INTEGRATING L2/L3 MIDDLEWARE OF NXP WITH BROADCAST RADIO MODULE OF ANDROID AUTOMOTIVE OS.

Arsalan Anwari – Student BSc. Technical Informatics

Marco Bosma – Project supervisor

Leo van Moergestel – First examiner

Jaap Crezee – Second examiner

Hogeschool Utrecht

StudentID: 1702825

Date: 09/12/2020

Preface

After spending 3 and a half years studying Technical Informatics at the Hogeschool Utrecht I have reached my graduation year. With all the obtained knowledge in these past years, I would like to propose a graduation project. This thesis rapport is the final document of the graduation project. The graduation period at NXP semiconductors was very intense and fast-paced but educational and I would like to thank my project supervisor Marco Bosma for answering any questions I had about the inner workings of the middleware of NXP which made the initial learning curve much simpler. I would also like to thank my second examiner Jaap Crezee for checking the project plan and thesis rapport for any grammatical or substantive ambiguities.

Versions

Date	Tag	Description
03-11-2020	DOC_INIT	Created document, added title, added preface, added headers, set up the structure of the document, and stylized according to APA-guidelines.
06-11-2020	DOC_PRE	Created Background information (no theoretical framework), added introduction, added material & methods (physical hardware not explained), added deliverables. Added sources, footnotes, new figures, and change figures to greyscale.
10-11-2020	DOC_NC	Almost done with results, waiting to complete experiments with NXP hardware before adding further documentation. The document has not been checked for any spelling mistakes and references are not linked with each other.
16-11-2020	DOC_CK	Concept version 1.0 checked and made changes to the new situation. Still need to add theoretical framework and summery in concept version 2.0 due 30th of November 2020/
21-11-2020	DOC_1.0	Concept version 2.0 made with all chapters complete. Fixed a lot of spelling and grammatic mistakes. Still needs to be checked by examiners and supervisor.
09-12-2020	DOC_2.0	Added changes proposed by Jaap Crezee. This version will also be the final version along side the self-reflection document.

Abstract

This paper will go through the process of running custom code added to the Broadcastradio module[15] of Android Automotive OS and how to test this code inside an emulator. Lastly, the paper will also describe how the L2/L3 middleware of NXP can be adapted to work within this Broadcastradio module[15] and why certain choices have been made during the development process.

Summary

Google has recently released Android Automotive OS which is an extension to the existing Android codebase, which allows manufacturers of infotainment systems like NXP to more easily integrate their hardware inside Android OS. This enabled regular Android applications to communicate with the infotainment system its hardware and potentially any other additional hardware supported by the car manufacturer. Unlike the Android application Android Auto which uses your phone to mirror content on your infotainment system its display, Android Automotive OS is natively running within Android OS in the infotainment system its hardware. Android Automotive OS also allows logic specific software to be reused in new hardware iteration through the Android Framework, which means an embedded software engineer only needs to focus on making any new hardware specific software compliant with a generalized API without having to worry about how this affects the higher level service software.

This paper focuses on writing software for the ‘Mercury’[1] line of SDR chips with an iMX8xx host processor. A newer revision of SDR chips named ‘Quantum’ has been planned to announce, but these chips are still too early into development to be used. But as the software being written only requires the use of the L2/L3 middleware, compatibility issues with the new hardware are unlikely due to the scalable nature of the L2/L3 middleware.

The software written in C++ is only targeted at the radio functionality embedded inside the Broadcastradio module[15] of Android Automotive OS. Here the software implements the Broadcastradio HAL[11] written in C++ included inside the Broadcastradio module[15] and acts as an adapter layer to the L2/L3 middleware. The Broadcastradio module[15] is included in the AOSP repository and any changes to this module requires partial recompilation of the source code in the AOSP repository. AOSP can only be compiled if certain minimum hardware requirements are met explained in the chapter ‘Building AOSP’. During this compilation 4 system images are created which are needed to run the modified instance of Android OS on the Android emulator. The Android emulator also has certain hardware requirements to run properly which are specified in the chapter ‘Android emulator’.

The software implementation of the Broadcastradio HAL[11] is compiled as a background service that is running in Android OS. This means real-time debugging in the emulator with tools like GDB or LLDB is not possible. Throwing and catching C++ exceptions is possible using unofficial methods but it is not recommended. Lastly, C++ assertions and signals are not supported. This means you can only rely on sending log messages to debug any software added to the Broadcastradio module[15]. This is further explained in the chapter ‘Testing and debugging code’.

For the adapter layer with the L2/L3 middleware, a singleton object is used named ‘RadioAdapter’ which communicates with an AuRa component named ‘external AuRa component’. An AuRa component is a ‘main function’ assigned to a thread which is chained with other AuRa component threads which communicate through queues.
This communication is handled by the ‘AuRa Access Library’. Data send to another AuRa component is called a message and data received is called a notification. Multiple messages and

notifications can be sent and received. The ‘external AuRa component’ initializes/registers itself by sending/receiving messages/notifications to/from the ‘SystemControl’ AuRa component. After initialization/registration, it can communicate with the ‘BroadcastRadioManager’ AuRa component to indirectly access radio hardware features like selecting a radio operating mode (DAB+, FM, etc.) or to tune to a specific frequency. This is further explained in the chapter ‘Using the L2/L3 middleware’.

The Broadcastradio module[15] communicates with an HMI(Human Machine Interface) in the form of an Android application called the ‘Radio App’ (see figures 5 and 6). This HMI handles features like displaying the list of available radio stations, displaying the information of the currently selected radio program, and adding radio programs to a list of favorites. By default Android already provides this ‘Radio App’ but this can be overwritten with any other Android application made. Communication between this ‘Radio App’ and the Broadcastradio module[15] is handled by calling callback functions from an ‘ITunerCallback’ object pointer in the Broadcastradio HAL[11] implementation. This object pointer is provided by the ‘Radio App’. The callback functions of this ‘ITunerCallback’ object are defined in the ‘Radio App’ and the Android application developer decided how the application will respond to these callback functions. This is explained in the chapter ‘Communicating with the radio application’.

Currently, there is no access to physical hardware which means the logic of the L2/L3 middleware needs to be simulated in a so-called ‘hardware stub’. This is not a true 1 to 1 hardware simulation but a best-case scenario simulation of the internal logic of the L2/L3 middleware which mimics the behavior of features like creating/registering AuRa components, communication between AuRa components, and running background tasks like scanning for new stations in the area. This is further explained in the chapter ‘Replicating hardware features’.

Table of Contents

Background information

Older infotainment systems require custom made software for each different part of the car, meaning little to no code can be reused in later revisions. For example, switching radio channels can have radically different hardware implementations in two revisions or models of cars, meaning the entire software package (from UX to controlling hardware) needs to be rewritten. In this industry, the software is made in a “single-use” manner to save costs on development time with little to no attention being put to reusability (Hardung, Kölzow, & Krüger, 2004). This means that each manufacturer is somewhat “reinventing the wheel” every time. This opened a new opportunity to create a modular and reusable platform where manufacturers can reuse some of their hardware-specific software underneath a generalized API which makes it easier to reuse logic specific software in the future.

One of these solutions which are starting to get traction is Android Automotive OS. In a recent study by market researcher Frost & Sullivan, it was stated that Android is increasingly in a position to challenge incumbent operating systems, in particular, QNX (Frost & Sullivan, 2020). Although Android Automotive OS is currently in its beta version and available for use with little to no documentation present as of today. It is estimated that the first stable version will be done somewhere near 2022 and 2025 with more documentation. As of today's version, 2.0 is the most stable one.

Android Automotive OS is not a new version of the mobile operating system Android or a modification of it, but rather an extension in the Android codebase which adds new functionality needed to integrate existing hardware specific software within the Android operating system. Here a HAL (Hardware Abstraction Layer) is provided, which an embedded software developer can use to base his software implementation on, where Android handles the rest like displaying content on the user interface, communicating with standalone Android applications, and more. This also allows regular Android application developers to modify their Android applications to work within Android Automotive OS with minimal effort (Android, 2020).

Android Automotive OS should also not be confused with Android Auto which is a regular Android application used to mirror content from your phone on your infotainment system its display. Running Android Automotive OS means running a native Android operating system running on the infotainment itself which has access to the hardware components inside the car. This means Android Automotive OS can access things like the radio hardware and other electronic equipment inside the car directly if the manufacturer so desires (Beedham, 2020).

Theoretical framework

I. **SDR chips** are integrated circuits used for SDR functionality. SDR stands for Software Defined Radio and is a radio communication system where components that have been traditionally implemented in hardware (e.g. mixers, filters, amplifiers, modulators/demodulators, detectors, etc.) are instead implemented utilizing software on a personal computer or embedded system (Wikipedia, 2020). SDR defines a collection of hardware and software technologies where some or all of the radio's operating functions (also referred to as physical layer processing) are implemented through modifiable software or firmware operating on programmable processing technologies. These devices include field-programmable gate arrays (FPGA), digital signal processors (DSP), general-purpose processors (GPP), programmable System on a Chip (SoC), or other application-specific programmable processors. The use of these technologies allows new wireless features and capabilities to be added to existing radio systems without requiring new hardware (WirelessInnovation).

II. **Middleware** is computer software that provides services to software applications beyond those available from the operating system. Middleware makes it easier for software developers to implement communication and input/output, so they can focus on the specific purpose of their application. Furthermore, what is middleware and how does it work? Middleware is software that is in the middle of an operating system and the applications working on it. It permits communication and data management for distributed applications by operating as a hidden translation layer. The term is considered vague since it is used to link two separate applications together (AskingLot, 2020).

III. An **infotainment system** is a collection of hardware and software in automobiles that provides audio or video entertainment. In-car entertainment originated with car audio systems that consisted of radios and cassette or CD players, and now includes automotive navigation systems, video players, USB, and Bluetooth connectivity, Carputers, in-car internet, and Wi-Fi. Once controlled by simple dashboard knobs and dials, ICE systems can include steering wheel audio controls and hands-free voice control (Wikipedia, 2020). An in-vehicle infotainment system is a combination of systems that deliver entertainment and information to the driver and passengers. They do it using audio and video interfaces, touch screen displays, button panels, voice commands, and many other features (Kkozyra, 2019).

IV. In computers, a **hardware abstraction layer (HAL)** is a layer of programming that allows a computer OS to interact with a hardware device at a general or abstract level rather than at a detailed hardware level. HAL can be called from either the OS's kernel or from a device driver. In either case, the calling program can interact with the device in a more general way than it would otherwise (Huang & Wu, 2018). A hardware abstraction layer (HAL) is an abstraction layer, implemented in software, between the physical hardware of a computer and the software that runs on that computer. Its function is to hide differences in hardware from most of the operating system kernel so that most of the kernel-mode code does not need to be changed to run on systems with different hardware. On a PC, a HAL can be considered to be the driver for the motherboard and allows instructions from higher-level computer languages to communicate with lower-level components, but prevents direct access to the hardware (Wikipedia, 14).

V. A tuner is a subsystem that receives radio frequency (RF) transmissions like radio broadcasts and converts the selected carrier frequency and its associated bandwidth into a fixed frequency that is suitable for further processing, usually because a lower frequency is used on the output. Broadcast FM/AM transmissions usually feed this intermediate frequency (IF) directly into a demodulator that converts the radio signal into audio-frequency signals that can be fed into an amplifier to drive a loudspeaker (Minkinzi, n.d.). A tuner can also refer to a radio receiver or standalone audio component that is part of an audio system, to be connected to a separate amplifier. The verb **tuning** in radio contexts means adjusting the receiver to detect the desired radio signal carrier frequency that a particular radio station uses (Wikipedia, 2020).

VI. A public service announcement (PSA) is a message in the public interest disseminated without charge, intending to raise awareness of, and changing public attitudes and behavior towards, a social issue (Wikipedia, 2020). In radio communication, PSA’s can be sent and received by both digital and analog transceivers which are called **radio announcements**.

VII. All wireless access point radios continually scan for other RF transmitters. There are two scanning methods, passive scanning, and active scanning. By default, radios perform both types of scans on all channels allowed by the country of operation, which is the regulatory domain set during initial access point deployment. Channels that are not authorized for unlicensed use and channels that require radar detection with dynamic frequency selection (DFS) are excluded from active scanning. **Background scanning** is a form of active scanning running on a separate thread (Juniper, 2017).

VIII. In this scope, **seamless blending** means switching between analog and digital radio without the user noticing this switch by tweaking the audio output to make it sound more natural and less jittery. One example of the use of seamless blending is with HD Radio. HD Radio is a hybrid technology that utilizes two separate but identical streams of audio simultaneously broadcast over the analog and digital program channel. Due to signal processing delays introduced in the digital audio path of HD Radio system, the analog audio must be delayed to synchronize (aligned in time) to the digital audio. This time alignment is necessary as the HD Radio receiver transitions or blends from the analog to the digital signal when first tuned to a station or at the edge of the digital signal coverage where the receiver will blend back to the analog signal. Aligning the signal requires the main analog audio signal to be delayed approximately 8 seconds to match the digital audio’s time delay. The audio levels between the two audio streams must also be matched so there is a smooth transition when the receiver blending occurs. When the audio delay and level are set correctly, the blend between the analog and digital streams are seamless (HD Radio, 2020).

IX. In computer programming, glue code is executable code (often source code) that serves solely to "adapt" different parts of code that would otherwise be incompatible. Glue code does not contribute any functionality towards meeting program requirements. Instead, it often appears in code that lets existing libraries or programs interoperate, as in language bindings or foreign function interfaces such as the Java native interface, when mapping objects to a database using object-relational mapping, or when integrating two or more commercial off-the-shelf programs. Glue code may be written in the same language as the code it is gluing together, or in a separate glue language. Glue code is very efficient in rapid prototyping environments, where several components are quickly put together into a single language or framework (Wikipedia, 2020). A **glue layer** is a software layer using this glue code.

Introduction

NXP Semiconductors is a Dutch semiconductor manufacturer that creates a wide range of products but is mainly active in the automotive industry. In the Netherlands, NXP is a leader in creating integrated circuits made for tuning analog and digital broadcast radios also referred to as SDR (Software Defined Radio) chips. These chips contain more and more software to implement complex tasks. Some of the higher-level functionality is normally implemented on the host and is referred to as middleware. This middleware is responsible for tasks like scanning the frequency band for strong stations, maintaining a service list, automatically switch to alternative stations, and more. NXP has divided its middleware into three levels where level 1 is closest to the hardware (driver-based) and level 3 is closest to the operating system (logic-based). This paper will focus on the integration of the level 2 and 3 (also abbreviated as L2/L3) middleware with the Broadcastradio module[15] of Android Automotive OS.

Topic and context

In the chapter 'Background information' it was mentioned that Android Automotive OS delivers a good starting base for the developer to integrate its implementation. Android Automotive OS allows you to implement the most essential functionality like tuning or seeking between radio stations, responding to radio announcements, and performing a background scan to find new radio stations as you drive. But is therefore limited to only these features. Implementing more advanced features like blending seamlessly between radio technologies (for example switching from DAB+ to FM) and other vendor-specific features cannot be implemented natively within Android Automotive OS without making drastic changes to the Android code base which removes future compatibility with newer versions of Android.

Therefore next to implementing the available functionality of Android Automotive OS, a method needs to be developed where vendor-specific functionality can be easily implemented within Android Automotive OS without breaking compatibility with future updates to the main Android codebase. There are many different methods of how this can be achieved but this paper will not cover them.

Focus and scope

This document will cover how a custom software implementation (and in particular the L2/L3 middleware of NXP) can be added to the Broadcastradio module[15] of Android Automotive OS and how this custom implementation can be compiled and tested in an emulator. Originally it was planned to also test this software implementation on hardware, but due to issues in the hardware supply due to the Corona outbreak, this had to be scrapped. This document is mainly targeted at embedded software developers who want to learn how Android Automotive OS works on a lower level and how it can be modified for custom functionality. Therefore I will not talk about how Android Automotive OS communicates to the Android Framework or how the glue layer between embedded C++ code and Android application Java code works, but rather how the Broadcastradio module[15] can be utilized for the desired result.

Motivation

NXP has released a demo where they showcase vendor-specific automotive features for their ‘Mercury’[1] line of SDR chips which can work with their line of i.MX6xx[2] and i.MX8xx[2] processors. These ‘Mercury’[1] SDR chips have limited memory which means part of the L2/L3 middleware needs to run on the host processor (for example the i.MX8xx). Recently NXP has also been working on a new line of SDR chips named ‘Quantum’ with more memory and processing power which can run the entire L2/L3 middleware on the SDR chip itself. This offloads a lot of resources on the host processor which can use its increased available resources to do more service level tasks like running Android Automotive OS. Quantum is still in the early stages of development so I will be focused on working on the ‘Mercury’[1] platform with a more powerful i.MX8xx[2] processor. Due to the scalable nature of the L2/L3 middleware, any software written for the ‘Mercury’[1] platform can be reused by Quantum whenever it is further in its development. In figure 1 you can see a schematic of the current demo and what has been realized.

There is currently no software written which allows the L2/L3 middleware to communicate with Android Automotive OS so this paper can be used as the initial research and implementation for this. This paper also will give some further clarification on how the HAL of Android Automotive OS can be utilized as there is little documentation online on the topic at hand.

