defined by SOA.

Chapter 8

8. Mixed Criticality in Android Automotive Systems

8.1 Introduction

Modern vehicles integrate a wide range of software functionalities with differing reliability, timing, and safety requirements. This coexistence of components with diverse assurance levels defines a **mixed-criticality system**, a core characteristic of today's automotive software architecture.

While traditional Electronic Control Units (ECUs) were designed for single-purpose, safety-critical tasks, recent technological trends—such as domain controllers and centralized computing—allow both **safety-critical** and **non-critical** software to share the same hardware resources. Managing these workloads safely and predictably is one of the central challenges of **Android Automotive OS (AAOS)** integration in modern vehicles. [11][13][14][15]

This chapter explores how mixed criticality is addressed in Android Automotive systems, the mechanisms used to achieve *freedom from interference*, and how the developed **Service-Oriented AdaptiveLight Controller (SOA-ALBC)** fits within this context as a representative infotainment-level application designed using **Model-Based Software Design (MBSD)** and integrated through Android Studio.

8.2 Concept of Mixed Criticality

8.2.1 Definition

Mixed criticality refers to the coexistence of software components with different levels of functional safety, real-time behaviour, and assurance requirements on a shared platform. According to **ISO 26262**, each function in a vehicle can be assigned an **Automotive Safety Integrity Level (ASIL)**, ranging from **ASIL D** (highest safety requirement) to **QM** (quality-managed, non-safety).[11]

Mixed criticality arises when both ASIL-classified and QM components execute on the same processor or within the same system. This situation demands careful system partitioning and resource management to prevent interference between components of different criticalities.

8.2.2 Challenges in Mixed-Criticality Integration

The coexistence of multiple criticality levels introduces several challenges:[11][13][14]

- **Freedom from Interference (FFI):** Lower-criticality software (e.g., Android applications) must not affect the timing, memory, or data integrity of higher-criticality components.
- **Timing Predictability:** Real-time control systems require deterministic response, which general-purpose systems like Android do not natively guarantee.

- **Resource Contention:** Shared use of CPUs, GPUs, and memory can lead to unpredictable performance if not controlled.
- **Security and Safety Assurance:** Non-safety domains must not compromise the operation or integrity of safety-critical subsystems.
- **Certification Complexity:** Demonstrating that the overall system meets safety standards becomes more demanding when criticality levels are mixed.

8.3 Mixed Criticality in Android Automotive Architecture

8.3.1 Overview of Android Automotive OS

Android Automotive OS (AAOS) is a Google-supported, embedded variant of the Android operating system specifically adapted for in-vehicle infotainment (IVI) systems. It manages tasks such as media playback, navigation, user interaction, and integration with vehicle sensors via the **Vehicle Hardware Abstraction Layer (VHAL)**.[2][3]

From a safety perspective, Android Automotive is considered a **non-safety (QM)** environment. However, it often operates in close proximity to safety-critical ECUs, forming part of a larger, mixed-criticality vehicle architecture.

8.3.2 Architectural Separation of Domains

To manage mixed criticality, automotive platforms employ **domain separation** through hardware and software partitioning. There is a conceptual separation between:

- **Safety/Real-Time Domain** — Executes safety-critical control functions (e.g., braking, steering, lighting logic) under a safety-certified Real-Time Operating System (RTOS) such as QNX, PikeOS, or AUTOSAR Classic. [11][13][14]
- **Infotainment Domain** — Hosts Android Automotive and its applications, including non-critical user-interface services such as the **SOA AdaptiveLight Controller** developed in this work.

Although Android Automotive applications may visualize or influence parameters related to safety systems, they are **not responsible for the real-time actuation or control logic**. Instead, Android acts as a **supervisory or monitoring layer** interfacing through standardized APIs.

8.3.3 Mechanisms Supporting Mixed Criticality

Android Automotive achieves coexistence with safety-critical systems through several key mechanisms:

1. **Hardware Virtualization and Hypervisors:** Modern SoCs (e.g., Qualcomm Snapdragon Ride, Renesas R-Car, NXP S32G) use hypervisors to host multiple isolated guest operating systems. Safety functions execute in a certified RTOS partition, while Android runs in a separate virtualized domain. This enforces *spatial and temporal isolation*. [13][14][15]

