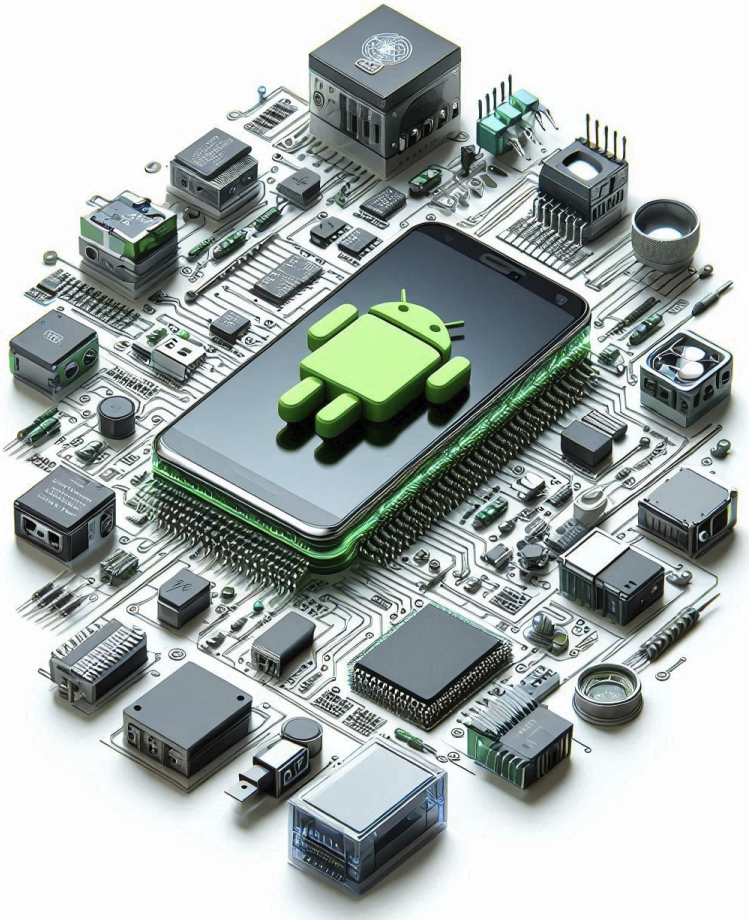
Information Technology Institute



Android Automotive

AOSP Customizations Training Track for ITI

Islam Samak

Software Architect

Contents

Overview

This course will take participants through a comprehensive journey of Android Open Source Project (AOSP) with a focus on Automotive Customization and hardware integration using Android OS Internals and Embedded Android concepts. This training program is designed for developers to understand the complete stack from operating system internals to build, customize, and extend Android Automotive by integrating 3rd-party devices, developing hardware abstraction layers (HAL), creating system services, and exposing APIs to the application layer. It includes detailed theoretical insights, practical labs, and a final project focused on automotive integration.

Objectives

- Master AOSP build systems and automotive architecture.
- Customize Android Automotive OS, integrate 3rd-party automotive devices.
- Master HAL and system service development for integrating 3rd-party hardware.
- Be able to expose system services to the Android application layer through APIs.
- Work through hands-on labs to build a complete automotive solution.
- Gain knowledge of debugging, testing, and certification processes for automotive projects.

Timeline

Week 1: Introduction to AOSP Automotive

1.1 Introduction to AOSP and Android Automotive

- Overview of AOSP and its significance in the automotive industry.
- Introduction to embedded systems and their relevance to Android Automotive.
- **Key Topics**:
    - AOSP Overview
    - Linux vs. Android Architecture
    - Automotive and Android – Integration

1.2 Setting Up the Environment

- **Practical Setup**:
    - Install required tools (Java, Python, Git, Repo, etc.).
    - Install and configure the REPO tool for source management.
    - Download and set up the AOSP source (e.g., Android 14 for Raspberry Pi 4).
    - **Lab**: Building and flashing Android Automotive ROM on Raspberry Pi 4.
- Build System Fundamentals:
    - Introduction to Soong and Make.
    - Overview of `envsetup.sh`, build commands (`m`, `mm`, `mmm`), module management.

1.3 Basic Android Build System and Commands

- **Key Topics**:
    - Understanding build architecture (Soong, Makefile, Ninja).
    - Running and debugging automotive images in the Android emulator.
    - Android Debugging Bridge (ADB) and Fastboot: basics for device communication.
    - **Lab**: Change Boot Animation.
    - **Lab**: Set a predefined app apk as System app.
    - **Lab**: Flash partitions using Fastboot and run ADB for diagnostics.

Week 2: Android Automotive Architecture and Framework

2.1 Android Automotive Architecture

- **Key Topics**:
    - Overview of Android Automotive architecture.
    - Introduction to important AOSP folders (e.g., system, device, vendor).
    - Android boot process deep dive: Init, Zygote, System Server.
    - **Lab**: Customizing init scripts for Raspberry Pi 4.

2.2 Car Service, Car API, and Car Framework

- Understanding Android's Car Service architecture.
- Car API deep dive and AIDL (Android Interface Definition Language).
- Customizing Car Service and integrating new vehicle features.
- **Lab**: Extend Car API to include a custom sensor (e.g., tire pressure, speedometer).

2.3 Exploring HAL for Automotive

- **Key Topics**:
    - Introduction to Vehicle HAL (VHAL).
    - HAL properties, areas, and zones.
    - Practical example: Custom VHAL integration with LSHAL.
    - **Lab**: Add custom VHAL properties for external sensors.