Questions and objectives

The main objectives of this paper can be summarized as followed:

I. Implementing the L2/L3 middleware of NXP within the Broadcastradio HAL[11] of Android Automotive OS.

II. The development process of the software written from downloading/installing/building to testing/debugging it on an emulator.

These main objectives can easily be converted to main question and sub-questions:

How can I integrate the L2/L3 middleware of NXP in the Broadcastradio HAL to ensure it can be tested on an emulator to such a degree the core hardware functionality is replicated in software?

- *What steps need to be followed to correctly use the L2/L3 middleware of NXP with an external application?*
- *What changes to the L2/L3 middleware of NXP need to be made so it is compatible or easier to use with the Broadcastradio HAL?*
- *What parts of the example inside the Broadcastradio HAL can be reused within the implementation of the HAL?*
- *How can the implementation of the Broadcastradio HAL and the Java radio application communicate with each other?*
- *Which steps need to be followed to get the implementation of the Broadcastradio HAL running inside an emulator?*
- *How can the core hardware functionality of the L2/L3 middleware be simulated in software?*

Overview of the structure

This paper will follow a logical order where I will describe the materials (hardware, software documentation) and methods used to create and test the implementation made in the chapter ‘Materials and methods’. In the chapter ‘Deliverables’ I will describe what products need to be delivered to NXP to complete the graduation of their part where a ‘product’ can either be software or documentation/research.

In the chapters ‘Results’ and ‘Discussion’, I go through the different findings and methods used for the software implementation which will indirectly answer the main and sub-questions where I will also answer why I made certain choices, list and compare the results with the quality criteria of NXP and give a brief evaluation of the results.

In the chapter ‘Conclusion’ I will directly answer the main and sub-questions regarding the results and answer the main question in a few sentences. In the chapter ‘Recommendation’, I will refer to any issues or troubles mentioned in the results and conclusion where I will give some advice on how both parties (NXP and Android) could improve to make future development easier.

This paper follows the guidelines of APA (6th edition) format[3] for academic papers and essays.

Materials and methods

Some software choices made reflect my personal preference and should not be seen as the only options when trying to experiment with Android Automotive OS so feel free to adjust some parts with your personal preference. But in saying that please keep into account that in doing so you might run into some issues not stated in this paper and during my experiments.

Building AOSP

As mentioned in the chapter ‘Background information’, Android Automotive OS is an extension of the main Android operating system within AOSP (Android Open Source Project). Here Android Automotive OS is added in any revision of Android OS above 10.0.0_r14. As of today, some minor changes have been made to the underlying codebase in later revisions so during the experiments another revision of Android OS has been used, namely 10.0.0_r32[4]. Although it is not mandatory it is advised (when talking to a Google employee directly) to use a Linux-based environment to build Android OS with Android Automotive OS underneath it. For my experiments, I used an Arch Linux based distribution named Manjaro[26] as this is what I am most comfortable with. Before you can build AOSP in Arch Linux you need to install some dependencies which can be found in the ‘aosp-devel’ package [5]. To install this package run the following console commands in any directory within the home directory.

```
> sudo pacman -Syyu
> git clone https://aur.archlinux.org/aosp-devel.git
> cd aosp-devel
> makepkg
> sudo pacman -U {some_name_that_differs_for_each_person}.pkg.tar.xz
```

After executing these commands you should have the dependencies installed without any errors. The next step will be to download the AOSP source code and build that. AOSP has some minimum hardware requirements:

- You have at least 200GB of storage space available on your computer.
- You are working with an x86_64 compatible processor.
- You have at least 12-16GB of DDR3 (or above) memory.
- Your processor has at least 2 physical cores.
    - It is recommended to disable "hyperthreading"[27] if you are using an Intel processor as for some applications it increases compile times (casualcoder, 2019). Although it is true that for compilers that are heavily threaded, 'hyperthreading'[27] can be beneficial to reduce the compile times (netvope, 2010). The different compilers used in AOSP do not seem to be one of them as tests with a dual-core intel i7 laptop processor took significantly longer when 'hyperthreading'[27] was enabled.

I downloaded and compiled AOSP on a separate external 500GB SSD to make it easier to test code on the go, but this is optional. If you have hardware compatible with the minimum requirements stated above you can download and build AOSP with the following commands in any directory within the home directory (Optionally create an 'android_main' directory)

```
> mkdir android_main && cd android_main
> repo init -u https://android.googlesource.com/platform/manifest -b
android-10.0.0_r32
> repo sync
> . build/envsetup.sh
> lunch aosp_car_x86_64-userdebug
> m
```

- The command `'m'` is similar to `'make'` but automatically calculates optimal build settings like the number of cores to use with the available memory available. It is recommended by Google to use this command instead of `'make'` (Android, 2020). The command 'lunch' is used for configuring the build environment.

Downloading the AOSP source code can take about 1-4+ hours depending on your internet speed and how busy the servers at Google are. The download package is about 100+GB. After you have downloaded AOSP it will take about 4-28+ hours to build AOSP depending on your CPU speed, the number of CPU cores, the amount of available memory, and the memory speed of your hardware. Building AOSP on my computer took about 6 hours with the following specs:

- Intel Xeon E5-2680v3 8 core CPU
- 40GB ddr3 memory

If the build was successful you should see 4 '.img' files being created somewhere in the directory 'android_main/out/' named 'kernel-qemu.img', 'system.img', 'ramdisk.img' and 'userdata.img'. These 4 files are needed to run Android OS (with optional modifications) on an emulator. The android emulator documentation [7] goes into more detail about what each of these files is used for. But knowing what those 4 files are used for is not relevant for the scope of this paper, but be sure that they do exist and have been changed after the compilation.

Python issue

When trying to compile AOSP you might run into some issues with python2.7 files being interpreted in python3.

- For example: "*File device/generic/goldfish/tools/mk_combined_img.py", line 48 print "'%s' cannot be converted to int" % (line[2]).*"

This is because Android is using the generic command `'python'` and not the specific command `'python2'` during the compilation process. I have made an issue report to fix this [6]. Android has no community board for issue and no git issue page, so Stack overflow is the only place to search for common issues.

Android emulator

Android has made an accurate emulator where Android OS can be tested during development which represents physical hardware. It can also be used to test if changes made to the Android codebase will result in a crash or unexpected behavior. Before you use the emulator make sure that your hardware setup complies with these requirements:

- CPU supports Virtualization Technology (VMX for Intel, SVM for AMD). This can be tested by looking if the command `‘egrep --color 'vmx|svm' /proc/cpuinfo | wc -l’` returns anything above the value `‘0’`.
- A dedicated or integrated GPU is present which supports Vulkan and at least OpenGL 3.

Every time you make changes to the Android codebase (for example adding an implementation to Android Automotive OS code) you need to rebuild AOSP by running the command `‘m’` in the main folder of android (in my case ‘android_main’). This will rebuild any changes made (but this will be much faster on average a couple of seconds as the main AOSP code has already been build) and recreate the 4 images mentioned in the chapter ‘Building AOSP’.

After this rebuild was successful without any compiler errors you can then test the code in the emulator by running the following command in the main android folder (in my case 'android_main'):

```
> emulator -wipe_data -no-snapshot-save
```

- It is recommended to completely delete the data each time you make changes to the Android codebase as you otherwise won't be able to see your changes made in the emulator. By default, Android saves the state and any data on the emulator which will be used after the next run. To prevent this and force Android to perform a "cold boot" you need to supply these two flags after the `'emulator'` command.

If your hardware complies with the requirements and there are no compilation errors during the rebuilding process you should see a window with a user interface as shown in figure 2.

Windows density issue

You might get a warped, stretched, or overlapping user interface with no touch support. This is because the android emulator does not play nice with the density scaling of every GPU or iGPU. Many GPUs have a different method to render or scale pixels so this might be a problem. Luckily this can be easily fixed by forcing the emulator to use a fixed density. When the emulator is started and has booted run this command in a separate terminal:

```
> adb shell wm density 90
```

- This will work with most GPU's at the default emulated resolution of 800x600. If you are still having issues, trying setting a lower or higher value than 90.

Testing and debugging code

Android allows you to either use GDB or LLDB for debugging native android apps running in the emulator[10]. This however is useless for finding any issues to modifications made to implementations of the provided Broadcastradio HAL[11]. Your implementation is compiled and loaded as a background service (not to be confused with an Android app background service) which cannot be debugged at runtime in the emulator. You can however run the background service separately and attach the daemon its PID to GDB. As a background service is similar to an executable [28]. But even this is useless in my opinion as you cannot "halt" the state of the emulator on a breakpoint, only a state of the background service. Halting this background service whilst the emulator is running will in most cases crash the emulator during testing.

The source code of the Broadcastradio HAL[11] can be found in the directory 'android_main/ hardware/ interfaces/ broadcastradio/ 2.0 /default'. Here you can freely change, remove or add any of the ".hpp" and ".cpp" files. Please keep in mind that you also have to change the "Makefile" of Android named 'Android.bp' if you remove or add any new files. Android uses the 'Soong Build System'[12] with 'Blueprints' to configure how and what to compile.

Although it is possible to compile any C++ code with exceptions by providing the compile flag '-fexceptions' to the section 'cppflags: […] ', there is little reason to use or rely on exceptions, as any exception thrown will automatically be ignored during runtime in the emulator without any error message. This is a bug specific to parts of the AOSP code of Android Automotive OS which means using exceptions in any other part of the AOSP code will work. This issue has been reported internally to Android and might be fixed by the time you read this paper. Although it should be mentioned that exception handling in AOSP itself is a bit of a hack as it requires you to throw a “Java” exception inside the default C++ exception by using a glue layer function to retrieve a Java class which internally calls a Java exception (Javier-Sanz, 2013). When a native C++ exception is thrown inside a virtual machine this is not caught, the virtual machine just dies. Practically to “catch” problems like loading a file that does not exist or trying to find an element that does not exist you will have to manually place checks in “if-statements” to prevent undefined behavior. Assertions and signals are also not supported and will be ignored during runtime. So you have to stick with generic log output to see if your program is stuck somewhere or reaches a certain part of the execution process. Sending messages to the log from your C++ code can be done by including `‘<android-base/logging.h>’`. Where you can then send something to the log with a syntax similar to `‘std::cout’` by using `‘LOG(DEBUG) << "Some messasge";’`. The logging documentation of Android explains the different filtering methods and how to use the tool ‘logcat’[13] to receive log output in a terminal at runtime. The most common filter type is ‘DEBUG’ and should be used for any custom code to distinguish between the default log output of Android with your code.

Each background service should have a file ‘service.cpp’[22] which acts as the “main” file of the background service. In the Broadcastradio HAL of Android Automotive OS you can find this file here: ‘android_main/ hardware/ interfaces/ broadcastradio/ 2.0 /default/ service.cpp’. When opening this file you see the line `‘android::base::SetDefaultTag("{SomeTagName}" );’`. This tag name can be used to further filter between log out of a single service. You can only receive log output if Android OS has started and booted.

One common method is to keep spamming the ‘logcat’ command until it displays something. If Android has not started you will get an error message when trying to request the log output which you can ignore.

To read all the log output without any filter you can run this command in a separate terminal:

```
> adb shell logcat
```

To only read the log output of a specific service run this command in a separate terminal:

```
> adb shell logcat {SomeTageName} *:S
```

- In my case I use the tagname: ‘NXP_MW_TEST’.

Development process

Before testing any custom code within Android OS I first took some time studying how the L2/L3 middleware works by looking at some example code given by colleges and reading the SWAS (Software architecture document). I am not allowed to share this documentation but will generalize the workings of the L2/L3 middleware in the chapter ‘Results’.

After I had a better understanding of how the L2/L3 middleware worked and how I should communicate with the AuRa+ protocol I started working on the implementation of the Broadcastradio HAL[11] given by Android Automotive OS. Here I first tested the code in an emulator and made a “hardware stub” for the logic of the L2/L3 middleware. This way I could quickly test whether the overall logic of the code written would work within the Broadcastradio module[15]. Another reason was that the hardware was still under development and not ready in the first period when I started development.

After I showed a short demo using the hardware stub I went to work on testing and modifying the code to make it work on physical hardware, but sadly due to the Corona outbreak and supply issues, I could not get my hand on a prototyping unit to test the code on.

Deliverables

During the graduation project, there were several products I needed to deliver to both parties (NXP and Hogeschool Utrecht) which can be divided as followed:

- NXP:
  - Demonstration with source code of L2/L3 middleware being integrated with Android Automotive OS using a stub and the Android emulator.
  - Documentation of inner and outer workings of the software written for the demonstrations mentioned above.
- Hogeschool Utrecht:
  - Plan of Action
  - Thesis Rapport (this paper)
  - Reflection Rapport
  - Small presentation or pitch explaining what I have learned during my graduation period and defend my thesis.

Results

Android Automotive OS contains a special HAL used for radio communication with the Android framework called ‘BroadcastRadio’[21] contained inside the header file ‘BroadcastRadio.h’[21]. This is an implementation of the underlying interface ‘IBroadcastradio’[14] present in Android OS which already defines a lot of default functionality to communicate with the Android framework. This Broadcastradio HAL[11] is part of a larger Broadcastradio ‘module’[15]. The term module is very generic in Android and refers to any combinations of files which are used to compile to a background service or executable with its Blueprint file. So Android Automotive OS consists of many modules where only the Broadcastradio module[15] and in particular the Broadcastradio HAL[11] is relevant to implement.

Most of the source code of the Broadcastradio HAL[11] can be considered dummy code and replacing or removing unused functionality is something you have to do yourself. However, said that some base functionality (a.k.a some functions) needs to be implemented for Android Automotive OS to work within Android. In the base directory of the Broadcast radio module[15], files ending with a ‘.hal’ are present. These files are written in what is called HIDL[16] (Hal Interface Defenition Language). This language has different use cases in the Android codebase but for Android Automotive OS it is used as a “glue layer” between the underlying C++ code in the Broadcastradio HAL[11] and the higher-level Java code which is used by Android app developer to create applications with.

If you look closely at the names of the HIDL[16] files and the names of the Broadcastradio HAL files you can see some overlap (for example ‘TunerSession.h’[19] and ‘ITunerSession.hal’[18]). When opening one of the HIDL[16] files you can also see that some functions in the section ‘interface’ are also defined in the corresponding C++ files. This means your C++ implementation only needs to include the functions mentioned in the corresponding HIDL[16] files, as the Android application developer only really has access to the functionality specified in the HIDL[16] files. This means a lot of redundant code and files can be removed which are not needed. I have removed a lot of unneeded code and made a cleaner “HAL folder” in the appendices{1}. This cleaner HAL folder is what I have used as a base for the implementation as it makes the HAL more readable.

Using the L2/L3 middleware

The complete middleware of NXP is quite complex as shown in figure 3. Luckily only the top part of the Host (i.MX8) and the ‘SystemControl’ are part of the L2/L3 middleware where mainly the AuRa+ protocol is used to communicate with the L1 middleware over SPI. AuRa+ itself is only used to communicate between the so-called ‘AuRa Components’ (yellow blocks in figure 3). These components, in reality, can be seen as individual tasks running in a thread that can communicate with each other through queues which in itself is handled by the ‘AuRa Access Library’. The entire flow of the application starts with creating the ‘SystemControl’ AuRa components which in turn will create the L1 AuRa components and the ‘BroadcastRadioManager’ AuRa component (this is not clearly shown in figure 3, but does happen). The ‘BroadcastRadioManager’ handles how the radio hardware is used on a higher level (ex. tuning to a frequency or radio station) but does not have access to any hardware-specific functionality (ex. resetting a timer chip). This means communicating with the ‘BroadcastRadioManager’ is preferred by NXP rather than directly using the hardware functionality of the other AuRa components as it is safer and harder to damage the hardware.

To communicate with the ‘BroadcastRadioManager’, you need to first register a new component of your own to the ‘SystemControl’ and then request what is called a ‘QueueLink’ between your component and the ‘BroadcastRadioManager’ through the ‘SystemControl’. This used to be very cumbersome as you had to do a lot of setting up and call functions of the ‘SystemControl’ directly, but after some discussion with Marco Bosma, it was decided that the ‘SystemControl’ now does this initialization of a single external component where the external component only needs to supply some meta-data and its ‘TaskFunction’ (function assigned to the AuRa component thread). This new method can be seen in figure 4.

Once a ‘QueueLink’ has been established between the external component and the ‘BroadRadioManager’, messages can be sent and received between the two components which will be the core method used to implement certain functions of the Broadcastradio HAL[11]. A message is a raw 1d byte array containing so-called ‘values’ which can be represented as one or more bytes. For example, the value ‘frequency’ in snippet 1 is a 32bit value split into 4 bytes. The messages are stored inside queue’s which can be read or written to.

For example in the implementation of the ‘tune()’ function in the file ‘TunerSession.h’[19], an encoded message will be sent to the ‘BroadcastRadioManager’ which in turn will send an encoded message back which can be read and used to change something in the Android framework. These response messages are called ‘notifications’ in the AuRa+ protocol. The size of these messages and notifications can never be larger than 24 bytes. Both notifications and messages have something called an ‘opcode’ which is just a byte value in the first element of the array which can be used to distinguish between different messages and notifications.

Any communication between Android, the ‘external AuRa component’, and the L2/L3 middleware will be explained in more detail in the chapters ‘HAL implementation’ and ‘HAL logic’.