2. **Safety Islands and Secure Execution Environments:** Dedicated hardware cores, often called “safety islands,” execute essential safety software independently from Android. Even if Android crashes, these cores maintain the system in a safe state. [14][15]
3. **Linux Kernel Isolation (Cgroups, Cpuset, and Namespaces):** Within Android, process isolation and resource control are achieved using Linux kernel features such as control groups (**cgroups**) and **cpusets**, which prevent resource starvation and enforce CPU scheduling boundaries. [12]
4. **SELinux and Permission Enforcement:** Android Automotive enforces **Security-Enhanced Linux (SELinux)** in *enforcing mode*, defining policies that tightly restrict access to devices, I/O, and vehicle interfaces. Applications can only access vehicle data through the **Car Service** and **Vehicle HAL** layers, not through direct hardware interfaces. [2][3][12]
5. **Vehicle HAL Mediation:** The **Vehicle HAL** acts as a secure communication bridge between Android and underlying ECUs. Safety-critical ECUs expose limited, read-only or verified interfaces, ensuring that infotainment apps cannot issue unsafe commands. [2][3][12]
6. **Safe Communication Channels:** When cross-domain communication is necessary, it is implemented via authenticated, rate-limited IPC mechanisms or hypervisor-mediated shared memory channels to avoid overloading safety partitions.[13][14]

8.4 Case Study: SOA AdaptiveLight Controller Application

8.4.1 Model-Based Design Workflow

The **SOA AdaptiveLight Controller (SOA-ALC)** application developed in this thesis demonstrates the use of **Model-Based Software Design (MBSD)** for automotive use cases within Android Automotive.

The control logic was first designed in **MATLAB/Simulink**, modelled as a service-oriented component. Using **Embedded Coder**, the model was automatically converted into C++ source code, ensuring consistency between design and implementation. The generated code was then integrated into **Android Studio** as part of an AAOS application module.

8.4.2 Role within Mixed Criticality Architecture

Although adaptive lighting behaviour in production vehicles is safety-critical, in this thesis the SOA-ALBC serves as a **non-critical infotainment demonstration**. It visualizes and simulates adaptive lighting responses rather than directly controlling hardware.

Thus, the application resides entirely within the **Android (QM)** domain, utilizing AAOS APIs to demonstrate vehicle service interaction without influencing real-time actuation. This approach makes it an ideal **research use case** to analyse how complex control algorithms can be safely hosted in a mixed-criticality environment.[1][5][16]

8.4.3 Interaction Boundaries

The integration ensures clear separation between:

- **Application Layer:** Android application hosting user interface and service logic.
- **System Services:** Car Service and VHAL layers mediating any communication with vehicle subsystems.
- **External ECUs (if connected):** Access limited to simulated or read-only channels.

This architecture maintains *freedom from interference* by ensuring that any misbehaviour in the SOA-ALC app (e.g., CPU spikes, software errors) does not affect the functioning of safety ECUs or other domains.[2][11][12]

8.5 Safety, Security, and System Assurance Considerations

8.5.1 Freedom from Interference (FFI)

AAOS ensures FFI between infotainment applications and safety-critical components through:

- Process sandboxing and SELinux policy enforcement.[2][12]
- Controlled access via Car Service and Vehicle HAL.[2][3]
- Resource quotas using Linux kernel control groups.
- Strict application signing and permission management.

8.5.2 Compliance Context

While AAOS itself is not ISO 26262-certified, it is designed to **coexist** with certified safety platforms. In such setups, the Android environment operates at the QM level, and its functions are excluded from the vehicle's safety case.[11]

The SOA-ALBC application, implemented in this work, thus aligns with the **non-safety (QM)** classification, illustrating a safe and structured integration of complex MBSD-generated software in the Android domain.

8.5.3 Security and Update Management

Android Automotive employs secure boot, verified updates, and application signing to ensure system integrity. These mechanisms are critical in mixed-criticality systems, as untrusted updates in non-critical domains must not jeopardize the safe operation of the overall vehicle architecture.[12]

8.6 Discussion

The study highlights that Android Automotive provides a flexible and robust platform for developing advanced, service-oriented automotive applications while maintaining strict boundaries between safety and non-safety domains.

From the perspective of system design:

- **Safety-critical control** should reside outside Android, in RTOS or safety partitions. [11][13][14]
- **Infotainment and visualization functions**, such as the SOA-ALC, can be implemented safely within Android, benefiting from its development ecosystem, connectivity, and user interface capabilities.
- Proper use of the VHAL, SELinux, and virtualization ensures compliance with mixed-criticality principles. [2][12]

This separation allows research and development teams to explore advanced automotive functionalities (like adaptive lighting logic) at the application level without endangering safety or violating functional safety constraints.

8.7 Conclusion

Mixed-criticality management in Android Automotive is achieved through a combination of architectural isolation, hardware support, and robust software mechanisms. Android, as a non-safety environment, complements safety-certified domains by enabling high-level, user-facing applications and services.