Week 3: System Services, HAL Development, and Customization

3.1 Customizing System Services

- Overview of Android System Services.
- Creating new system services and using Binder IPC.
- Exposing system services to the application layer.
- **Lab**: Build a custom system service for an external battery monitor.

3.2 HAL Integration

- Hardware Abstraction Layer (HAL) in detail.
- Developing and integrating HAL for new automotive devices.
- **Practical**: Adding support for 3rd-party devices in the HAL (e.g., speedometer, battery).
- **Lab**: HAL integration for a custom speedometer device.

3.3 Customizing Android Automotive Framework

- Modifying Android Automotive UI framework.
- **Key Topics**:
    - Customizing the car settings UI.
    - Overlaying car UI themes using RRO (Runtime Resource Overlay).
- **Lab**: Create a custom car launcher and modify the boot animation.

Week 4: Native Services and AIDL Development

4.1 Native C++ Services in Android

- Introduction to Native Android Services using C++.
- Developing native services that communicate with HAL.
- **Practical**: Creating native AIDL services to interface with custom hardware.
- **Lab**: Build a native C++ AIDL service for a 3rd-party telephony device.

4.2 AIDL and Inter-Process Communication

- Understanding AIDL and inter-process communication in Android.
- Creating and integrating AIDL interfaces to expose system APIs.
- **Lab**: Build AIDL interface to expose custom speedometer data to apps.

4.3 SELinux Customization and Security

- Introduction to SELinux in Android.
- Customize and create SELinux policies for secure services.
- **Lab**: Configure SELinux for newly created native services and system services.

Week 5: Application Layer Integration and Testing

5.1 Exposing APIs to Android Apps

- Introduction to the app layer and API integration.
- Exposing HAL and system service data to Android applications.
- **Lab**: Develop a simple Android app to read speedometer and battery data.

5.2 Building Custom Android Apps into AOSP

- Integrating custom applications into AOSP build.
- Adding Java/Kotlin apps to the AOSP framework.
- **Lab**: Build and integrate a telephony and battery monitor app into AOSP.

5.3 Debugging and Testing in Android Automotive

- ADB deep dive: advanced debugging and diagnostics.
- Running Google Compatibility Test Suite (CTS) on Android Automotive ROM.
- **Lab**: Running CTS tests and resolving common issues in automotive builds.

Week 6: Final Capstone Project

6.1 Final Project Overview

Participants will apply all knowledge acquired during the course to integrate and build a complete automotive system that interfaces with 3rd-party hardware devices.

6.2 Project: Full Automotive System Integration

- **Objective**: Build a fully integrated Android Automotive solution that communicates with external devices and exposes APIs to apps.
- **Requirements**:
    - Create a custom Vehicle HAL for 3rd-party sensors (e.g., telephony, speedometer).
    - Develop native services and system services for hardware devices.
    - Expose APIs via AIDL for use by Android applications.
    - Build and integrate custom Android applications (e.g., speedometer, telephony, battery monitor).

6.3 Final Project Presentation and Review

- Students will present their projects, demonstrating their custom ROM, services, and apps.
- Group review and feedback from instructors.

Week 1: Introduction to AOSP Automotive

1 Introduction to AOSP Automotive

1.1 Introduction to AOSP and Android Automotive

Overview of AOSP

Android, initially developed as an operating system for digital cameras, was acquired by Google in 2005 and repurposed as a smartphone operating system. Android builds upon the Linux kernel, but it diverges considerably from standard Linux distributions due to significant changes in both the kernel and user-space components, designed to cater specifically to mobile and embedded environments.

AOSP, or **Android Open Source Project**, is the base platform for Android devices. It includes the source code for the Android operating system and all associated tools, documentation, and APIs. AOSP's code is open and can be modified by anyone, providing developers with a flexible platform for building custom versions of Android.

AOSP and Its Significance in the Automotive Industry

The **Android Open Source Project (AOSP)** platform serves as the foundation for the development of custom Android systems for various smart devices, starting from **smartphones, tablets,** and **iot** to **automotive systems**.

Significance in Automotive Industry:

- **Customization**: AOSP allows manufacturers and developers to customize the Android operating system to meet the specific requirements of automotive environments. This includes integrating automotive-specific features, hardware, and services such as infotainment systems, car diagnostics, navigation, and telematics.
- **Cost Efficiency**: AOSP helps automotive OEMs (Original Equipment Manufacturers) reduce costs by leveraging a ready-made, highly scalable operating system rather than developing proprietary platforms from scratch.
- **Community and Ecosystem Support**: By using AOSP, automotive manufacturers can tap into the vast Android developer community, which provides continued innovation and support for new features, security updates, and patches.
- **Integration of Android Apps**: Automotive systems based on AOSP can utilize the massive ecosystem of Android applications, enabling rich user experiences that go beyond the traditional in-car experience, such as integrating entertainment, connectivity, and smartphone projection (like Android Auto).

- **Rapid Development Cycle**: AOSP provides a flexible platform for innovation, allowing automotive manufacturers to develop features and customizations rapidly, responding to market demands more quickly.

As automotive technology advances towards autonomous driving, the importance of seamless integration between hardware, software, and services is more critical than ever. AOSP provides the perfect foundation to ensure that vehicles can run state-of-the-art features while supporting new standards like **ADAS** (Advanced Driver-Assistance Systems) and **V2X** (Vehicle-to-Everything) communication.