Communication with the radio application

Figure 2 presents the default interface of Android Automotive OS which contains a clickable app named ‘Radio’ which presents you with the default radio application in figure 5. This is a native android application loaded by default to test whether the low-level native C++ code in the Broadcastradio HAL[11] produces the desired result. It is also used to test whether certain functionality like background scanning will not intervein with the main GUI thread of this application. This radio application interacts with the Broadcast radio android module by responding to callback functions. In the file ‘ITunerCallback.h’[17] you can see a large class with virtual functions and in the HIDL[16] file ‘ITunerCallback.hal’[18] you can see which of these functions are mapped to the Java code of the radio application. Normally an Android application developer has to implement the GUI functionality when a callback function is being called from the C++ code in the HAL but in the default radio application, this is already done. The ITunerCallback[17] class has many functions with different use-cases but only ‘onCurrentProgramInfoChanged()’ and ‘onProgramListUpdated()’ are important for the team working on the radio application of NXP (Xperi[29]).

- The first function ‘onCurrentProgramInfoChanged()’ should be used when the user has switched to a new station directly (by selecting one or choosing a frequency) or indirectly (by swiping to the next or previous station). The effects of this function can be seen in the station name, artist, and song change in the bottom info bar shown in figure 5. You need to supply a ‘ProgramInfo’[23] struct from the C++ code.
- The second function ‘onProgramListUpdated()’ should be used when the L2/L3 middleware has performed a background scan which means the internal station list database has been updated, removed, or changed. The effects of this function can be seen in the station list change with different stations in the tab ‘Browse’ in figure 6. You need to supply a ‘ProgramListChunk’[24] struct from the C++ code.

Looking at the dummy code of the files ‘TunerSession.h’[19] ‘TunerSession.cpp’[20] it is shown that an ITunerCallback[17] class is stored inside a shared pointer given by the constructor of the TunerSession[19] class which got its reference the function ‘openSession()’ in the BroadcastRadio[21] class in ‘BroadcastRadio.h’[21]. It might seem weird that the ITunerCallback[17] class is nowhere constructed in the service main file ‘service.cpp’[22], or passed to the TunerSession[19] and BroadcastRadio[21] classes. This is because it is up to the Android application developer to do this.

It has been mentioned that the “*.hal” HIDL[16] files in the Broadcast radio module directory are used as the glue layer with any Java code used by the radio application.

This means during the startup of the radio application the function ‘openSession()’ will be called by the Java code which in turn will supply a self pointer to a java object which can be used in the native C++ code of the HAL to communicate back to the radio application. This is the only way the native C++ code can communicate with the Java code of the radio application and is illustrated in figure 7.

When looking at the cleaner “HAL folder”{1} in the appendices you might see that I am not storing a pointer to the ITunerCallback[17] inside the ‘TunerSession.h’[19] class. This is because I am storing a static pointer to this ITunerCallback[17] class in another file. You can store this pointer everywhere but please make sure the pointer does not go out of scope. If you want to call functions from this callback pointer multiple times from different threads, please make sure you lock it with a mutex.

In the chapters ‘HAL implementation’ and ‘HAL logic’ I will go into more detail about why I have decided to store this pointer in a separate file and how you can supply the struct data needed by the callback functions, ‘onCurrentProgramInfoChanged()’ and ‘onProgramListUpdated()’.

HAL implementation

During a meeting with NXP, it was requested that any custom code which interacts with the L2/L3 middleware should be stored inside its own directory which I named ‘vendor’. The source files of the Broadcastradio HAL[11] should also interact with the L2/L3 middleware by including some header files and the L2/L3 middleware should have minimal dependency on Android itself so it can be easily reused or ported for something else. Another factor is that this code should be able to be compiled to a static library which can be added to the ‘Android.bp’ blueprint file for better portability.

In the chapter ‘Using the L2/L3 middleware’ It was mentioned that some changes have been made to the middleware which made interacting with the middleware easier in the future. During this change, it was stated that only one additional AuRa component can be added to the chain between the ‘SystemControl’ and ‘BroadcastRadioManager’. This in essence means that this adapter code can only contain one AuRa component. Therefore it is logical to use a singleton to limit the number of instances of AuRa components that can be made to one. Using a singleton as the main ‘external AuRa component’ also makes the interaction between the Android code and the custom code in the vendor directory easier, as any function of the Android code can just retrieve an instance and call any needed function from this instance. The vendor directory contains many files but only the file ‘nxp_middleware_adapter.h’ is relevant for the Android code as this contains the code which directly interacts with the L2/L3 middleware.

Any custom code in the vendor directory will be encapsulated in the main namespace ‘nxp’ which can have sub-namespaces like ‘tools’ or ‘message’.

The vendor directory contains a couple of files with their use case, namely:

- ‘nxp_common.hpp’: This file is used to include some common header files, define some common data types, and load the settings which are used in many files in the vendor directory.
- ‘nxp_concurrency_tools.hpp’: This contains concurrency tools as classes like the ‘ThreadFlag’ which make handling resources between the different files in the vendor directory easier.
- ‘nxp_data.hpp’: Contains raw image data represented as an array of bytes which is used by the function ‘getImage()’ of the file ‘BroadcastRadio.cpp’ of the Android code.
- ‘nxp_messages.hpp’: This contains the datatype ‘Message’ which is just a vector[33] of bytes. Also contains constant variables defined which represent the different types of AuRa+ messages that can be sent to the middleware (see snippet 1).
- ‘nxp_middleware_adapter.hpp’: Contains singleton class ‘RadioAdapter’ which contains different functions like ‘startMiddleware()’ that the android code can use to communicate with the L2/L3 middleware. Stores the ITunerCalback send from the Android code and uses this callback to then interact with the Android framework described in chapter ‘Communication with the radio application’.
- ‘nxp_notification_controller.hpp’: Contains the class ‘NotificationController’ which stores a hash map of ‘NotificationTask’ (function pointer) elements and responds to different notifications send by AuRa components. The ‘RadioAdapter’ class can subscribe (add) or unsubscribe (remove) to different notifications send by AuRa components and decide how the NotificationController should respond to certain notifications received by supplying a corresponding NotificationTask.
- ‘nxp_settings.hpp’: Contains simple #define values settings that can be turned off or on to change the behavior of the logic deciding to not print any debug output to the console and more.
- ‘nxp_tools.hpp’: Contains static functions, with simple logic like splitting and merging binary numbers and more.

The interface, relation, and interaction between classes in these files can be seen in figure 8.

HAL logic

By the rules of the nondisclosure agreement, it is prohibited to show any source code (any file containing a ‘.c’ or ‘.cpp’ extension) contained inside the vendor directory. It is however allowed to share the generalized logic of this vendor directory source code as a black box (meaning not containing any real reference to the code) and the normal logic of any code inside the Broadcastradio HAL[11] folder. In figures 9 to 20, you can see this workflow represented as a state machine diagram with references to the class diagrams shown in figure 8. This workflow is also available in the archive in the appendices{2} as the file ‘hal_middleware_logic.drawio’. A program named DrawIO[25] is needed to open this workflow containing multiple pages corresponding to figures 9 to 20. In figure 9 you can see the 3 main states used throughout the lifecycle of the entire workflow. These figures only cover the perfect scenario when nothing in the lifecycle of the workflow goes wrong and it never terminates. I am not able to explain the termination process of the middleware without showing any source code as this is quite complex.

- The ‘Java Radio App’ (see figures 9 and 11) corresponds to the android radio application written in java (see figure 5).
- The ‘Broadcastradio HAL’ state (see figures 9 and 10) corresponds to the background service of the Broadcastradio module[15] inside the file ‘service.cpp’[22] in the Broadcastradio HAL[11] directory.
- The ‘NXP Middleware’ state (see figures 9 and 12) corresponds to the ‘external AuRa component’ which communicates with the L2/L3 middleware of NXP (in reality talks to the ‘SystemControl’ and ‘BroadcastRadioManager’ AuRa components).

The lifecycle from the perspective of the three different main states:

- ‘Java Radio App’ state: It first starts with some basic initialization of the android application take place and the signal (JAVA_APP_START_SERVICE) is sent to the ‘Broadcastradio HAL’ state to create itself in a new thread.

  It now has to wait for the signal (BROADCAST_ RADIO_ SERVICE_STARTED) of the ‘Broadcastradio HAL’ state that the background service has been started. When it receives this signal has been received two additional signals (JAVA_ APP_ OPEN_SESSION and JAVA_APP_START_PROGRAM_LIST_UPDATE) are sent to the ‘Broadcastradio HAL’ state.

  After that it will create two threads where the states ‘Handle user interaction’ and ‘Respond to ITunerCallback’ are handled).

  - In the state ‘Handle user interaction’ a large list of user interactions are and appropriate signals are sent to the state ‘Broadcastradio HAL’. But mainly it revolved around: “tuning to a station”, “seeking up/down from a selected or favorite station”, “adding/removing stations as a favorite”.
  - The state ‘Respond to ITunerCallback’ remains idle until it receives one of the two signals ‘ON_CURRENT_PROGRAM_INFO_CHANGED’ and/or ‘ON_PROGRAM_LIST_UPDATED’. This state can respond to both signals at the same time asynchronously.
    - If the signal ‘ON_CURRENT_PROGRAM_INFO_CHANGED’ is received the current playing program is changed and this is stored and displayed in the user interface.
    - If the signal ‘ON_PROGRAM_LIST_UPDATED’ is received the currently stored found stations list is updated and displayed. This affects what stations can be tuned to, added to favorites, and what next station will be “seeked” up or down to.

- ‘Broadcastradio HAL’ state: This state will not exist until it receives the signal (JAVA_APP_START_SERVICE) from the ‘Java Radio App’ state that it needs to create itself and execute the state ‘Broadcast service’.

  After that, it will idle in its main loop state but can respond to signals sent by other states.

  - In the state ‘Broadcast service’ some basic configuring of the background service is done and a signal is sent to the ‘NXP middleware’ state to create itself in a new thread.
  - If the signal ‘JAVA_APP_OPEN_SESSION’ is received the state ‘Broadcastradio::openSession’ is executed. In this sub-state, the ‘TunerSession::TunerSession’ is executed which will store the ITunerCallback[17]* from the java application in the ‘RadioAdapter’ class.
  - If the signal ‘JAVA_APP_START_PROGRAM_LIST_UPDATE’ is received the state ‘TunerSession::startProgramListUpdates’ is executed which will set a flag in the ‘RadioAdapter’ class to allow this class to store any new station data coming in its underlying ‘external AuRa component’. Here the signal ‘NXP_START_PROGRAM_LIST_UPDATE’ is sent to the state ‘Background scan thread’ to create itself in a new thread
    - In the state ‘Background scan thread’ the state ‘Perform Background Scan’ is executed which periodically will scan the area for new stations and store them. After that, the stored ITunerCallback[17] is used to send the signal ‘ON_PROGRAM_LIST_UPDATED’ to the state ‘Java Radio App’. After that, it will go back to scanning the area and storing the found stations
  - If the signal ‘JAVA_APP_TUNE_TO’ is received the state ‘TunerSession::tune’ is executed which will just send a message to the ‘RadioAdapter’ which in turn will forward it to the ‘external AuRa component’ (which then forwards it to the ‘BroadcastRadioManager’ component) to switch the currently selected station.
  - If the signal ‘JAVA_APP_SEEK’ is received the state ‘TunerSession::scan’ is executed which will just send a message to the ‘RadioAdapter’ class which in turn will forward it to the ‘external AuRa component’ (which then forwards it to the ‘BroadcastRadioManager’ component) to switch to the next or previous found station from the current station based on the message sent to the ‘RadioAdapter’ class.

- ‘NXP Middleware’ state: This state will not exists until it receives the signal (BROADCAST_RADIO_SERVICE_STARTED) from the ‘Broadcastradio HAL’ state that it needs to create itself in a new thread. It first will start with the initialization process to register itself with the ‘SystemControl’ and received a ‘QueueLink’ to the ‘BroadcastRadioManager’ AuRa components (see figure 4). After that, it will wait for an event that specifies that one or more AuRa components have sent it a notification. If this event is triggered the state will read all the different notifications from all the AuRa components it can communicate with. All these notifications are sent to the ‘NotificationController’ class which handles what to do for each notification. There are a lot of different notifications specified in the AuRa+ protocol but only 2 are relevant for the scope of this workflow namely ‘RECEIVER_INFO_NOT’ and ‘STATION_NOT’.
    - ‘STATION_NOT’ is received whenever a tune or seek request has been made to the ‘BroadcastRadioManager’ component. Sadly this notification does not contain the needed metadata to send to the stored ITunerCallback[17] so therefore in the NotificationTask function of this notification the message ‘GET_RECEIVER_INFO’ is sent to the ‘BroadcastRadioManager’ component.
    - ‘RECEIVER_INFO_NOT’ is received whenever the ‘GET_RECEIVER_INFO’ message has been sent previously by any AuRa component. This notification contains all the necessary metadata like the frequency in Hz and pi-code needed to send to the store ITunerCallback[17]. After that, the stored ITunerCallback[17] is used to send the signal ‘ON_ CURRENT_ PROGRAM_ INFO_ CHANGED’ to the state ‘Java Radio App’.

Replicating hardware features

Due to limitations in the supply of test hardware and to reduce the development time needed for the demonstration of NXP it was advised to create a “hardware stub” which could replicate the behavior of the hardware in the middleware. This however should not be confused with emulating hardware directly most commonly used to play older retro games like Mario on your computer by emulating the hardware 1 to 1. Rather this hardware stub should only mimic the behavior of the middleware on a higher level. This means returning certain predefined values when a function is called with certain parameters or sending predefined data to the external (optionally slightly modified at runtime) AuRA application periodically to mimic behavior like background scanning. In the chapter ‘HAL logic’ it was mentioned that sharing source code by any means was prohibited which also applies to this hardware stub as this contains a reference to actual functions and variables used in the L2/L3 middleware. But sharing the logic of this hardware stub in a generalized matter is allowed.

The hardware stub mainly emulates 8 key features of the middleware and hardware being:

- The creation and initialization process of the ‘SystemControl’ AuRa component.
- The creation and initialization process of the ‘BroadcastRadioManager’ AuRa component.
- The registering of an ‘external AuRa component’.
- Receiving a ‘QueueLink’ between the ‘external AuRa component’ and ‘SystemControl’.
- Receiving a ‘QueueLink’ between the ‘external AuRa component’ and ‘BroadcastRadioManager’.
- Sending messages from AuRa components to other AuRa components.
- Reading notifications from AuRa components in other AuRa components.

- Finding new stations in the background scan and sending a notification that the station list has been changed.

Creating AuRa components

To simulate this process a function with the same name in the L2/L3 middleware is used which serves as the AuRa component its main function. By using a C library named ‘pthread’[30] an asynchronous thread is made by using the function provided function ‘pthread_attr_setdetachstate(… , PTHREAD_CREATE_ DETACHED);’ before creating it with ‘pthread_create(…);’. Each of those simulated main functions of the AuRa components contains a while loop that waits on a static Event flag inside a shared file which can be set anywhere this file is included. This mimics the shutdown behavior of the hardware by the user. The ‘SystemControl’ AuRa component is a special case however as this is the only AuRa component that creates new asynchronous threads and assigns ‘main functions’ of AuRa components to them.

Registering ‘external AuRa component’

This can be done by storing an instance of a struct when the function ‘cnrmw_registerExternalComponent()’ has been called. This can also be seen done in the first part of the state machine diagram in figure 4. When this instance has been stored in the hardware stub, the ‘SystemControl’ AuRa component can use this metadata to create the asynchronous thread and assign the main function to the thread embedded inside the metadata. This metadata contains things like the scheduling priority, the delay period, and other parameters needed by the actual hardware to create threads (as the actual hardware does not use the ‘pthread’ library [30]).

Retrieving QueueLinks

A ‘QueueLink’ is a pointer to a struct containing metadata needed for the L2/L3 middleware functions to know what locally stored queue to access and how to access this. On the actual hardware, all the different queues are made pre-hand on the stack with fixed sizes and each queue is stored inside the hardware with a so-called identifier named ‘QueuePairID’ which are random after each bootup. So when an AuRa component requests a ‘QueueLink’ it just receives a pointer to a struct containing this ‘QueuePairID’ which it can then used to write/read to/from a specific queue. Mimicking this in the hardware stub can be done by pre-defining the ‘QueueLink’ structs with ‘QueuePairIDs’ for each different AuRa component. So when a “QueuePair request” has been made in the actual application you can simply use a large switch statement to see which AuRa component has made this request and return the appropriate ‘QueueLink’ struct.

Sending messages

In the actual hardware, messages are stored in a write queue where the asynchronous thread of the receiving AuRa component checks if new data has been written in this queue and acts upon this message by performing some internal logic and sending a notification back and sometimes optionally creating a shared buffer with raw byte data if the content of the notification does not fit in 24 bytes. The hardware stub does something similar. Here the hardware stub retrieves the opcode from the message and checks it against a large switch statement which either just prints some dummy log output or creates a new notification based on the content of the message and stores this notification inside a read queue based on the ‘QueueLink’ given. If specific messages are sent which will always result in a notification larger than 24 bytes a 1d byte buffer will be allocated on the heap which can be accessed by a special function in the ‘AuRa Access Library’.