The **SOA Adaptive Light Beam Controller** developed in this thesis exemplifies how a model-based, service-oriented application can be deployed within AAOS, respecting mixed-criticality boundaries and demonstrating safe integration of advanced functionalities in the infotainment layer.

This approach reinforces the potential of combining **Model-Based Software Design** with **Android Automotive** to accelerate innovation, maintain software quality, and preserve system integrity in future vehicle architectures.

Chapter 9

9. Vehicle Signal Integration in Android Automotive

9.1 Overview

In a vehicle equipped with Android Automotive OS (AAOS), input signals from physical sensors and actuators—such as steering angle, speed, ambient light, or headlamp status—are typically transmitted over a **Controller Area Network (CAN) bus**. However, Android does **not directly access the CAN bus** for both **safety** and **architectural** reasons. [2][3][12]

Instead, signal acquisition and distribution occur through a **layered integration architecture**, where the **Vehicle HAL (Hardware Abstraction Layer)** acts as the interface between Android and the vehicle's underlying ECUs or middleware.[2]

This section describes how CAN signals flow through the system, the role of intermediate components, and how Android applications like the **SOA Adaptive Light Beam Controller** can consume those signals safely.[11]

9.2 Signal Flow: From CAN Bus to Android

The typical signal path can be described in the following stages:

```
[Vehicle Sensors/Actuators]
       ↓
[CAN Bus]
       ↓
[Vehicle Gateway ECU / Vehicle Interface Processor]
       ↓
[Vehicle HAL]
       ↓
[Car Service / Car API]
       ↓
[Android Application Layer (e.g., SOA Adaptive Light Beam Controller)]
```

9.2.1 Vehicle Sensors and CAN Bus

At the hardware level, vehicle sensors and actuators communicate through one or more CAN networks (e.g., Powertrain CAN, Body CAN, Chassis CAN). Each signal is encoded as a **CAN frame** identified by a unique CAN ID, and contains payload data such as:

- Vehicle speed
- Steering angle
- Ambient light intensity
- Headlamp status

These messages are broadcast periodically, typically every 10–100 ms, depending on their criticality.[12]

9.2.2 Vehicle Gateway ECU

The **Vehicle Gateway ECU** (or Body Control Module, or dedicated middleware gateway) acts as a **bridge** between the CAN bus and higher-level systems such as Android Automotive. Its main responsibilities include:

- Receiving and decoding CAN messages.
- Converting raw CAN data into **abstracted vehicle signals** (e.g., VehicleSpeed = 45 km/h).
- Filtering, scaling, or rate-limiting the signals.
- Providing these processed signals to higher-level software components through standard interfaces, often via Ethernet or shared memory.

This gateway may run:

- A **real-time OS (RTOS)** or **AUTOSAR stack** for safety and timing.
- A communication middleware such as **Some/IP**, **DDS**, or **gRPC over IPC** for higher-level communication.

It isolates Android from direct bus access, enforcing the **freedom from interference** principle.[2][3][12]

9.3 Vehicle HAL (Hardware Abstraction Layer)

9.3.1 Purpose

The **Vehicle HAL (VHAL)** is the Android Automotive system component that provides a standard interface for vehicle-related data to the Android framework. It abstracts away the details of how vehicle signals are obtained, presenting a unified API to Android services and applications.

VHAL defines a set of **Vehicle Properties**, each identified by an integer constant (e.g., VEHICLE_PROPERTY_SPEED, VEHICLE_PROPERTY_STEERING_ANGLE). These are standardized in the Android Open Source Project (AOSP).[2][3]

9.3.2 Structure

VHAL is implemented in **C++** and runs in native space (under /vendor partition). It typically communicates with the vehicle gateway or middleware via:

- **Socket-based IPC**
- **Binder interface**
- **Shared memory**
- **Some/IP or gRPC interface**

It translates external data into standardized property structures (VehiclePropValue), which are published to the Android **Car Service**.

Example (simplified flow):

```
VehiclePropValue value;
value.prop = VEHICLE_PROPERTY_SPEED;
value.value.floatValues[0] = decodedSpeed;
mVehicleHal->setProperty(value);
```

This data is then pushed to the **Car Service** in the Android framework layer.[2][3][12]

9.4 Car Service and Car API

9.4.1 Car Service

The **Car Service** runs in the Android System Server process. It communicates with the Vehicle HAL via Binder IPC and provides higher-level access to vehicle data through a managed API layer.

It defines the permission model and ensures that only authorized components can read or modify certain vehicle properties.
For example:

- Speed or fuel level may be read by any system UI component.
- Door lock status or ignition control is restricted to system-only components.