Introduction to Embedded Systems and Their Relevance to Android Automotive

Relevance to Embedded Systems: Embedded systems are specialized computing systems that are designed to perform dedicated functions or control specific hardware. They are typically optimized for performance, reliability, and real-time execution, and can be found in a wide variety of devices including cars, medical devices, industrial machines, and more.

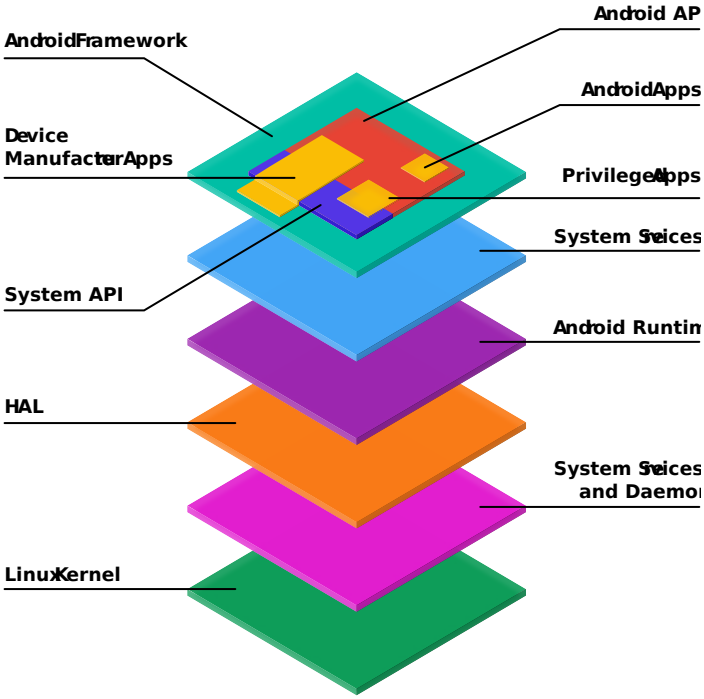
- **Real-time Processing**: Embedded systems often need to meet real-time constraints where actions must occur within a strict timeframe (e.g., a brake control system in an automotive environment).
- **Resource Constraints**: These systems typically run on hardware with limited resources (memory, processing power) compared to general-purpose systems like PCs or smartphones.
- **High Reliability and Safety**: Many embedded systems are used in safety-critical applications, such as automotive systems, where failure can lead to accidents or malfunctions.

Relevance to Android Automotive: Android Automotive is effectively a specialized embedded system, tailored specifically for in-car environments, with features like infotainment, navigation, diagnostics, and car settings management. The combination of Android's robust operating system and the nature of embedded systems makes it an ideal platform for automotive applications.

- **Automotive Control**: Android Automotive can interact with vehicle control systems via **Vehicle HAL (Hardware Abstraction Layer)**, which interfaces with hardware components such as sensors, cameras, and actuators.
- **Real-Time Requirements**: While Android Automotive focuses more on infotainment than mission-critical systems, it still needs to support certain real-time features, such as real-time data display from the vehicle's diagnostics.
- **Infotainment and Connectivity**: Embedded systems power in-car infotainment systems, providing smooth multimedia playback, internet connectivity, and integration with mobile devices via AOSP's standard services.
- **Custom Hardware Integration**: Android Automotive is highly modular, allowing developers to integrate custom hardware such as speedometers, GPS systems, third-party sensors, and telematics devices.

AOSP Architecture Overview

- **Android Framework**: Provides high-level APIs for application development. For automotive, this includes specialized APIs like the Car API.
- **System Services**: Core services that manage device functions like power management, telephony, or networking.
- **Hardware Abstraction Layer (HAL)**: A middle layer that bridges the Android framework with hardware drivers, allowing Android to interact with different devices like cameras, displays, sensors, etc.
- **Linux Kernel**: The core of Android, providing the low-level management of hardware resources, memory, and processes.

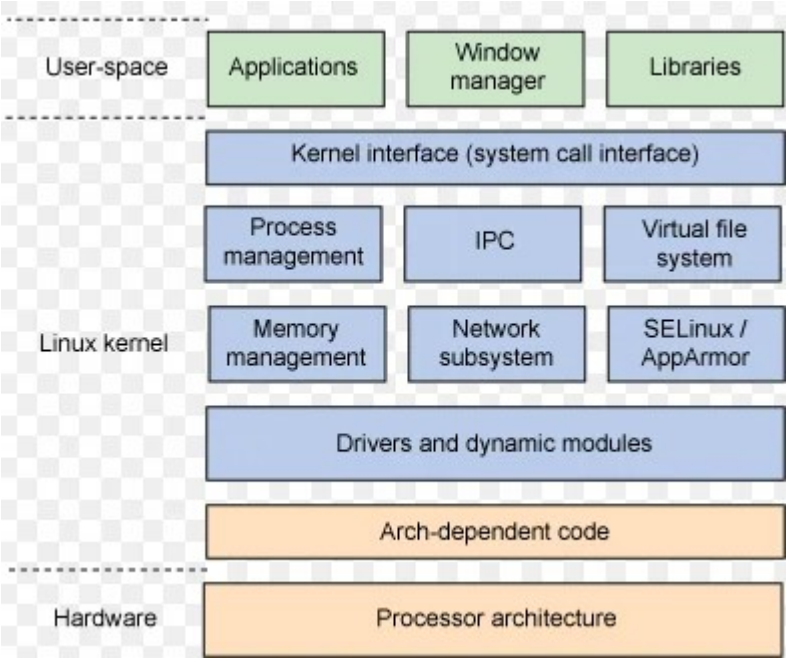


For the *automotive sector*, **Android Automotive** builds on top of AOSP by adding specific components and services such as **Vehicle HAL** and **Car Service**, designed to interface with in-car systems like navigation, audio, and diagnostics.