Reading notifications

To mimic the queues of the actual hardware, an STL queue[31] is used for each different AuRa component. In the chapter above it was mentioned that sending a message in the hardware stub sometimes resulted in new notifications being made and stored in the read queue. This read queue is the same queue used for reading the notification. A mutex is used to lock this resource between the two different functions for reading and writing to the queue of each AuRa component and the reading logic is very similar to the actual hardware as it simply checks the size of the queue before popping an element, pops and stores the front element of the queue in a raw byte pointer given by the user and returns the size before popping.

The read function of the ‘AuRa Access Library’ requires you to pass a raw byte pointer where the actual hardware fills this pointer with a 24-byte notification. Notifications in the STL queue of the hardware stub are stored as STL vectors [32] for simplicity, which means a C library function like ‘memcpy’ [33] needs to be used to copy the data from the STL vector in the raw byte pointer as followed: ‘memcpy(rawBytePointer, vecElem.data(), sizeof(uint8_t) * vecElem.size());’ where ‘vecElem’ is the retrieved element of the read queue.

In the chapter ‘Hal logic’ and figure 12 it was explained that each AuRa component needs to wait for an event to be triggered before reading notifications with the read function. This waiting for an event is handled by another function in the ‘AuRa Access Library’. This logic is mimicked in the hardware stub by requesting the size of the read queue of the corresponding AuRa component and return true if the size is larger than 0 or false if it is 0.

Background scan

The background scan is mimicked as followed. A predefined 2d vector named 'stationDatabase' is made and initialized with elements called entries. Each entry is a list of structs containing metadata like the station name, pi-code, and frequency (see snippet 2). A static variable locked with a mutex inside a file is shared between multiple files including it. This variable acts as the index of found stations. The while loop of the main function of the 'SystemControl' AuRa component waits on a timer set every 40 seconds. After every 40 seconds, the index of found stations is changed with a random number in the range of the entries of the 'stationDatabase'. The entry of the new index of found stations is stored temporarily and transposed to a 1d shared buffer of raw bytes with some additional metadata added to correspond with the AuRa+ protocol. Lastly, a notification is created which contains details to access this 1d buffer with a special function in the 'AuRa Access Library'. This notification is then sent to the read queue of the 'BroadcastRadioManager' AuRa component where the 'external AuRa component' can use the details in the notification to get access to the shared 1d buffer of bytes.

In the chapter 'Sending messages' I already mentioned this special function in the 'AuRa Access Library' which returns a pointer to a temporary of permanent 1d buffer of bytes containing additional metadata of each notification too large to store in 24 bytes. These special notifications contain some bytes representing the values 'DataType' and 'Index'

which are needed by this special function of the 'AuRa Access Library' to distinguish between other shared buffer created. The hardware stub mimics this by comparing the Index value inside a switch and return a pointer to the right buffer allocated on the heap. On the actual hardware, nothing is allocated on the heap but pre-made buffers of different sizes are made on the stack which can be overwritten. Nothing in the middleware is heap-allocated.

Discussion

Substantiation choices

Throughout this paper, some choices were made during the development process. This chapter will go through some of these choices, the reason they were chosen, and how this might have impacted the end-result positively or negatively.

You might ask yourself why one would bother making software compatible with it Android Automotive OS as most people already are familiar with the Android Auto application on their phones. The big reasoning is that software currently developed for the radio part of Android Automotive OS can be seen as a stepping stone to fully migrate all of the hardware NXP makes in the automotive industry to Android Automotive OS. Unlike the phone application Android Auto, Android Automotive OS has full control of the car's hardware and electronic devices. Meaning features like smart climate control, seat heating, and adjusting windows can all be done from one central infotainment system. This also makes adding new features to a car easier, as there is no need to place physical buttons on the dashboard for each new feature added to cars over the years. Car manufacturers can simply add additional features as android apps running Android Automotive OS underneath.

In the chapter ‘Motivation’ it was explained that NXP is looking to switch to a new platform named ‘Quantum’ which is in its early stages, but the software developed in this paper targets the now to be replaced ‘Mercury’[1] platform. So why not wait until quantum is finished before writing software in the first place? The are multiple reasons for this, both on a management and technical level but this can be summarized as follows. Only the L2/L3 middleware needs to communicate with Android Automotive OS. The L2/L3 middleware is defined generically through the AuRa+ protocol and some other standards. This means any future renditions or lineups of hardware should be compatible with this predefined standard. Which in turn means that software that only uses the L2/L3 middleware should be compatible on any hardware platform with sufficient resources. This eliminates any threat that too many changes have to be made to the software which uses the L2/L3 middleware. Another reason is time to market. By delaying this integration with Android Automotive OS to wait on Quantum, you set yourself open to the competition which can take the lead NXP currently has in the Automotive industry away. This can in turn also damage the reputation of NXP as a market leader which can lead to reduced sales and in worst cases a reduced value of their stock price.

It has been mentioned in the chapter ‘HAL implementation’ that a singleton design pattern was used to represent the translation layer named ‘RadioAdapter’ between the ‘external AuRa component’ and the code of the Broadcastradio HAL[11]. This design pattern is quite controversial and often associated with a “bad software practice”. Although this claim is true if a singleton is used in cases where this is not needed, the power a singleton can provide to the readability and usability of the code should not be overlooked. A singleton shines in moments where only one single instance of an object can be made which has to be used in multiple scopes running in one or more threads. This is the case for the ‘RadioAdapter’ class. It needs to be used in multiple files, in multiple scopes of functions, and lambda’s, which in turn can potentially all run in different threads which can be both asynchronous or synchronous with the main thread. The flexibility, readability, and safety a singleton provides in these cases are astonishing, as at each of the above-mentioned stages the ‘RadioAdapter’ can simply be accessed with all its important functionality included without having to make the code too complicated for the user by forcing them to consider things like resource management or when to lock/unlock mutexes. The software currently developed has to be reusable in the future for other developers, so making the code needlessly complicated can be seen as a worse “software practice” than using a singleton. One drawback of using a Singleton is that you have manually reset the “state” between each unit test. This can be done by calling the function ‘RadioAdapter::stopMiddleware()’ at the end of each unit test. I agree that this can be more tedious to do, but this drawback is worth the advantages you gain during actual software development where you do not use the Singleton in different unit tests.

The last topic that will be addressed is the choice to make the hardware stub more simplistic mentioned in the chapter ‘Replicating hardware features’. The first answer is time. There simply was not enough development time to increase the accuracy of the hardware stub let alone make it a 1 to 1 emulation which can take allot of time. The second answer is ‘purpose’, as there is no reason to spend this much time on hardware emulation when the end product will be a simple demo showcased once. This remaining time can be much better spent on fine-tuning the existing code, finding and fixing bugs in the currently written software, or a whole range of other smaller tasks that could be performed.

Quality accountability

In document ‘PlanofAction.docx’ which can be found in the appendices{3}, it was briefly mentioned in the chapter ‘Quality control’ that another team is responsible for testing the code on a production level with methods which are classified. This however is only done in the last phase of the software, meaning at that point there aren’t any major or minor bugs left and the code should be stable, but needs to go through some regression testing and more. Before the software reaches this point however some other tests need to be performed. These are mainly functional and unit tests. Performing unit tests are quite flexible as you can choose out of a large selection of tools. I have chosen the tool ‘Catch2[34]’ as this is what I am most comfortable with. Functional tests have to follow certain guidelines set by NXP. Here the code needs to be tested on an emulator or physical hardware for scenarios like loss of power, spamming certain options, how receiving corrupt data is handled and corrected, and more. I will not cover all of the criteria as that is beyond the scope of this paper.

One additional test that is performed next to the unit and functional tests is testing for memory leaks using a tool named ‘Valgrind[35]’.

On the usability side, any software written has to follow certain naming and code conventions set by NXP documents in their SWAS (Software architecture document). This is to ensure a new developer will not be confused with different types of naming schemes when trying to understand and read the code. This of course should also be substituted with good code documentation. You are free to document your code with your own tool, but most people at the company use Doxygen[36] to document their code with additional comments in the different stages of the code. Although this is not mandatory (as some pieces of the middleware lack this) a readme page with simple instructions on how to set up the build environment, a summary of what each file is used for, and an example of how the user can use the code is all provided in the git repository where the code resides in. With this readme, I hope to somewhat summarize what the code does and help a new user with the initial phase of understanding how to use the code.

All this documentation contains direct references to the middleware so sadly I am not allowed to share these files.

Process evaluation

The initial purpose of the software made and this paper was to explain to a new user how the L2/L3 middleware of NXP could be implemented with the Broadcastradio module[15] of Android Automotive OS. This however can be seen as a stepping stone to gain some additional clarification and knowledge on how other features of the middleware can be integrated into Android Automotive OS. Features that control more of the hardware and electronic devices in the car like the heating and mirrors. This means the development of this project can also be relevant for any future projects concerning integrating middleware features in Android Automotive OS. During this project, I had to collaborate with my project supervisor Marco Bosma and colleges Henk Kelder for guidance and help. Marco mainly helped on the software side with understanding how the middleware works under the hood. Henk mainly helped on the software architecture side, so I could understand how different parts of the middleware communicate with each other and how everything is laid out. If I have not done this in the preface, I would like to personally thank these two for making my life a bit easier when trying to figure out any questions or problems I had. I have learned that you can generally understand how the code of others works without any documentation, but that you cannot do this on your own. In such cases, the expertise of multiple people is needed to figure this out.

Conclusion

Throughout the development cycle, it has been found out that integrating the L2/L3 middleware of NXP with the Broadcastradio module[15] of Android Automotive OS is possible by using an adapter layer which communicates with an 'external AuRa component' in the form of a Singleton object as only one additional 'external AuRa component' can be initialized and created. Some minor changes need to be made to the initialization/registration process of the 'SystemControl' AuRa component to make it more compatible with the Broadcastradio HAL[11] implementation. Almost none of the example code of the Broadcastradio HAL[11] can be used. The Java radio application and the implementation of the Broadcastradio HAL[11] communicate in one direction from the Broadcastradio HAL[11] to the Java radio application by using the ITunerCallback[17] object containing generic functions defined in the Java radio application. To test any added to the Broadcastradio HAL[11] the AOSP source code needs to be downloaded and compiled and for any changes recompiled to create images ran on the emulator. There are strict minimum hardware requirements before any of this can happen. The core functionality of the L2/L3 middleware needs to be simulated using a generic hardware stub which is not a 1 to 1 simulation of the hardware but only a best-case scenario simulation of the logic of the L2/L3 middleware.

This adapter layer needs to translate generic functionality like sending messages or enabling the background scan to L2/L3 middleware specific code which is executed in the 'external AuRa component' itself. This reduces the complexity of using the L2/L3 middleware for new developers using the written software as only one object is used with generic functions to handle communication with the radio functionalities of the L2/L3 middleware. An AuRa component in essence is just a main function assigned to an asynchronous thread. The 'external

AuRa component' needs to communicate with the 'SystemControl' AuRa component for its initialization/registration process and the 'BroadcastRadioManager' AuRa component to indirectly access the radio hardware through so-called 'QueueLinks'. These 'QueueLinks' are used to send messages and/or retrieve notifications between one or more AuRa components. 'QueueLinks' between AuRa components need to be created by sending a QueuePairRequest to the 'SystemControl' AuRa component. Messages send or notifications received follow the standard of the AuRa+ protocol and communication with the 'QueueLinks' is handles by the 'AuRa Access Library'.

One change that has to be made to the L2/L3 middleware is the registration process of an external component to the 'SystemControl' AuRa component. Previously the 'external AuRa component' needed to manually call functions specific to the 'SystemControl' AuRa component to successfully register itself to the chain of AuRa components. This now has been changed to where the 'SystemControl' AuRa component can receive metadata from the 'external AuRa component' through the function 'cnrmw_registerExternalComponent()' and uses this metadata to initialize/register the 'external AuRa component' to the chain of AuRa components. This change had to be made to further reduce the complexity of the adapter layer and allow the adapter layer to be functional as a Singleton object. This however does mean that a special function 'startMiddleware()' needs to be called in the 'main function' of the 'service.cpp'[22] file of the Broadcastradio HAL[11] before any other function of the adapter layer can be used. But this is less tedious than the older initialization/registration process.

Almost no code of the example code of the Broadcastradio HAL[11] can be used as this example code only contains dummy implementations which does not reflect the actual L2/L3 middleware. This means a lot of the example code needed to be removed and in some cases entire files or objects as they were not needed. This cleaning process can be seen in the archive 'clean_hal_dir.zip' in the appendices {1}.

Communication between the implementation of the Broadcastradio HAL[11] and the Java radio application is handled through the ITunerCallback[17]. This is a class with some generic functions declared in the Broadcastradio HAL[11] but defined in the Java radio application. During the initialization of the Java radio application, a pointer to an ITunerCallback[17] made in the Java radio application is sent to the Broadcastradio HAL[11] when the function 'openSession()' of the BroadcastRadio[21] class in the Broadcastradio HAL[11] is called from the Java radio application. This pointer needs to be stored and specific callback functions like 'onProgramListUpdated()' need to be called from the Broadcastradio HAL[11] to notify that the Java radio application needs to change something on the screen or internally in the radio application. This communication is one-directional from the Broadcastradio HAL[11] to the Java radio application.

To test and run any new code added as an implementation of the Broadcastradio HAL[11] inside an emulator you need to have downloaded the AOSP source code from the Android git repository and have built it with certain settings. A lot of dependencies are needed and it is recommended by Google to compile on a Linux based distribution. You also need to have a decent PC to compile the AOSP source code which can take a long time. Next, you need to make sure you have a somewhat recent x86 based CPU that supports virtualization and a semi-modern dedicated or integrated GPU to run the emulator. If files are added or removed from the

Broadcastradio HAL[11] directory the blueprint file ‘Android.bp’ also needs to be changed accordingly. Any change made to source code in the Broadcastradio HAL[11] means AOSP needs to be recompiled and new images need to be generated. There is little to no exception handling, assertion handling, or debugging functionality available for any source code changed in the Broadcastradio HAL[11] as it is compiled as a background service.

To simulate the hardware functionality of the L2/L3 a hardware stub needs to be created which mimics the behavior of the L2/L3 hardware by sending and retrieving correct dummy data between different functions used and mimicking tools used in the L2/L3 middleware. This hardware stub should only simulate the best-case scenario and not simulate the hardware 1 to 1, but only represent the logic of the hardware. Features like background scanning can be implemented in new threads. Premade lookup tables and variables can be used in combination with switch statements to handle different use cases.

The software written, research done and documentation made in this paper and the project, in general, can be utilized as a stepping stone for any further implementations of modules of Android Automotive OS with the L2/L3 middleware of NXP. This paper can also be used as a generic guidebook, to understand how custom C/C++ code can be added and compiled to modules within AOSP as there is little to no documentation about this online.

Recommendation

There are not that many recommendations I could make to NXP directly as their code was quite readable and understandable after reading the documentation and getting help from colleges, but if allowed I would advise them to create a simple 'beginners guide to our middleware' document for new employees at NXP to reduce the introduction time a new developer needs before he can contribute with new software. As I did struggle in the first phases of the project but after a while, everything became more clear to me and I could pick up the pace more easily.

Recommendations to Android however during the development process are discussed in private so cannot be shared in this paper.