9.4.2 Car API (Application Layer)

Applications in the Android domain use the **Car API**, part of the **android.car** package, to read or subscribe to vehicle signals.

Example Java/Kotlin usage:

```
val car = Car.createCar(context)
val     carPropertyManager     =     car.getCarManager(Car.PROPERTY_SERVICE)     as
CarPropertyManager

carPropertyManager.registerCallback(object                                          :
CarPropertyManager.CarPropertyEventCallback {
    override fun onChangeEvent(value: CarPropertyValue<*>) {
        if (value.propertyId == VehiclePropertyIds.PERF_VEHICLE_SPEED) {
            val speed = value.value as Float
            // Use vehicle speed as input for adaptive lighting visualization
        }
    }
    override fun onErrorEvent(propId: Int, zone: Int) {}
}, VehiclePropertyIds.PERF_VEHICLE_SPEED, CarPropertyManager.SENSOR_RATE_ONCHANGE)
```

This abstraction ensures that the app never directly interacts with CAN data; instead, it receives high-level vehicle properties published by the VHAL. [2][3][12]

9.5 Integration Example: SOA Adaptive Light Beam Controller

In the context of the thesis:

1. **Signal Origin:**
    - Real-world: Vehicle's ambient light sensor and steering angle sensor transmit data over the CAN bus.
    - Simulated case: The system emulates these signals (e.g., through test datasets or synthetic generators in Simulink).
2. **Gateway Translation:**
    - The vehicle gateway decodes CAN frames and exposes standardized properties such as:
        - VehiclePropertyIds.STEERING_ANGLE
        - VehiclePropertyIds.AMBIENT_LIGHT_LEVEL
3. **Vehicle HAL Integration:**
    - The HAL receives these signals through IPC (e.g., Some/IP or shared memory) and converts them into VehiclePropValue objects.
4. **Android Framework:**
    - The Car Service receives updates and broadcasts them through the CarPropertyManager interface.
5. **Application Consumption:**
    - The **SOA Adaptive Light Beam Controller** (Android app) subscribes to relevant properties and uses them to drive adaptive-lighting logic (simulated or visualized).

In a real automotive deployment, the adaptive lighting *actuation* would occur in a separate, safety-certified ECU (e.g., Body Controller), while Android handles **visualization, settings, or simulation** only.

9.6 Safety and Isolation Considerations

Even though vehicle signals from CAN reach Android, the **data path is strictly one-way** for most properties. Android applications are:

- **Read-only** for safety-critical properties.
- **Rate-limited** to prevent flooding the HAL or gateway.
- **Permission-restricted** using Android's automotive permission model (android.car.permission.CAR_SPEED, etc.).

This ensures **freedom from interference**, where failures in the Android domain cannot compromise the safety-critical vehicle control logic.[2][3][12]

9.7 Summary

Layer	Function	Example Technology
Sensors/ECUs	Generate raw signals	CAN, LIN, FlexRay

Vehicle Gateway	Decode & abstract data	AUTOSAR, RTOS, Some/IP
Vehicle HAL	Standardize vehicle properties	C++ HAL module
Car Service	Manage access & publish data	Android system server
Application	Consume vehicle data	SOA Adaptive Light Beam Controller

REFERENCES/BIBLIOGRAPHY

[1] Migrating Traditional Automotive Applications to SOA for Software-Defined Vehicles - MATLAB & Simulink

[2] Automotive | Android Open Source Project

[3] Android Automotive OS overview | Android for Cars | Android Developers

[4] software defined vehicles - Video e Webinar - MATLAB & Simulink

[5] SDV: Integrating Simulink C++ Generated Code in Android Automotive Environment - MATLAB & Simulink

[6] Run apps on the Android Emulator | Android Studio | Android Developers

[7] https://it.mathworks.com/videos/sdv-integrating-simulink-c-generated-code-in-android-automotive-environment-1691429159219.html

[8] OTA update in Android Automotive

[9] System Composer - MATLAB

[10] https://medium.com/androiddevelopers/getting-started-with-c-and-android-native-activities-2213b402ffff

[11] ISO 26262: Road Vehicles – Functional Safety.

[12] Android Open Source Project: *Vehicle System Isolation in Android Automotive*

[13] SYSGO: *Mixed-Criticality Partitioning in PikeOS*

[14] QNX: *QNX Hypervisor for Safety – Product Brief*

[15] Qualcomm Technologies: *Snapdragon Ride Flex – Mixed-Criticality SoC Architecture*

[16] MathWorks: *Embedded Coder for Automotive Applications*

[17] Android Automotive build your first app