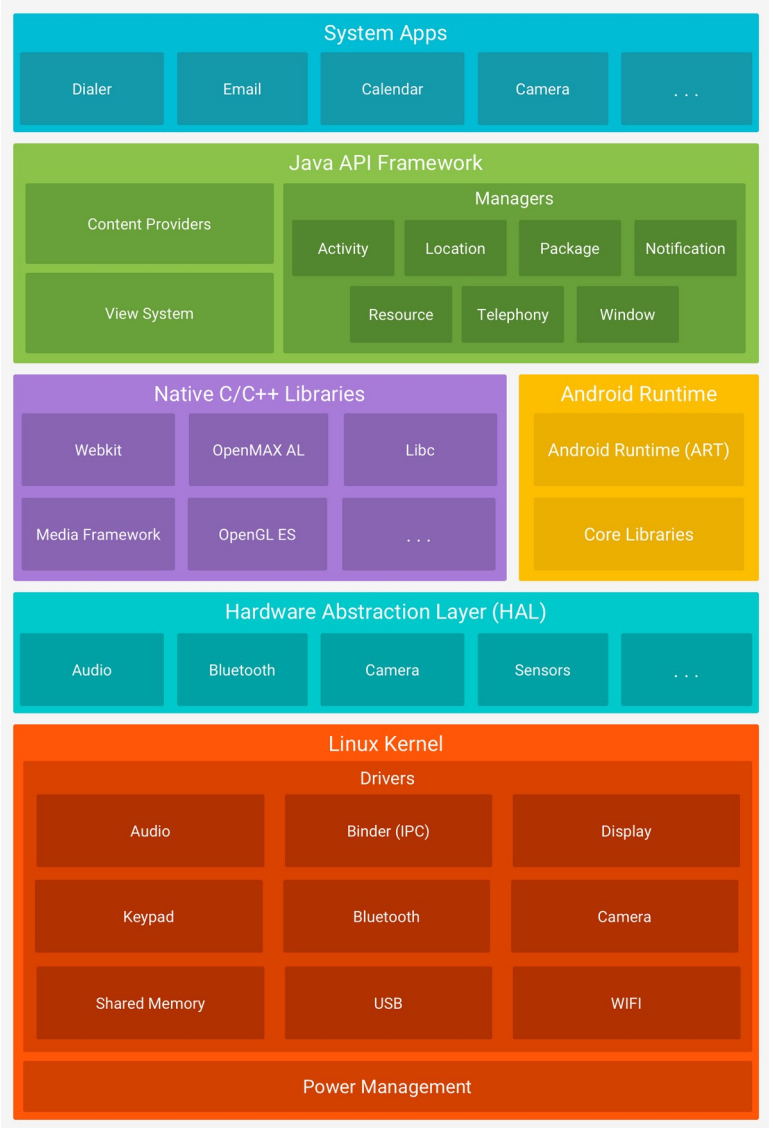
Android vs Linux Architecture Overview

Linux Architecture:

- **Shell & Command Line Interface**: Linux provides a terminal-based command-line interface for administrative control.
- **Userspace**: Contains the components and libraries required to run applications, such as system daemons, file systems, and system utilities.
- **Kernel**: The Linux kernel manages hardware resources (CPU, memory, I/O, etc.) and provides an interface for software applications to communicate with the hardware.



Android Architecture: While Android builds on top of the Linux kernel, it adds additional layers and abstractions tailored for mobile and embedded devices like automobiles.



- **Application Layer**: AOSP adds a higher-level Applications layer with different Applications types such as **Android app**, **Privileged App**, and **Device Manufacturer App**.
- **Application Framework**: Android adds a higher-level API framework that allows developers to create apps without needing to interact directly with hardware or system libraries.
- **Android Runtime (ART)**: Instead of the traditional Linux userspace, Android runs apps within the Android Runtime. ART replaces the Java Virtual Machine (JVM) and is optimized for low memory usage, crucial in embedded systems.
- **System Libraries**: Android has its own libraries (Bionic libc, WebKit, etc.) designed to provide the functionality required for apps, including multimedia processing, database management, and internet connectivity.
- **Hardware Abstraction Layer (HAL)**: A middle layer that bridges the Android framework with hardware drivers, allowing Android to interact with different devices (i.e. Cameras, Displays, Sensors, ... etc)
- **Linux Kernel**: Like standard Linux, the Android kernel manages hardware, but Android has its own set of drivers, optimized for battery life, touchscreen, wireless communication, etc.

Android vs Linux

- ***Kernel and System Architecture Differences***

Android uses a modified version of the Linux kernel. While it retains many core components, Android introduces numerous patches and unique subsystems:

- **Bionic C Library vs. Standard C Libraries:**
    - The Bionic C library in Android is distinct from the GNU C Library (glibc) or musl used in typical Linux distributions. Bionic is designed to be lightweight, optimized for the resource constraints of mobile devices, with faster performance and reduced memory footprint.
    - It also incorporates Android-specific functionalities, such as specialized system calls, which make it less compatible with applications written for glibc-based environments.
    - In contrast, standard Linux distributions typically allow users to choose among multiple C libraries (glibc, musl, uClibc, etc.) depending on the application requirements, but Android exclusively uses Bionic.
- **Process Scheduling:**
    - Android and Linux both utilize the Completely Fair Scheduler (CFS), but Android includes modifications to enhance real-time performance for multimedia applications. This involves using additional scheduling policies like the Real-Time (RT) scheduler for audio and video processing, ensuring low-latency response for time-sensitive tasks.
    - Starvation, where some processes may get less CPU time due to the fairness of CFS, can be mitigated in Android through priority-based scheduling mechanisms. Android distinguishes between foreground and background processes, giving higher priority to tasks that directly affect the user experience.