References

Android. (2020, 9 10). *Automotive*. Retrieved from Automotive: https://source.android.com/devices/automotive

Android. (2020, 11 3). *Building Android*. Retrieved from source.android.com: https://source.android.com/setup/build/building#build-the-code

AskingLot. (2020, 3 7). *What is considered middleware?* Retrieved from AskingLot: https://askinglot.com/what-is-considered-middleware

Beedham, M. (2020, 8 18). *Android Auto vs Android Automotive: What's the difference?* Retrieved from TheNextWeb: https://thenextweb.com/shift/2020/08/18/android-auto-vs-android-automotive-whats-the-difference/

casualcoder. (2019, 2 27). Retrieved from StackExchange: https://softwareengineering.stackexchange.com/questions/387752/comparison-of-build-times-on-various-hardware-why-non-linear-results

Frost & Sullivan. (2020). *Global Automotive Operating Systems Market.* Retrieved from Android and Linux OS to Challenge Incubent QNX Leadership, Forcast to 2025: https://go.frost.com/EI_PR_KCekani_MD7B_OperatingSystems_Mar19

Hardung, B., Kölzow, T., & Krüger, A. (2004, 9 27). *Reuse of Software in Distributed Embedded AutomotiveSystems.* Retrieved from core.ac.uk: https://core.ac.uk/download/pdf/189742192.pdf

HD Radio. (2020). *Properly Adjust Time and Level Alignment*. Retrieved from hdradio.com: https://hdradio.com/broadcasters/engineering-support/properly-adjust-time-and-level-alignment/

Huang, D., & Wu, H. (2018). *Hardware Abstraction Layer.* Retrieved from Hardware Abstraction Layer: https://www.sciencedirect.com/topics/computer-science/hardware-abstraction-layer

Javier-Sanz. (2013, 12 13). *Catching exceptions thrown from native code running on Android*. Retrieved from StackOverflow: https://stackoverflow.com/questions/8415032/catching-exceptions-thrown-from-native-code-running-on-android

Juniper. (2017, 4 20). *Understanding Wireless Scanning*. Retrieved from Juniper: https://www.juniper.net/documentation/en_US/junos-space-apps/network-director3.1/topics/concept/wireless-scanning.html

Kkozyra. (2019, 7 24). *What is a car infotainment system?* Retrieved from ConciseSoftware: https://concisesoftware.com/car-infotainment-system-guide/

Minkinzi. (n.d.). *Tuner*. Retrieved from minkinzicircuits.com: http://minkinzicircuits.com/html_news/Difference-of-Blind-Via-Hole(BVHBuried-Via-Hole-(BVH)-and-Plating-Through-Hole(PTH)-39.html

netvope. (2010, 4 1). *Impact of hyperthreading on compiler performance*. Retrieved from StackOverflow: https://stackoverflow.com/questions/2015134/impact-of-hyperthreading-on-compiler-performance

Wikipedia. (14, 11 2020). *Hardware abstraction*. Retrieved from Hardware abstraction: https://en.wikipedia.org/wiki/Hardware_abstraction

Wikipedia. (2020, 8 28). Retrieved from https://en.wikipedia.org/wiki/Tuner_(radio)

Wikipedia. (2020, 5 24). *Glue code*. Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Glue_code

Wikipedia. (2020, 10 30). *In-car entertainment*. Retrieved from In-car entertainment: https://en.wikipedia.org/wiki/In-car_entertainment

Wikipedia. (2020, 11 2). *Public service announcement*. Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Public_service_announcement

Wikipedia. (2020, 10 2). *Software-defined radio*. Retrieved from Software-defined radio: https://en.wikipedia.org/wiki/Software-defined_radio

WirelessInnovation. (n.d.). *What is Software Defined Radio*. Retrieved from SoftwareDefinedRadio: https://www.wirelessinnovation.org/assets/documents/SoftwareDefinedRadio.pdf

Footnotes

[1]: https://www.nxp.com/products/audio-and-radio/multi-standard-digital-radio:MULTI-STANDARD-DIGITAL-RADIO#/

[2]:

https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-6-processors:IMX6X_SERIES

https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-processors:IMX8-SERIES

[3]: https://www.scribbr.com/apa-style/format/

[4]: https://android.googlesource.com/platform/manifest/+/refs/heads/android-10.0.0_r32

[5]: https://aur.archlinux.org/packages/aosp-devel/

[6]: https://stackoverflow.com/questions/64230597/file-device-generic-goldfish-tools-mk-combined-img-py-line-48-print-s-can/64230598#64230598

[7]: https://en.wikipedia.org/wiki/ARM_Cortex-A53

[8]: https://en.wikipedia.org/wiki/ARM_Cortex-M#Cortex-M4

[9]: *Upon further notice, sharing the BSP file was not allowed.*

[10]: https://source.android.com/devices/tech/debug/gdb

[11]:

https://android.googlesource.com/platform/hardware/interfaces/+/master/broadcastradio/2.0/default/

[12]: https://source.android.com/setup/build

[13]: https://developer.android.com/studio/command-line/logcat

[14]:

https://cs.android.com/android/platform/superproject/+/master:out/soong/.intermediates/hardware/interfaces/broadcastradio/2.0/android.hardware.broadcastradio@2.0_genc++_headers/gen/android/hardware/broadcastradio/2.0/IBroadcastRadio.h;drc=master;l=0

[15]:

https://android.googlesource.com/platform/hardware/interfaces/+/master/broadcastradio/2.0

[16]: https://source.android.com/devices/architecture/hidl

[17]:

https://cs.android.com/android/platform/superproject/+/master:out/soong/.intermediates/hardware/interfaces/broadcastradio/2.0/android.hardware.broadcastradio@2.0_genc++_headers/gen/android/hardware/broadcastradio/2.0/ITunerCallback.h;drc=master;l=20

[18]:

https://cs.android.com/android/platform/superproject/+/master:hardware/interfaces/broadcastradio/2.0/ITunerCallback.hal;l=1?q=ITunercallback.hal&sq=

[19]:

https://cs.android.com/android/platform/superproject/+/master:hardware/interfaces/broadcastradio/2.0/default/TunerSession.h

[20]:

https://cs.android.com/android/platform/superproject/+/master:hardware/interfaces/broadcastradio/2.0/default/TunerSession.cpp

[21]:

https://cs.android.com/android/platform/superproject/+/master:hardware/interfaces/broadcastradio/2.0/default/BroadcastRadio.h

[22]:

https://cs.android.com/android/platform/superproject/+/master:hardware/interfaces/broadcastradio/2.0/default/service.cpp

[23]:

https://cs.android.com/android/platform/superproject/+/master:out/soong/.intermediates/hardware/interfaces/broadcastradio/2.0/android.hardware.broadcastradio@2.0_genc++_headers/gen/android/hardware/broadcastradio/2.0/types.h;drc=master;l=467

[24]:

https://cs.android.com/android/platform/superproject/+/master:out/soong/.intermediates/hardware/interfaces/broadcastradio/2.0/android.hardware.broadcastradio@2.0_genc++_headers/gen/android/hardware/broadcastradio/2.0/types.h;drc=master;l=807

[25]: https://github.com/jgraph/drawio-desktop/releases

[26]: https://manjaro.org/

[27]: https://techlibrary.hpe.com/docs/iss/proliant-gen10-uefi/s_enabling_hyperthreading.html

[28]: https://stackoverflow.com/questions/2050766/how-to-run-gdb-against-a-daemon-in-the-background

[29]: https://www.xperi.com/

[30]: https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

[31]: http://www.cplusplus.com/reference/queue/queue/

[32]: http://www.cplusplus.com/reference/vector/

[33]: http://www.cplusplus.com/reference/cstring/memcpy/

[34]: https://github.com/catchorg/Catch2

[35]: https://valgrind.org/

[36]: https://www.doxygen.nl/index.html

Snippets

```
    const Message SEEK_FM_DOWN = {
        CNRMW_ARP_SEEK_STATION,     // Opcode
        0x00,                       // version
        0x00, 0x00,                 // Handle
        0x00, 0x00, 0x00, 0x00,     // Length
        0x00, 0x00, 0x00, 0x00,     // Frequency
        0x00, 0x00,                 // PI-code
        0x00,                       // SeekType
        0x00,                       // Direction down (0x01 is up)
        0x40,                       // AudioQualityThreshold 64%
        0x00,                       // FilterType
        0x00, 0x00, 0x00, 0x00,     // FilterData
    };
```

Snippet 1: Example of the AuRa+ message 'CNRMW_ARP_SEEK_STATION' to perform a seek down.

```
std::vector<_stationList> _stationDatabase = {

    // Entry 0
    {
        { "Veronic", 103200000U, 0x83E1 },
        { "Radio10", 103800000U, 0x83D2 },
        { "RadioNL", 104200000U, 0x8421 },
        { "NPO1", 104400000U, 0x8201 },
        { "100% NL", 104600000U, 0x83D3 },
        { "DNO", 104800000U, 0x84E3 }
    },

    // Entry 1
    {
        …
    },

    …
};
```

Snippet 2: Example of the internal database used for mimicking background scanning in the hardware stub

Figures

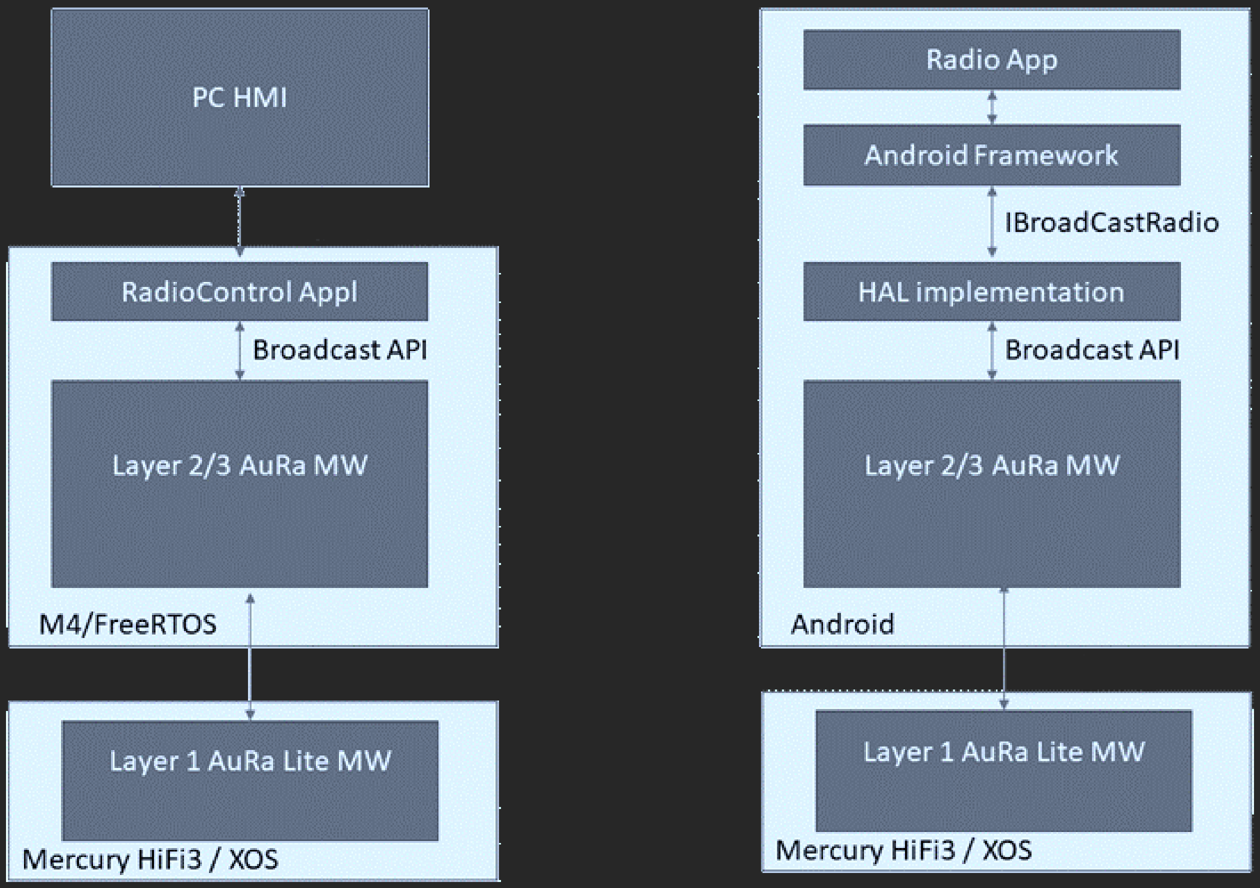


Figure 1. Left shows the current demo with the middleware running on the i.MX6, right the integration with Android Automotive OS labeled as 'Android' to run on i.MX8.

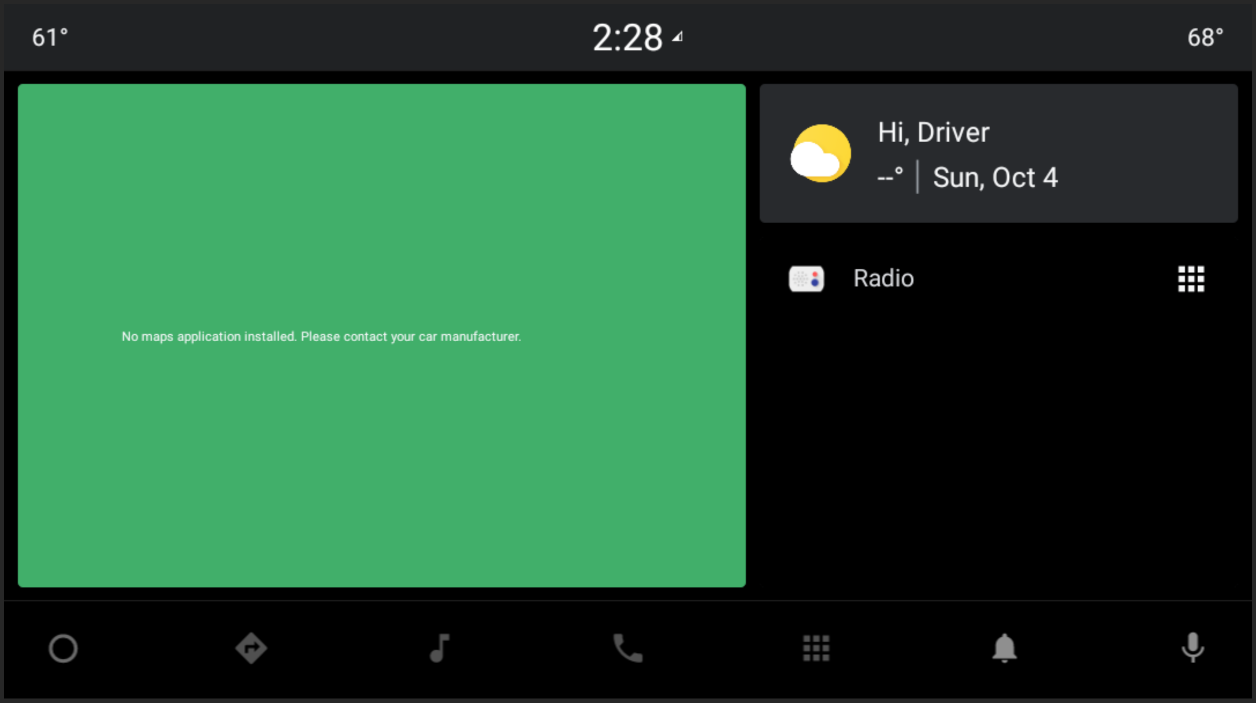

61°
2:28
68°
No maps application installed. Please contact your car manufacturer.
Hi, Driver
--° | Sun, Oct 4
Radio

Figure 2. Android OS with Android Automotive OS underneath running in a car interface on an Android emulator.

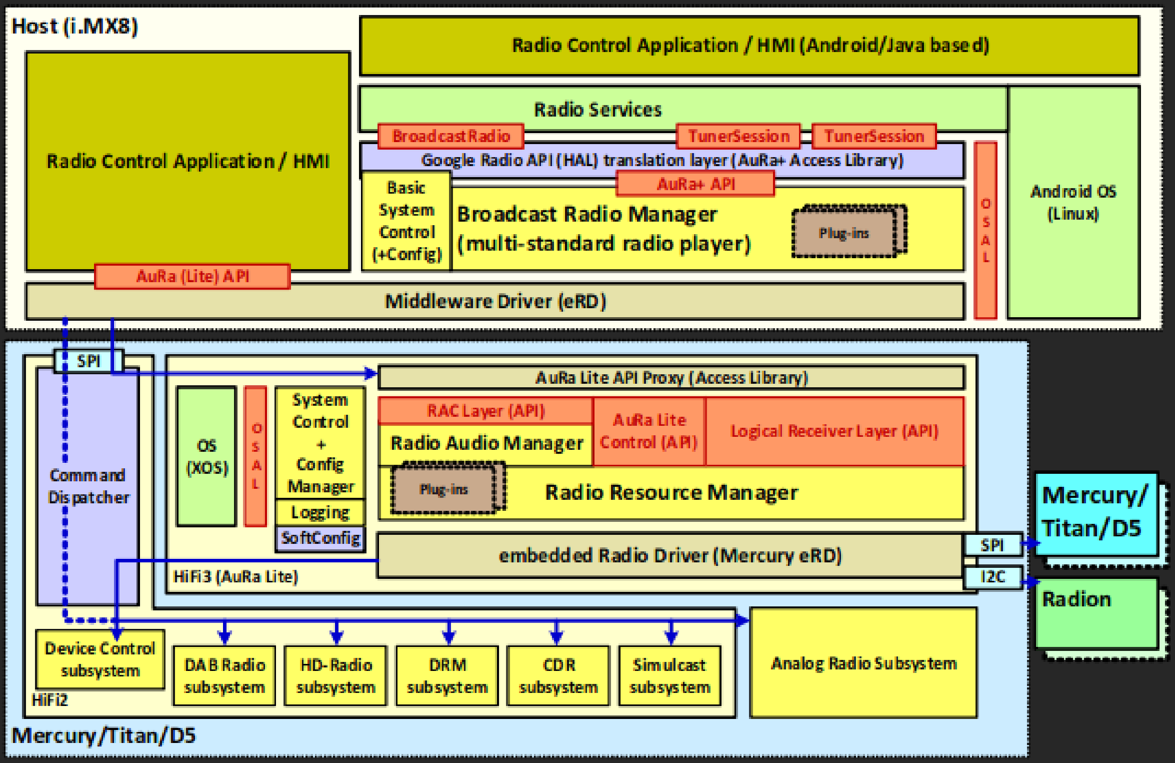


Figure 3. A higher-level overview of the L1/L2/L3 Radio Middleware with AuRa components (light yellow).