- ***Memory Management Enhancements***

Memory management is handled differently in Android compared to desktop Linux:

- **Low Memory Killer vs. Out-of-Memory (OOM) Killer:**
    - In earlier versions of Android, a Low Memory Killer (LMK) was used to terminate low-priority applications proactively when memory was low. This mechanism was integrated into the Android kernel and worked alongside the Dalvik or ART runtime to manage application states.
    - In recent Android versions, the LMK has been replaced by a newer Userspace Low Memory Killer (lmkd) daemon. The lmkd operates in user space and is more flexible, as it uses a kernel mechanism called "Memory Pressure Notification" to track memory usage and

respond accordingly.
- The traditional OOM Killer in Linux is used only as a last resort, while Android's memory management system is more proactive, terminating background processes based on priority and recent usage to ensure a smooth user experience.

- ***Power Management Differences***

Power management is one of the most significant areas of divergence between Android and Linux:

- **Suspend to RAM and Wake Locks:**
  - Android's "Suspend to RAM" feature goes beyond typical Linux power management by frequently suspending the device to conserve battery life. It is heavily utilized to ensure mobile devices remain efficient while idle.
  - Wake locks are a critical feature in Android, allowing applications to prevent the system from suspending when they perform critical tasks, such as downloading files or playing music. Wake locks are more than just a user-space concept; they integrate deeply with Android's power management system, coordinating kernel and application behavior to balance power use and responsiveness.

- ***Inter-Process Communication (IPC)***

Android's IPC mechanism is designed with a mobile-first approach:

- **Binder IPC vs. System V IPC:**
  - While traditional Linux provides System V IPC (message queues, shared memory) and POSIX shared memory for inter-process communication, Android relies on Binder IPC. The Binder framework is integral to Android's architecture and supports Remote Procedure Call (RPC)-style communication, enabling applications and services to interact seamlessly across different processes.
  - Binder is a lightweight, high-performance IPC mechanism that includes security features, such as permission checks, to control access to system services. Android's service manager uses Binder to facilitate system-wide services like telephony, location, and application lifecycle management.
  - Unlike standard Linux IPC, which requires manual handling of memory management and synchronization, Binder simplifies these processes, allowing developers to work with higher-level interfaces.

- ***Hardware Abstraction Layer (HAL)***

Android’s HAL is a distinct abstraction layer not commonly found in desktop Linux:

- **HAL's Role in Resource Management:**
  - Android's HAL provides a standard interface for interacting with hardware components, such as cameras, GPS, or audio devices, which may have different implementations across device manufacturers.
  - Unlike traditional Linux, where device drivers are accessed directly through `/dev` nodes or other kernel interfaces, Android's HAL allows the operating system to abstract the hardware details, ensuring that application developers work with a consistent interface regardless of the underlying hardware.
  - This is particularly important in the Android ecosystem, where device hardware varies widely across different manufacturers and models.

- ***Logging and Diagnostics***

Android introduces a robust logging system to aid in development and diagnostics:

- **Logcat vs. Linux Logging Mechanisms:**
  - In traditional Linux systems, logging can be managed using tools like `dmesg` (kernel logs), `syslog`, or `journalctl` for user-space logs. However, these tools are not suitable for the diverse and complex logging needs of mobile devices.
  - Android’s Logcat provides a unified logging system that captures messages from various sources, including the kernel, Android framework, system services, and applications. This allows developers to filter logs based on tag or priority level (e.g., DEBUG, ERROR, INFO), making it easier to trace issues.
  - Additionally, Logcat logs are integrated with Android's development tools, such as Android Studio, for enhanced debugging capabilities.

- ***Security Model***

Android's security model also sets it apart from typical Linux systems:

- **Application Sandboxing:**
  - Each Android application runs in its own Linux process with a unique user ID (UID), ensuring a sandboxed environment where applications cannot directly access each other's data.
  - Mandatory Access Control (MAC) frameworks like SELinux are enforced to restrict the behavior of applications, even if they exploit a vulnerability. This mandatory policy further enhances Android’s security compared to the more optional SELinux deployments in traditional Linux.

For **Android Automotive**, the **Vehicle HAL (VHAL)** is critical as it enables the Android system to communicate with various hardware components in the vehicle. It uses a combination of C++ and AIDL (Android Interface Definition Language) to create hardware-agnostic interfaces that can be implemented for different car hardware.

Conclusion

While Android is built on the Linux kernel, it significantly extends and customizes it to cater to the unique needs of mobile devices. These customizations include memory and power management optimizations, a distinct IPC system (Binder), a robust HAL, and advanced security measures, all of which help make Android more suited to embedded and mobile environments. Understanding these distinctions is crucial for developers working with Android, as it demonstrates how mobile OS requirements differ from traditional desktop or server systems.

Automotive and Android – Integration

The integration of Android into the automotive industry brings significant advantages in terms of flexibility, customization, and a wide ecosystem of apps and developers. Here are the main elements of integration:

1. **Vehicle HAL (VHAL)**:
    - Acts as the middleware between the vehicle's hardware and the Android Automotive framework.
    - VHAL abstracts various hardware components, such as speedometers, GPS systems, and climate control systems, allowing Android to interact with vehicle-specific hardware in a standardized way.
    - Developers can extend VHAL by defining new hardware interfaces and exposing vehicle properties (e.g., battery level, engine state) to the Android framework.
2. **Car Service**:
    - The Car Service runs in the **System Server** and acts as the central communication point between VHAL and Android Automotive features.
    - It manages vehicle-specific services like navigation, climate control, telephony, and audio.
3. **Car API**:
    - Provides high-level APIs for app developers to interact with automotive services, such as accessing vehicle speed, controlling media playback, or reading battery state.
    - App developers can use these APIs to create automotive-optimized apps that provide real-time interaction with vehicle data.
4. **Customization and Extensions**:
    - One of the key benefits of integrating Android into automotive systems is the ability to customize and extend both the operating system and its interface.
    - Manufacturers can customize the user interface, modify default settings, add proprietary apps and services, and adjust the system to interact with custom hardware.

With Android Automotive, the boundaries between in-car infotainment, driver assistance, and vehicle control systems are blurred. The flexibility of AOSP combined with the power of Android's app ecosystem allows manufacturers to build cars with seamless integration of hardware and software, providing better user experiences and robust customization options for the future of smart vehicles.

1.2 Setting Up the Environment for AOSP Development

Practical Setup

Setting up the environment for AOSP development is a critical first step to ensuring a smooth development experience. This includes installing the necessary tools, configuring the source management system, and downloading the Android Open Source Project (AOSP) source code for building Android Automotive ROMs. For this guide, we will focus on setting up **Android 14** for **Raspberry Pi 4** as the target device.

1. Install Required Tools

AOSP development requires several tools and libraries to be installed on your development machine. Here's a breakdown of the essential tools:

1. **Repo**:
    - Repo is a Google-built tool that manages multiple Git repositories for large projects like AOSP.
    - Install Repo:

      ```
      mkdir -p ~/bin
      curl https://storage.googleapis.com/git-repo-downloads/repo >
      ~/bin/repo
      chmod a+x ~/bin/repo
      ```

    - Add Repo to your `PATH` by editing your `.rc` or `.zshrc`:

      ```
      export PATH=~/bin:$PATH
      ```

2. **Additional Tools**:
    - Install required packages for building AOSP (this list may vary depending on your host operating system):

      ```
      sudo apt-get install git-core gnupg flex bison build-essential zip curl
      zlib1g-dev libc6-dev-i386 x11proto-core-dev libx11-dev lib32z1-dev
      libgl1-mesa-dev libxml2-utils xsltproc unzip fontconfig
      ```

2. Configure the Repo Tool for Source Management

The **Repo** tool is used to fetch and manage the AOSP source code from multiple Git repositories. It simplifies the process of downloading large Android projects and handling updates.

1. **Configuring Repo**:
    - Create a directory for your AOSP workspace:
      ```
      mkdir ~/aosp
      ```
      ```
      cd ~/aosp
      ```
    - Initialize the Repo for Android 14:
      ```
      repo init --depth=1 -b android-14.0.0_r67 -u
      https://android.googlesource.com/platform/manifest
      ```
2. **Downloading the Source**:
    - Use the following command to sync (download) the AOSP source code:
      ```
      repo sync -j8
      ```
    - The `-j8` flag sets the number of threads used for downloading; adjust this according to your CPU cores.

3. Download and Set Up the AOSP Source (e.g., Android 14 for Raspberry Pi 4)

Next, you need to set up the AOSP environment specifically for the **Raspberry Pi 4**:

1. **Download Device-Specific Files**:
    - For Raspberry Pi 4, you'll need to download the device and kernel files from a third-party repository:
      ```
      cd ~/aosp
      ```
      ```
      curl -o .repo/local_manifests/manifest_brcm_rpi.xml -L
      https://raw.githubusercontent.com/raspberry-vanilla/android_local_manif
      est/android-14.0/manifest_brcm_rpi.xml --create-dirs
      ```
2. **Sync the Updated Repositories**:
    - After adding the manifest, run the following command again to fetch the additional repositories:
      ```
      repo sync -j8
      ```

3. **Setting Up Build Environment**:

    - Set up your environment for building the source:

      ```
      source build/envsetup.sh
      ```

    - Choose the build target:

      ```
      lunch aosp_rpi4_car-ap2a-userdebug
      ```

4. Lab: Building and Flashing Android Automotive ROM on Raspberry Pi 4

1. **Building the AOSP Image**:

    - After setting up the environment, you can start building the source code:

      ```
      make bootimage systemimage vendorimage -j$(nproc)
      ```

    - This will take time based on your machine's capabilities. Ensure that you have enough storage and RAM.

2. **Flashing the Image to Raspberry Pi 4**:

    - Once the build is complete, you'll find the output in the `out/target/product/rpi4` directory.
    - Use the following steps to flash the image onto an SD card:
        1. Insert the SD card into your computer.
        2. Write the boot image to the SD card:

           ```
           sudo dd if=out/target/product/rpi4/boot.img of=/dev/sdX bs=4M
           ```

           Replace `/dev/sdX` with your SD card device path.

        3. Copy the system image:

           ```
           sudo dd if=out/target/product/rpi4/system.img of=/dev/sdX bs=4M
           ```

3. **Booting Raspberry Pi 4**:

    - Insert the SD card into your Raspberry Pi 4 and power it on.
    - The Raspberry Pi 4 should boot into Android Automotive.