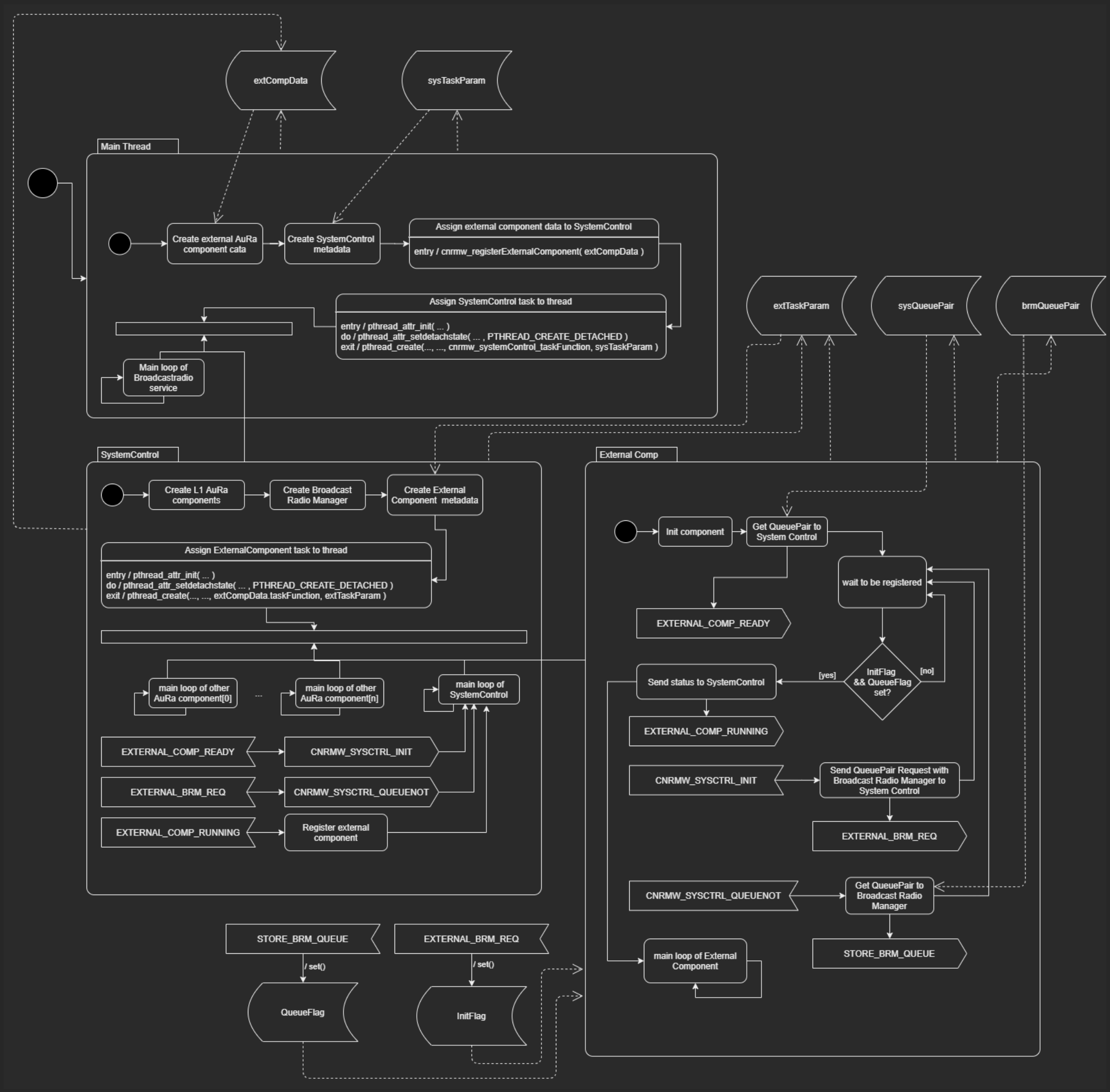

extCompData
sysTaskParam
Main Thread
Create external AuRa component cata
Create SystemControl metadata
Assign external component data to SystemControl
entry / cnrmw_registerExternalComponent(extCompData)
Assign SystemControl task to thread
entry / pthread_attr_init(...)
do / pthread_attr_setdetachstate(... , PTHREAD_CREATE_DETACHED)
exit / pthread_create(..., ..., cnrmw_systemControl_taskFunction, sysTaskParam)
Main loop of Broadcastradio service
extTaskParam
sysQueuePair
brmQueuePair
SystemControl
Create L1 AuRa components
Create Broadcast Radio Manager
Create External Component metadata
Assign ExternalComponent task to thread
entry / pthread_attr_init(...)
do / pthread_attr_setdetachstate(... , PTHREAD_CREATE_DETACHED)
exit / pthread_create(..., ..., extCompData.taskFunction, extTaskParam)
main loop of other AuRa component[0]
...
main loop of other AuRa component[n]
main loop of SystemControl
EXTERNAL_COMP_READY
CNRMW_SYSCTRL_INIT
EXTERNAL_BRM_REQ
CNRMW_SYSCTRL_QUEUENOT
EXTERNAL_COMP_RUNNING
Register external component
External Comp
Init component
Get QueuePair to System Control
wait to be registered
EXTERNAL_COMP_READY
InitFlag && QueueFlag set?
[yes]
[no]
Send status to SystemControl
EXTERNAL_COMP_RUNNING
CNRMW_SYSCTRL_INIT
Send QueuePair Request with Broadcast Radio Manager to System Control
EXTERNAL_BRM_REQ
CNRMW_SYSCTRL_QUEUENOT
Get QueuePair to Broadcast Radio Manager
STORE_BRM_QUEUE
main loop of External Component
STORE_BRM_QUEUE
/ set()
QueueFlag
EXTERNAL_BRM_REQ
/ set()
InitFlag

Figure 4. Workflow to register 'external AuRa component' with 'BroadcastRadioManager'.

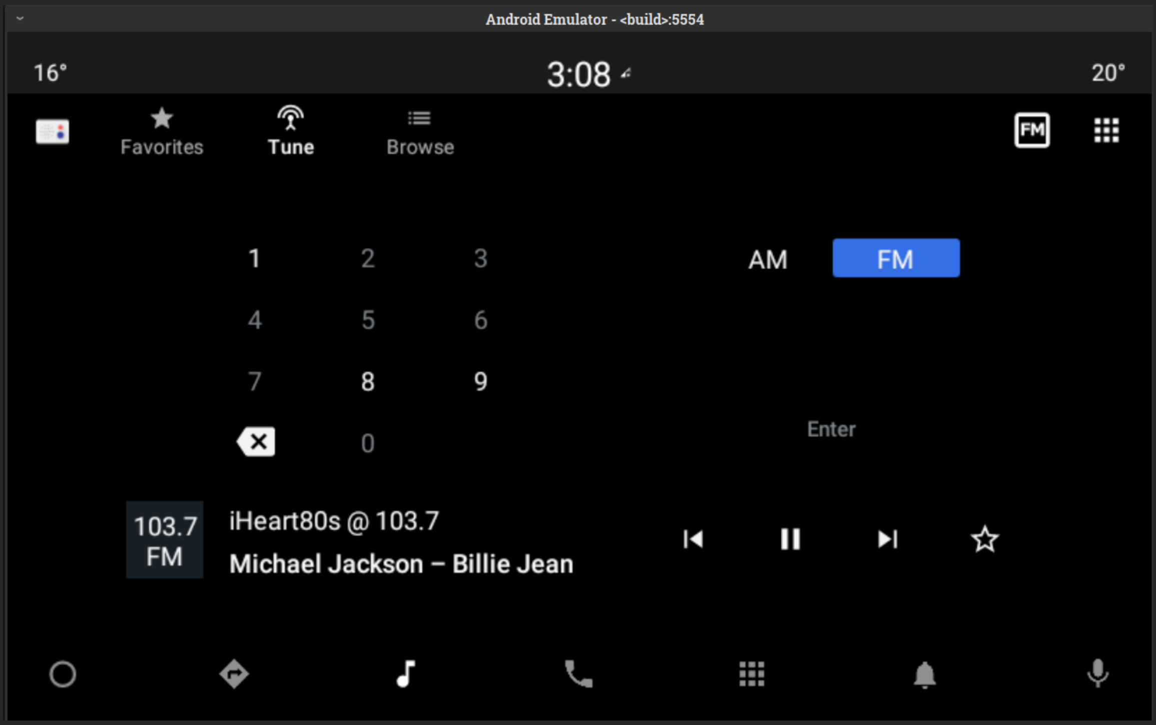


Figure 5. Default radio application of Android Automotive OS

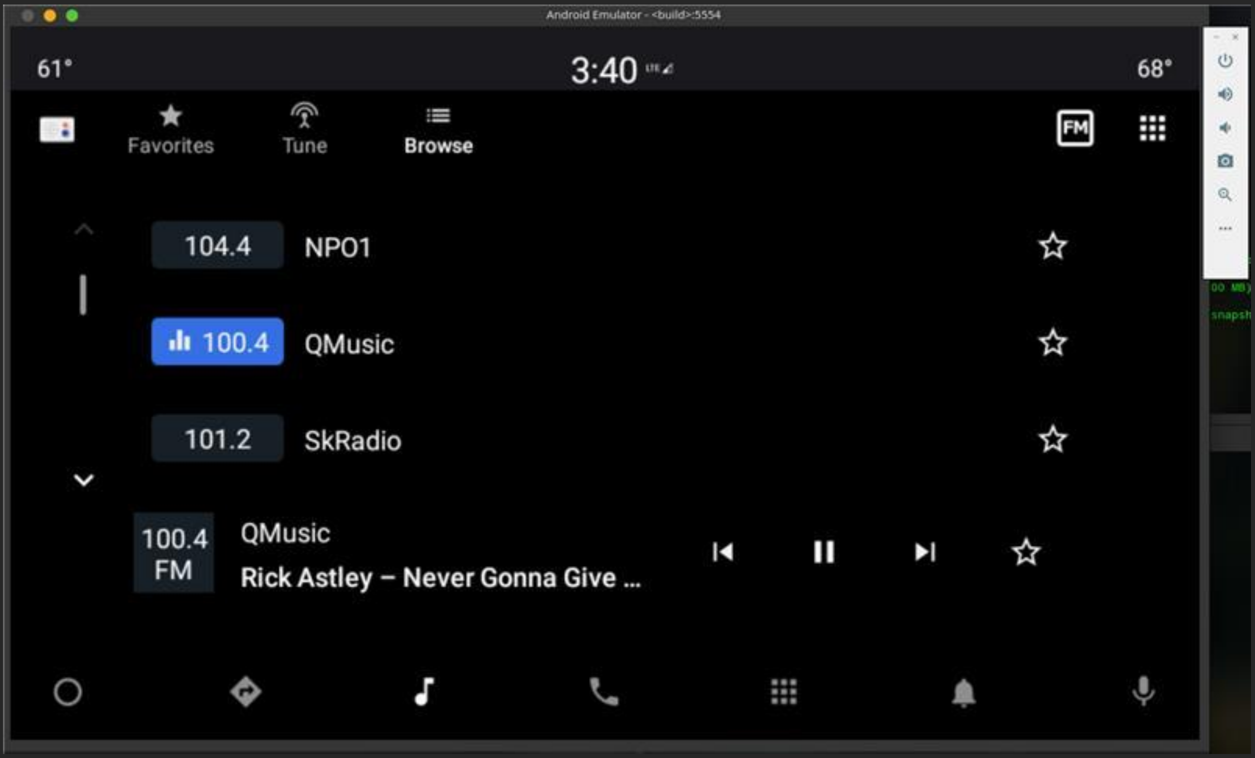


Figure 6: Station list of default radio application of Android Automotive OS

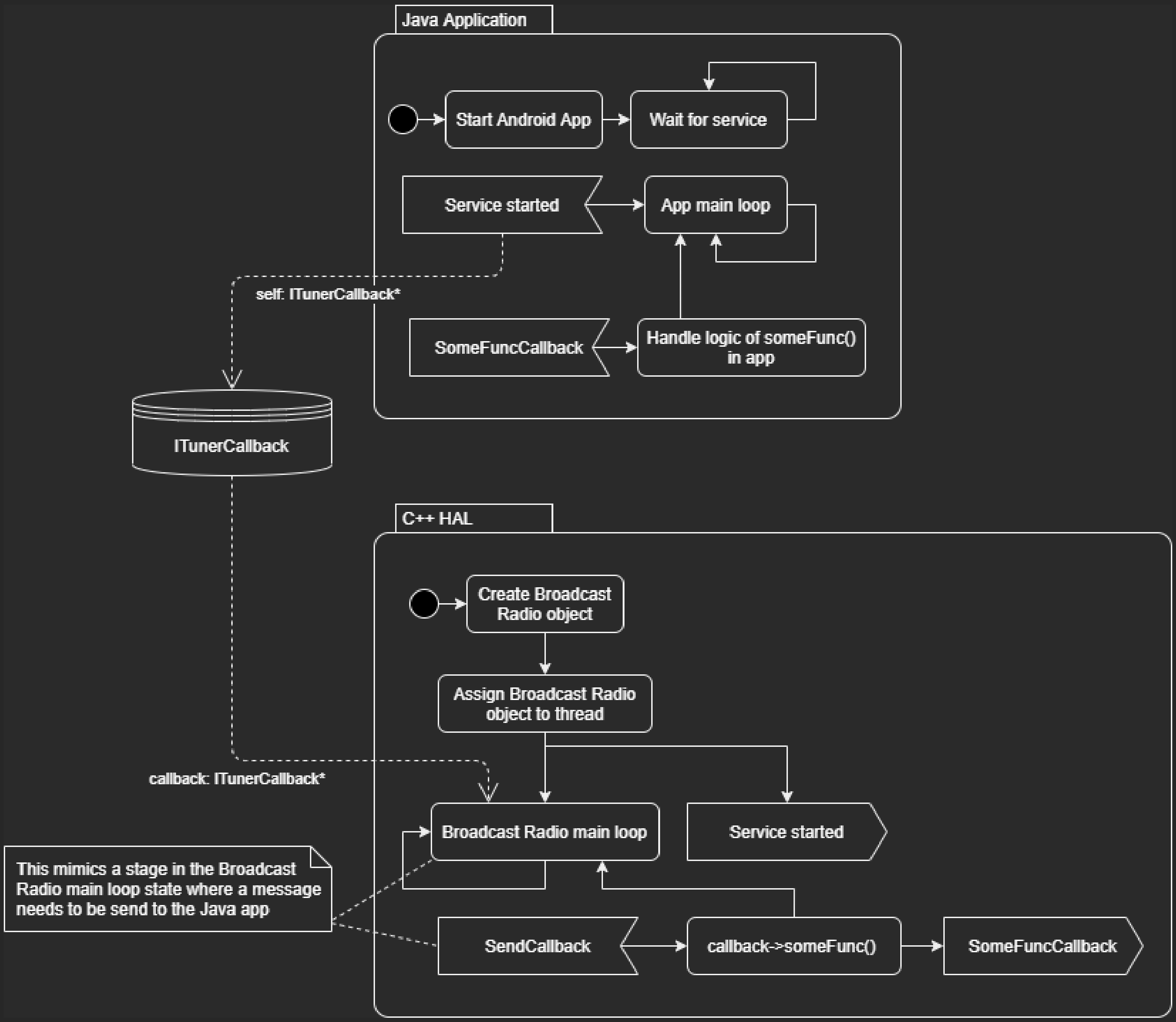

Java Application
Start Android App
Wait for service
Service started
App main loop
self: ITunerCallback*
SomeFuncCallback
Handle logic of someFunc() in app
ITunerCallback
C++ HAL
Create Broadcast Radio object
Assign Broadcast Radio object to thread
callback: ITunerCallback*
Broadcast Radio main loop
Service started
This mimics a stage in the Broadcast Radio main loop state where a message needs to be send to the Java app
SendCallback
callback->someFunc()
SomeFuncCallback

Figure 7: Communication between C++ HAL and Java radio application through ITunerCallback[17]