Build System Fundamentals

The AOSP build system is highly customizable and flexible, allowing developers to configure, build, and modify different components of the Android system. Understanding the basics of the build system is essential to working effectively with AOSP.

1. Introduction to Soong and Make

- **Make**: Android traditionally used the **Make** build system, which uses `Makefiles` to define the rules for compiling the source code. However, Android is transitioning to **Soong**, which offers a more modern and efficient build experience.

- **Soong**: This is the new build system that Android has adopted starting from Android 8.0 (Oreo). It uses configuration files (`Android.bp`) written in **Blueprint**, a declarative language, to define how modules are built.

    - **Soong vs. Make**:
        - **Modular**: Soong allows for more modular builds compared to Make.
        - **Speed**: It offers faster incremental builds by only rebuilding what has changed.
        - **Simpler Configuration**: `Android.bp` files are simpler and more readable compared to Makefiles.

2. Overview of `envsetup.sh`, Build Commands (m, mm, mmm), Module Management

- **envsetup.sh**:

    - The `envsetup.sh` script is used to initialize the build environment. It sets up the environment variables and provides access to useful build commands. You must run this script every time you open a new terminal for building AOSP:

      ```
      source build/envsetup.sh
      ```

- **Common Build Commands**:

    - **m**: The `m` command is a wrapper that invokes the correct build system (Soong or Make) to build all the modules in the current build configuration. For example:

      ```
      m -j8
      ```

    - **mm**: This command is used to build a specific module in the current directory and its dependencies.

```
mm
```

- **mmm**: This command builds a module in a specified directory. For example, to build a specific app, you would run:

  ```
  mmm packages/apps/Launcher3
  ```

3. Module Management

AOSP is made up of various modules (e.g., system services, apps, HALs). Understanding how to manage these modules is crucial for customizing Android. Each module has its own `Android.bp` or `Android.mk` file that defines how it is built.

- **Android.bp**: Blueprint file used by Soong to configure the build of a module.
- **Android.mk**: Makefile used in the old build system. Some legacy components may still use Makefiles, though most have transitioned to Soong.

You can manage and customize modules by modifying these files, adding new build targets, or removing unnecessary components.

Summary

This section provides a practical and fundamental guide to setting up and configuring your environment for AOSP development, with a focus on building and customizing Android Automotive systems. Understanding the build system is key to effectively working with Android source code and making modifications for automotive use-cases.

For more details for AOSP environment setup and Raspberry Pi customization:

- AOSP requirements and env. setup [https://source.android.com/docs/setup/start/requirements]
- Raspberry Pi Vanilla [https://github.com/raspberry-vanilla/android_local_manifest]

1.3 Basic Android Build System and Commands

Understanding Build Architecture (Soong, Makefile, Ninja)

The Android build system is a complex yet highly efficient system that compiles the Android Open Source Project (AOSP) code into a fully functional operating system. It uses several tools like **Soong**, **Make**, and **Ninja** to organize and manage the build process.

Soong

- Soong is the main build system for Android since version 7.0. It replaces much of the older **Makefile** system, bringing efficiency and modularity to the build process.
- It uses **Blueprint** files (`Android.bp`) to describe how modules are built.
- Modules in Soong can be libraries, apps, or components of the Android OS itself.
- Soong handles dependency tracking and parallelization of tasks to optimize the build speed.

Example: Sample Android.bp

```
java_library {
    name: "MyLibrary",
    srcs: ["src/**/*.java"],
    sdk_version: "current",
}
```

Makefile

- **Make** was the traditional build system used by Android before Soong. Android still uses it for some legacy components.
- It uses **Makefiles** (`Android.mk`) to define the build rules for each module.

Example: Sample Android.mk

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES := src/main.cpp
LOCAL_MODULE := MyModule

include $(BUILD_SHARED_LIBRARY)
```

Ninja

- Ninja is a low-level build tool that is optimized for handling large projects like Android. It executes the actual build tasks based on the build rules generated by Soong and Make.
- **Soong** generates **Ninja** build files, and **Ninja** executes the build commands, such as compiling source code and linking binaries.

Example: Ninja Build

```
ninja -C out/target/product/rpi4
```

Running and Debugging Automotive Images in the Android Emulator

The Android Emulator is a powerful tool to test Android Automotive images without needing a physical device like a Raspberry Pi or car infotainment system. Setting up the Android Automotive image in the emulator helps test vehicle-related functionality efficiently.

Steps to Run Automotive Images on Emulator:

1. **Step 1: Install the Android Emulator**

    - Use the Android SDK to install the emulator using the following command:

    ```
    sdkmanager "emulator"
    ```

2. **Step 2: Set Up an Automotive Virtual Device (AVD)**

    - Create an Android Automotive Virtual Device using Android Studio's **AVD Manager**.
    - Select a vehicle image from the available system images and configure the emulator's hardware to simulate automotive environments like screen size and car sensors.

3. **Step 3: Running the Emulator**

    - Launch the Android Emulator with the Automotive image:

    ```
    emulator -avd automotive_avd_name
    ```

4. **Step 4: Debugging the Automotive Image**

    - Use **logcat** and **Android Studio** debugging tools to monitor the behavior of your Android Automotive build.
    - You can interact with the **Car API**, run apps, and test UI/UX interactions similar to how it

would work on a real vehicle.

Common Debugging Commands:

- **Logcat**: View logs generated by the system.