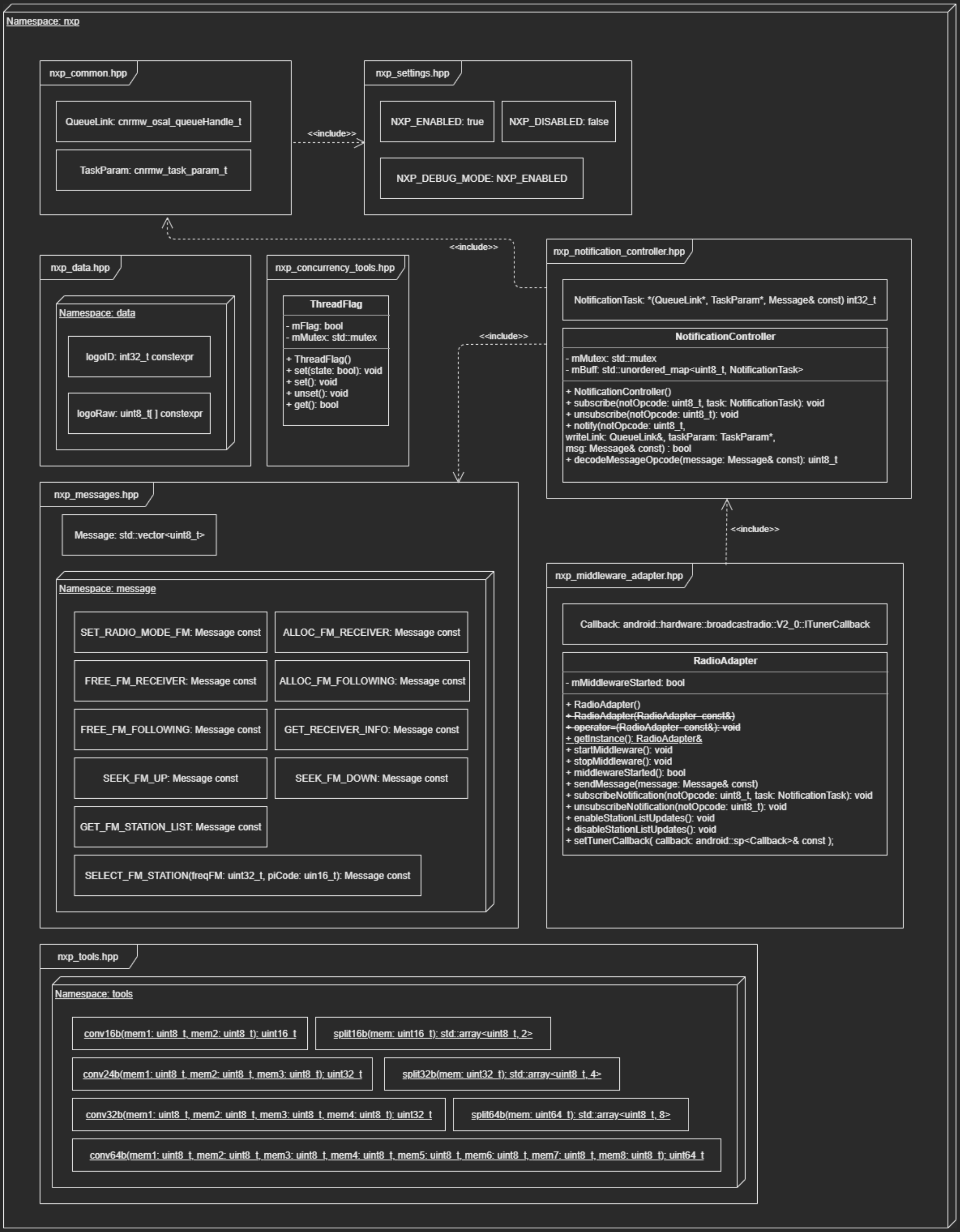

Namespace: nxp
nxp_common.hpp
QueueLink: cnrmw_osal_queueHandle_t
TaskParam: cnrmw_task_param_t
<<include>>
nxp_settings.hpp
NXP_ENABLED: true
NXP_DISABLED: false
NXP_DEBUG_MODE: NXP_ENABLED
<<include>>
nxp_data.hpp
Namespace: data
logoID: int32_t constexpr
logoRaw: uint8_t[ ] constexpr
nxp_concurrency_tools.hpp
ThreadFlag
- mFlag: bool
- mMutex: std::mutex
+ ThreadFlag()
+ set(state: bool): void
+ set(): void
+ unset(): void
+ get(): bool
nxp_notification_controller.hpp
NotificationTask: *(QueueLink*, TaskParam*, Message& const) int32_t
NotificationController
- mMutex: std::mutex
- mBuff: std::unordered_map<uint8_t, NotificationTask>
+ NotificationController()
+ subscribe(notOpcode: uint8_t, task: NotificationTask): void
+ unsubscribe(notOpcode: uint8_t): void
+ notify(notOpcode: uint8_t,
writeLink: QueueLink&, taskParam: TaskParam*,
msg: Message& const) : bool
+ decodeMessageOpcode(message: Message& const): uint8_t
<<include>>
nxp_messages.hpp
Message: std::vector<uint8_t>
Namespace: message
SET_RADIO_MODE_FM: Message const
ALLOC_FM_RECEIVER: Message const
FREE_FM_RECEIVER: Message const
ALLOC_FM_FOLLOWING: Message const
FREE_FM_FOLLOWING: Message const
GET_RECEIVER_INFO: Message const
SEEK_FM_UP: Message const
SEEK_FM_DOWN: Message const
GET_FM_STATION_LIST: Message const
SELECT_FM_STATION(freqFM: uint32_t, piCode: uin16_t): Message const
<<include>>
nxp_middleware_adapter.hpp
Callback: android::hardware::broadcastradio::V2_0::ITunerCallback
RadioAdapter
- mMiddlewareStarted: bool
+ RadioAdapter()
~~+ RadioAdapter(RadioAdapter const&)~~
~~+ operator=(RadioAdapter const&): void~~
+ getInstance(): RadioAdapter&
+ startMiddleware(): void
+ stopMiddleware(): void
+ middlewareStarted(): bool
+ sendMessage(message: Message& const)
+ subscribeNotification(notOpcode: uint8_t, task: NotificationTask): void
+ unsubscribeNotification(notOpcode: uint8_t): void
+ enableStationListUpdates(): void
+ disableStationListUpdates(): void
+ setTunerCallback(callback: android::sp<Callback>& const);
nxp_tools.hpp
Namespace: tools
conv16b(mem1: uint8_t, mem2: uint8_t): uint16_t
split16b(mem: uint16_t): std::array<uint8_t, 2>
conv24b(mem1: uint8_t, mem2: uint8_t, mem3: uint8_t): uint32_t
split32b(mem: uint32_t): std::array<uint8_t, 4>
conv32b(mem1: uint8_t, mem2: uint8_t, mem3: uint8_t, mem4: uint8_t): uint32_t
split64b(mem: uint64_t): std::array<uint8_t, 8>
conv64b(mem1: uint8_t, mem2: uint8_t, mem3: uint8_t, mem4: uint8_t, mem5: uint8_t, mem6: uint8_t, mem7: uint8_t, mem8: uint8_t): uint64_t

Figure 8: Interface, relation, and interaction between classes and files of L2/L3 middleware code in the vendor directory

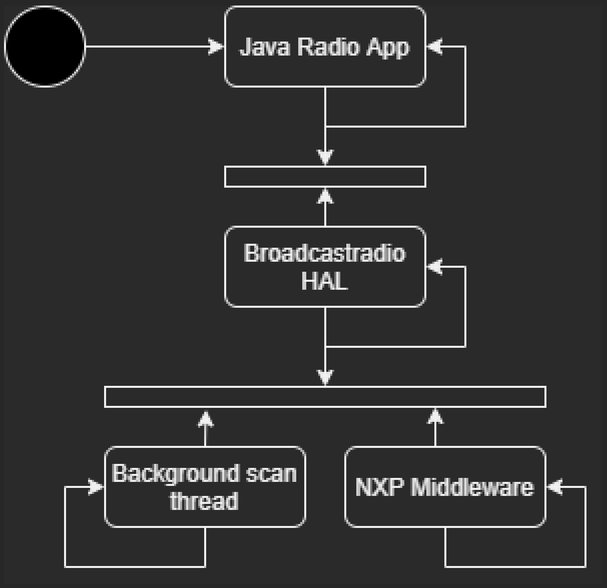


Figure 9: Generalized logic of the entire logic from Android application to L2/L3 middleware

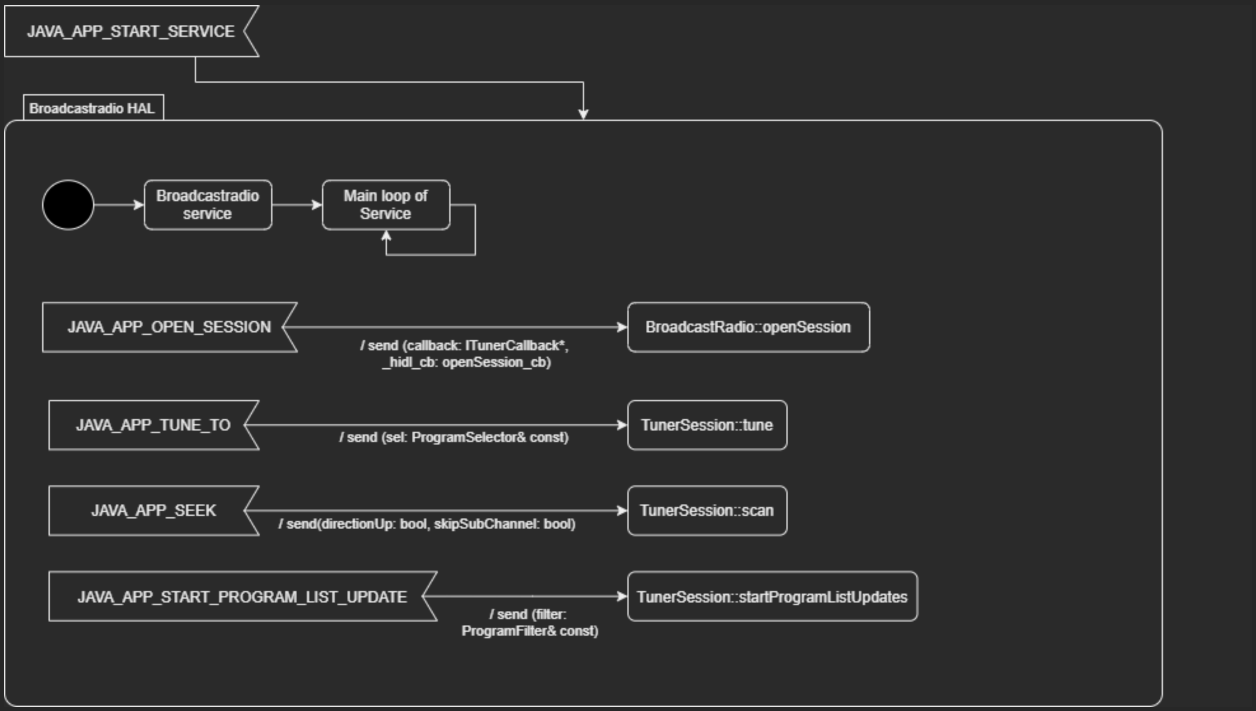

JAVA_APP_START_SERVICE
Broadcastradio HAL
Broadcastradio service
Main loop of Service
JAVA_APP_OPEN_SESSION
/ send (callback: ITunerCallback*, _hidl_cb: openSession_cb)
BroadcastRadio::openSession
JAVA_APP_TUNE_TO
/ send (sel: ProgramSelector& const)
TunerSession::tune
JAVA_APP_SEEK
/ send(directionUp: bool, skipSubChannel: bool)
TunerSession::scan
JAVA_APP_START_PROGRAM_LIST_UPDATE
/ send (filter: ProgramFilter& const)
TunerSession::startProgramListUpdates

Figure 10: logic of the state 'Broadcastradio HAL'

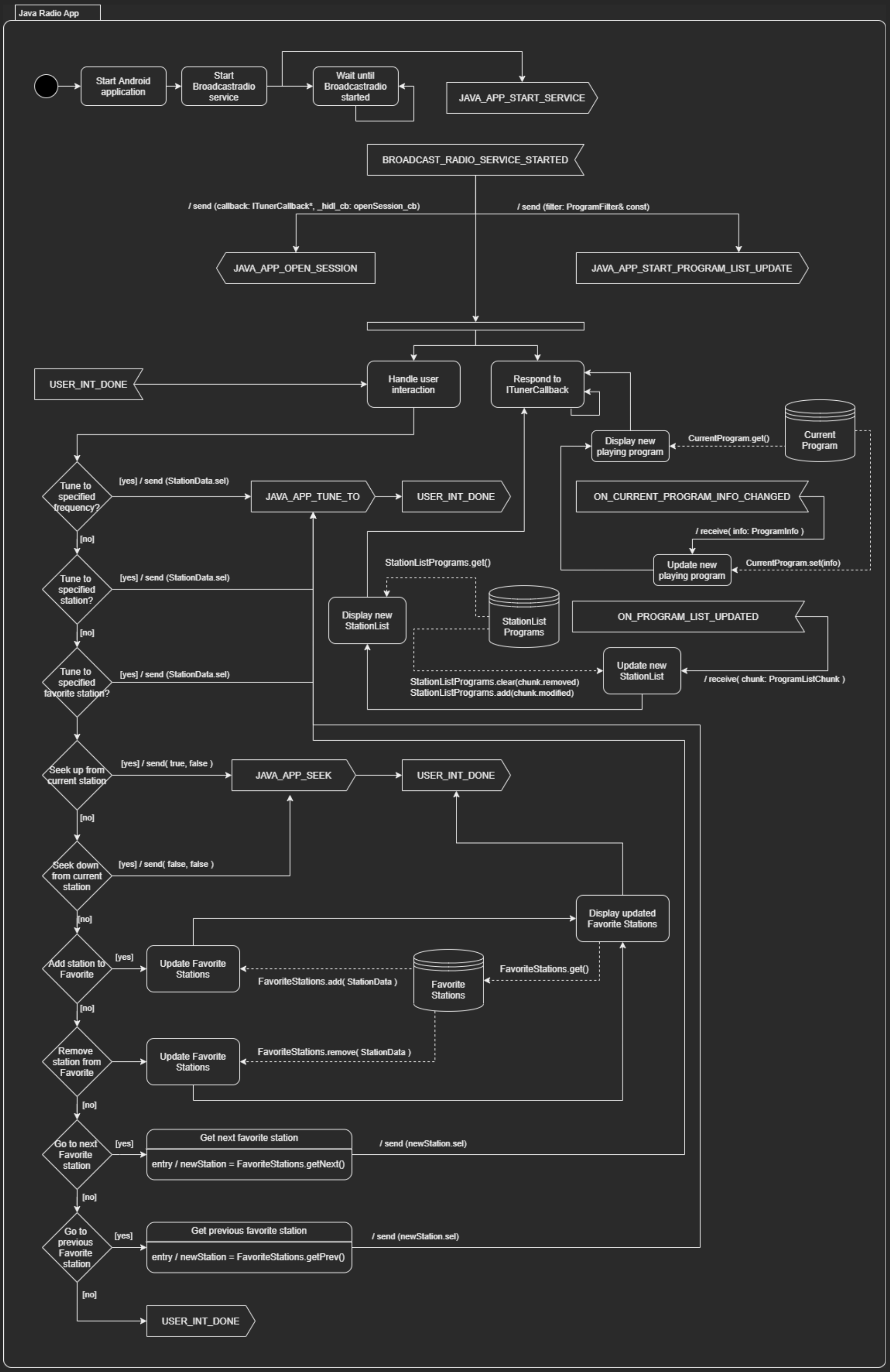

Java Radio App
Start Android application
Start Broadcastradio service
Wait until Broadcastradio started
JAVA_APP_START_SERVICE
BROADCAST_RADIO_SERVICE_STARTED
/ send (callback: ITunerCallback*, _hidl_cb: openSession_cb)
/ send (filter: ProgramFilter& const)
JAVA_APP_OPEN_SESSION
JAVA_APP_START_PROGRAM_LIST_UPDATE
USER_INT_DONE
Handle user interaction
Respond to ITunerCallback
Current Program
Display new playing program
CurrentProgram.get()
Tune to specified frequency?
[yes] / send (StationData.sel)
JAVA_APP_TUNE_TO
USER_INT_DONE
ON_CURRENT_PROGRAM_INFO_CHANGED
/ receive(info: ProgramInfo)
[no]
StationListPrograms.get()
Update new playing program
CurrentProgram.set(info)
Tune to specified station?
[yes] / send (StationData.sel)
Display new StationList
StationList Programs
ON_PROGRAM_LIST_UPDATED
[no]
Tune to specified favorite station?
[yes] / send (StationData.sel)
Update new StationList
StationListPrograms.clear(chunk.removed)
StationListPrograms.add(chunk.modified)
/ receive(chunk: ProgramListChunk)
Seek up from current station
[yes] / send(true, false)
JAVA_APP_SEEK
USER_INT_DONE
[no]
Seek down from current station
[yes] / send(false, false)
[no]
Display updated Favorite Stations
Add station to Favorite
[yes]
Update Favorite Stations
FavoriteStations.add(StationData)
Favorite Stations
FavoriteStations.get()
[no]
Remove station from Favorite
Update Favorite Stations
FavoriteStations.remove(StationData)
[no]
Go to next Favorite station
[yes]
Get next favorite station
entry / newStation = FavoriteStations.getNext()
/ send (newStation.sel)
[no]
Go to previous Favorite station
[yes]
Get previous favorite station
entry / newStation = FavoriteStations.getPrev()
/ send (newStation.sel)
[no]
USER_INT_DONE

Figure 11: logic of the state 'Java Radio App'