  ```
  adb logcat
  ```

- **GDB**: Run the **GNU Debugger** on native code within the emulator.

- **Systrace**: Capture detailed performance traces to analyze system behavior and performance issues.

Android Debugging Bridge (ADB) and Fastboot: Basics for Device Communication

Android Debugging Bridge (ADB) and **Fastboot** are two essential tools for communicating with Android devices and managing their software.

ADB (Android Debug Bridge)

ADB is a versatile command-line tool that enables communication between a computer and an Android device (physical or emulator). It is widely used for tasks like installing APKs, running shell commands, and viewing device logs.

Common ADB Commands:

- **adb devices**: Lists all connected Android devices.

  ```
  adb devices
  ```

- **adb install**: Installs an APK on the device.

  ```
  adb install myapp.apk
  ```

- **adb logcat**: Streams system logs.

  ```
  adb logcat
  ```

- **adb shell**: Opens a remote shell to run commands on the device.

  ```
  adb shell
  ```

Fastboot

Fastboot is a tool used to flash partitions on Android devices, especially during the boot process when the device is in fastboot mode. It is commonly used for flashing system images, recovery, bootloaders, etc.

Common Fastboot Commands:

- **fastboot devices**: Lists all devices connected in fastboot mode.

  ```
  fastboot devices
  ```

- **fastboot flash**: Flash a partition (e.g., boot, recovery, system).

  ```
  fastboot flash boot boot.img
  ```

- **fastboot reboot**: Reboots the device from fastboot mode.

  ```
  fastboot reboot
  ```

Lab: Change Boot Animation

In this lab, you will customize the **boot animation** of Android Automotive by replacing it with a new custom animation.

Steps:

1. **Prepare Your Boot Animation**:
    - Define the boot animations graphics and audio files following this recommended format https://cs.android.com/android/platform/superproject/+/android-14.0.0_r61:frameworks/base/cmds/bootanimation/FORMAT.md
    - Boot animations are stored in `/system/media/bootanimation.zip`.
    - You can create a custom boot animation by following the structure in the `bootanimation.zip` file, which consists of frames and configuration files.
2. **Test by replacing the current Boot Animation**:
    - ```
      Enter root mode to have access to a system file
      adb root
      adb remount
      ```
    - ```
      Push new animation file to the recommended path(s)
      adb push bootanimation.zip /system/media/bootanimation.zip
      ```
    - ```
      Optionally change the animation file mode
      adb shell chmod 644 /system/media/bootanimation.zip
      ```
    - ```
      Reboot the device and check the animation
      adb reboot  # restart to see the new animation
      ```
3. **Add the animation to the product configurations**
    - ```
      Copy boot/shutdown/reboot animation files to recommended path(s)
      PRODUCT_COPY_FILES +=
      [animation_path/bootanimation.zip]:system/media/bootanimation.zip
      ```
    - ```
      Rebuild the product ROM image
      make bootimage systemimage vendorimage -j$(nproc)
      ```

Lab: Integrate Apps in Android Build System

System apps are embedded in the `/system/` partition and are available to the user without installation. In this lab, we will install an APK as a system app on an Android Automotive image.

Steps:

1. **Define the prebuilt app module configurations**
    1. Create new folder structure for the apps
    2. Create Android.mk that will include info of single or multiple apps

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := MY_APP_MODULE
LOCAL_MODULE_TAGS := optional
LOCAL_LICENSE_KINDS := legacy_restricted
LOCAL_LICENSE_CONDITIONS := restricted
LOCAL_SRC_FILES := $(LOCAL_MODULE).apk
LOCAL_MODULE_CLASS := APPS
LOCAL_MODULE_SUFFIX := .apk
LOCAL_BUILT_MODULE_STEM := package
LOCAL_CERTIFICATE := PRESIGNED
LOCAL_DEX_PREOPT := false
LOCAL_PRIVILEGED_MODULE := true

include $(BUILD_PREBUILT)
```

    3. Add the defined module name to the product packages

PRODUCT_PACKAGES += MY_APP_MODULE

2. **Test by injecting App into file system**

- Enter root mode to have access to a system file

```
adb root
adb remount
```

- Push new animation file to the recommended path(s)

```
adb push MY_APP_MODULE.apk /system/app/MY_APP_MODULE/MY_APP_MODULE.apk
```

- Reboot the device and check the animation

```
adb reboot
```

- Verify Installation

```
adb shell pm list packages | grep myapp
```

Lab: Flash Partitions Using Fastboot and Run ADB for Diagnostics

Flashing partitions is an essential task for system updates, kernel modifications, or device recovery. This lab focuses on using **Fastboot** to flash partitions and **ADB** to run diagnostics.

Steps:

- **Flash Using Fastboot**
  1. **Boot into Fastboot Mode**:
     - Power off your device and reboot it into fastboot mode. This usually involves holding the power button and volume down button together.
     - Alternatively, use **ADB**:

     ```
     adb reboot bootloader
     ```

  2. **Flash a Partition**:
     - Flash a custom image to the system partition using the **fastboot flash** command:

     ```
     fastboot erase <partition_name>
     fastboot flash <partition_name> <image_file>
     fastboot flashall
     ```

  3. **Reboot the Device**:
     - After flashing the partition, reboot the device:

     ```
     fastboot reboot
     ```

  4. **Run Diagnostics with ADB**:
     - Once the device is booted up, use **ADB** to run diagnostics, check logs, or verify the flashed system.

     ```
     adb logcat
     ```

- **Flash using GUI tools such as Etcher** https://etcher.balena.io/

- **Flash using DD command-line tool**
  `dd if=<AOSP_ROM_image_file> of=/dev/<SD_card_device_path> bs=1M conv=fdatasync`