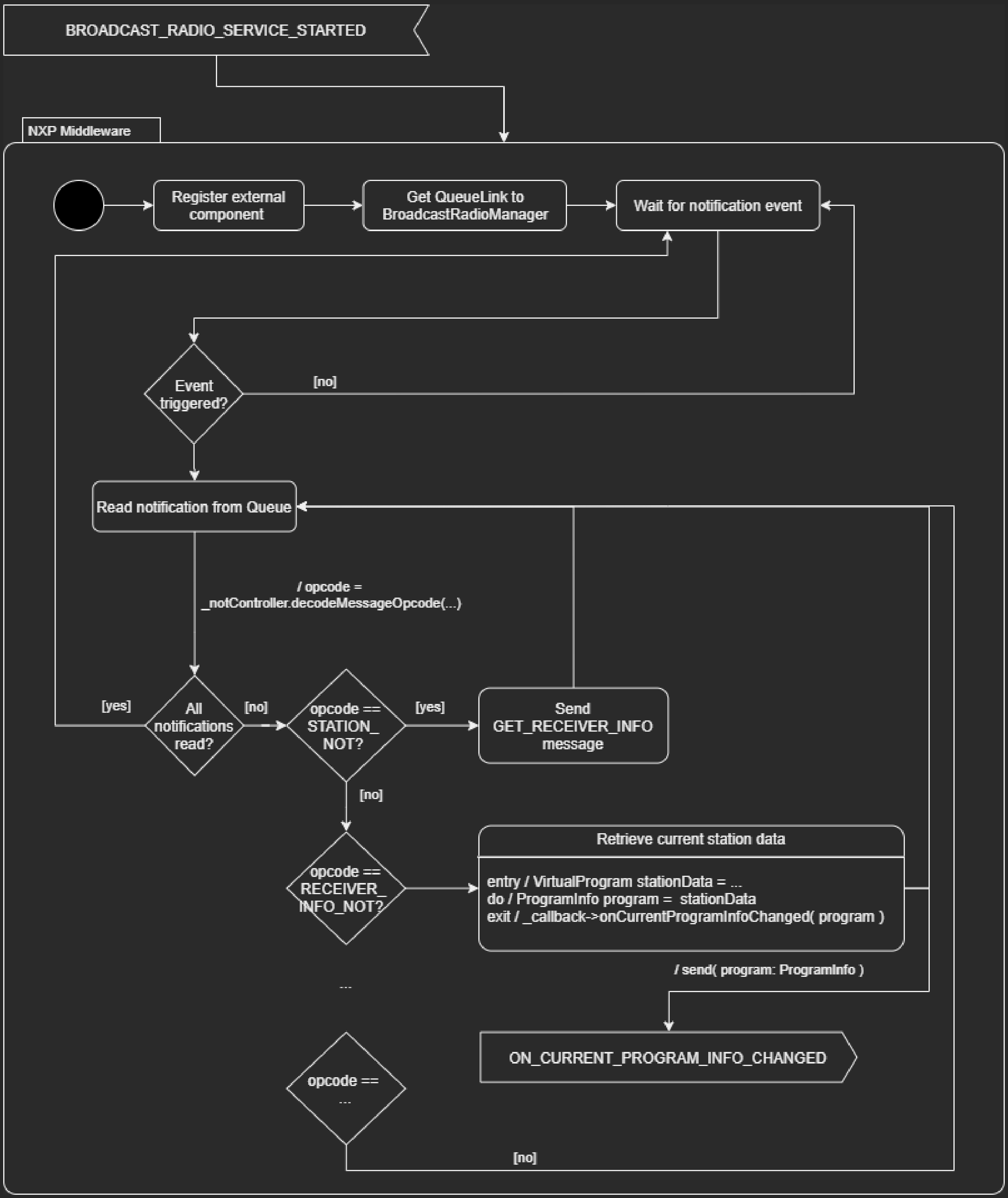

BROADCAST_RADIO_SERVICE_STARTED
NXP Middleware
Register external component
Get QueueLink to BroadcastRadioManager
Wait for notification event
Event triggered?
[no]
Read notification from Queue
/ opcode = _notController.decodeMessageOpcode(...)
[yes]
All notifications read?
[no]
opcode == STATION_ NOT?
[yes]
Send GET_RECEIVER_INFO message
[no]
opcode == RECEIVER_ INFO_NOT?
Retrieve current station data
entry / VirtualProgram stationData = ...
do / ProgramInfo program = stationData
exit / _callback->onCurrentProgramInfoChanged(program)
/ send(program: ProgramInfo)
...
ON_CURRENT_PROGRAM_INFO_CHANGED
opcode == ...
[no]

Figure 12: logic of the state 'NXP Middleware'

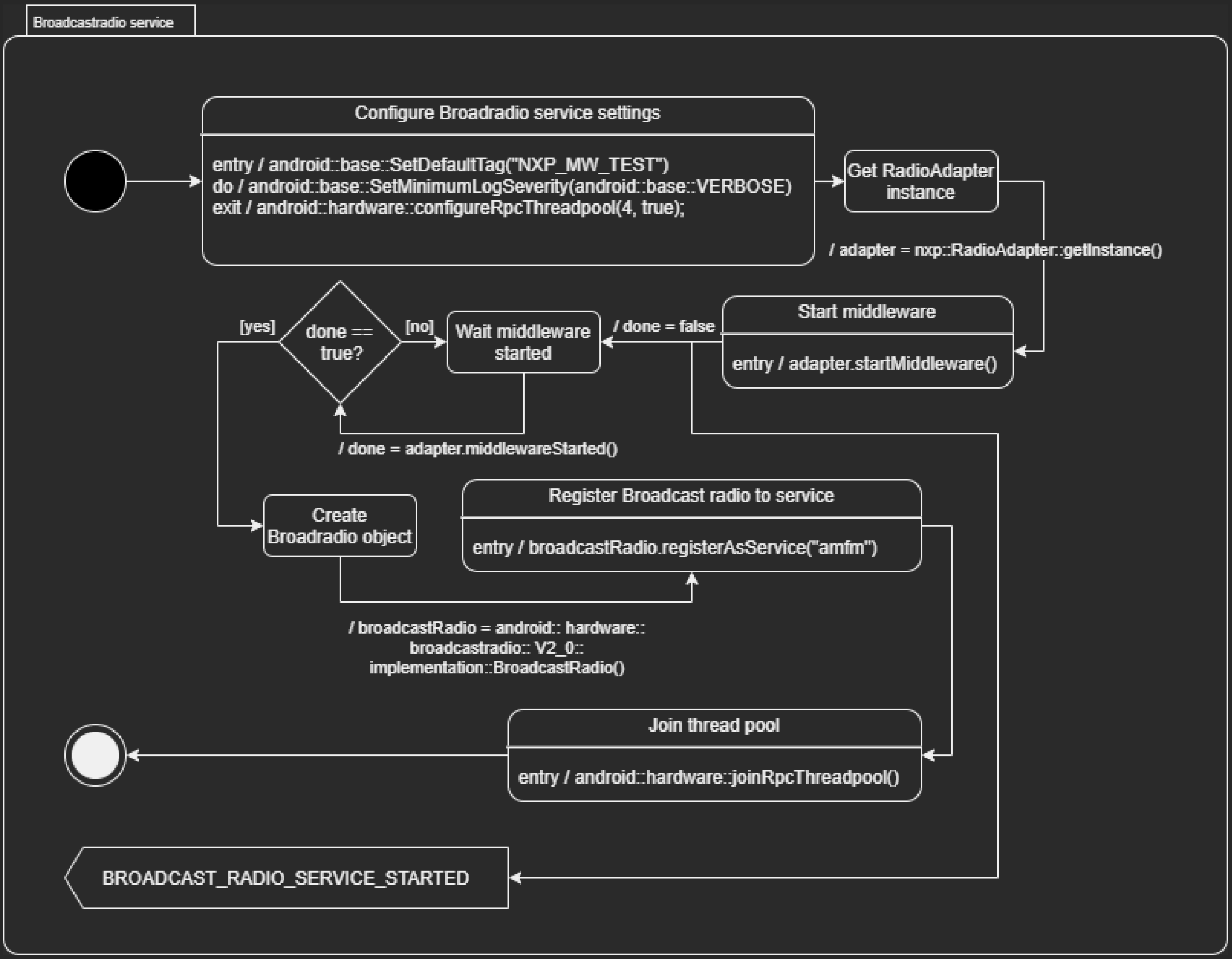

Broadcastradio service
Configure Broadradio service settings
entry / android::base::SetDefaultTag("NXP_MW_TEST")
do / android::base::SetMinimumLogSeverity(android::base::VERBOSE)
exit / android::hardware::configureRpcThreadpool(4, true);
Get RadioAdapter instance
/ adapter = nxp::RadioAdapter::getInstance()
Start middleware
entry / adapter.startMiddleware()
/ done = false
Wait middleware started
[no]
done == true?
[yes]
/ done = adapter.middlewareStarted()
Create Broadradio object
Register Broadcast radio to service
entry / broadcastRadio.registerAsService("amfm")
/ broadcastRadio = android:: hardware:: broadcastradio:: V2_0:: implementation::BroadcastRadio()
Join thread pool
entry / android::hardware::joinRpcThreadpool()
BROADCAST_RADIO_SERVICE_STARTED

Figure 13: logic of the state 'Broadcastradio service'

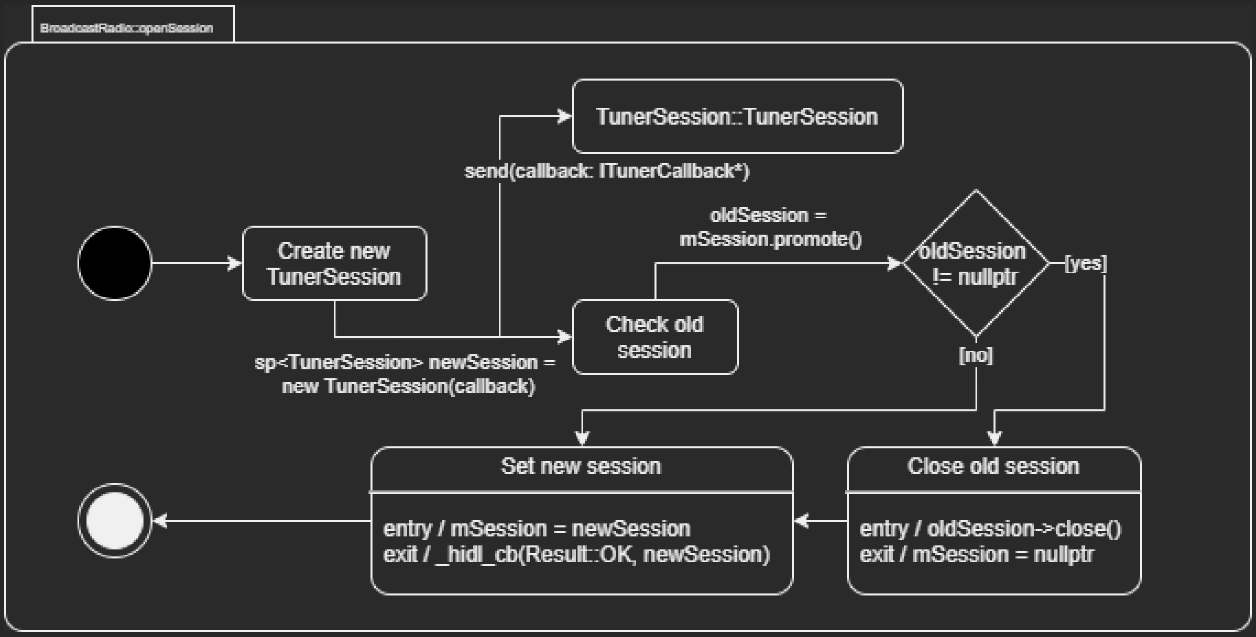

BroadcastRadio::openSession
TunerSession::TunerSession
send(callback: ITunerCallback*)
Create new TunerSession
oldSession = mSession.promote()
oldSession != nullptr
[yes]
Check old session
[no]
sp<TunerSession> newSession = new TunerSession(callback)
Set new session
entry / mSession = newSession
exit / _hidl_cb(Result::OK, newSession)
Close old session
entry / oldSession->close()
exit / mSession = nullptr

Figure 14: logic of the function 'Broadcastradio::openSesion()'

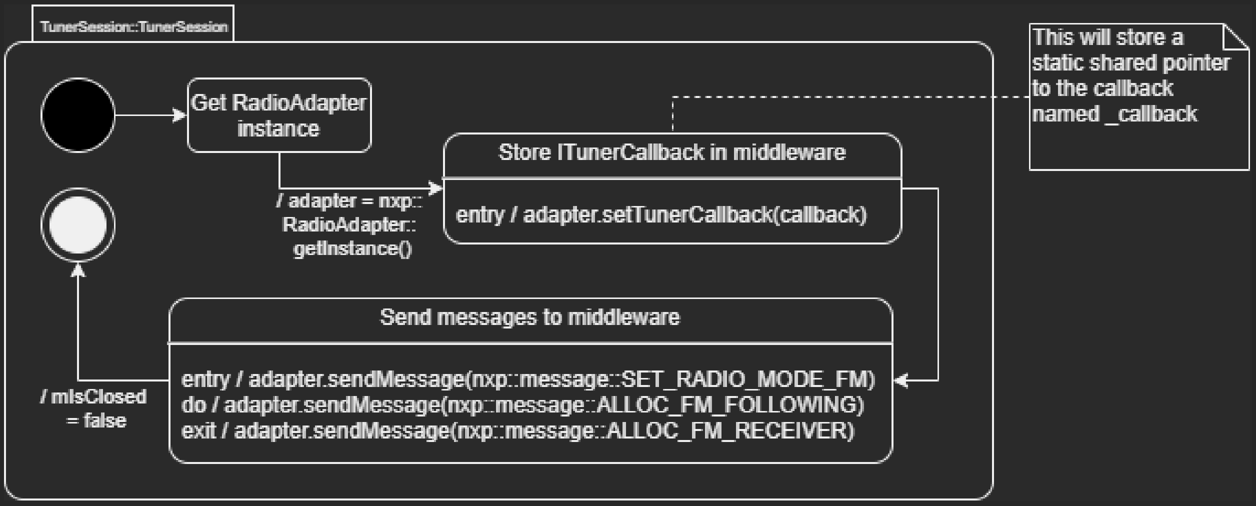
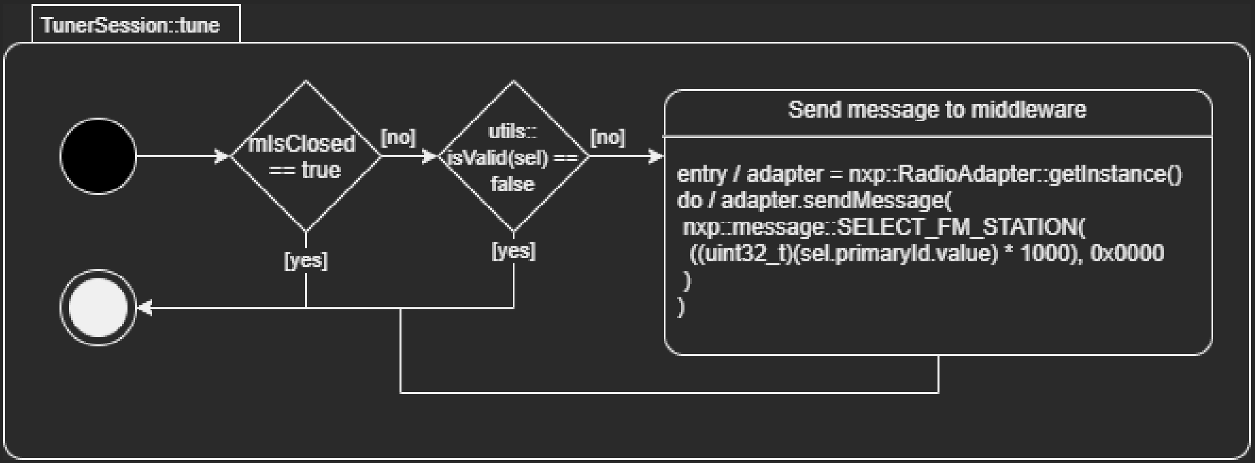
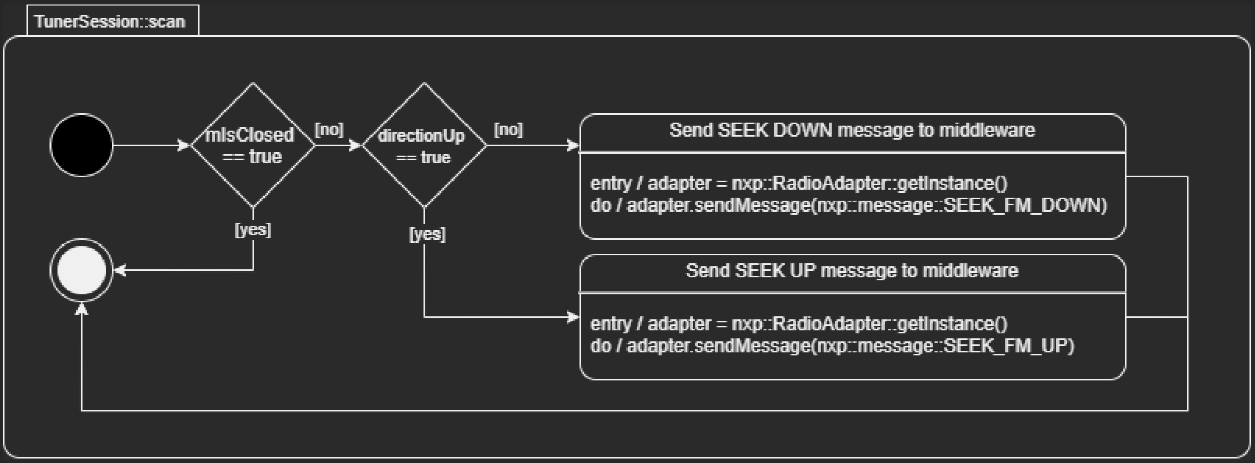

TunerSession::TunerSession
Get RadioAdapter instance
This will store a static shared pointer to the callback named _callback
Store ITunerCallback in middleware
entry / adapter.setTunerCallback(callback)
/ adapter = nxp::RadioAdapter::getInstance()
Send messages to middleware
entry / adapter.sendMessage(nxp::message::SET_RADIO_MODE_FM)
do / adapter.sendMessage(nxp::message::ALLOC_FM_FOLLOWING)
exit / adapter.sendMessage(nxp::message::ALLOC_FM_RECEIVER)
/ mIsClosed = false

Figure 15: logic of the constructor of 'TunerSession::TunerSession'



```
entry / adapter = nxp::RadioAdapter::getInstance()
do / adapter.sendMessage(
 nxp::message::SELECT_FM_STATION(
  ((uint32_t)(sel.primaryId.value) * 1000), 0x0000
 )
)
```

Figure 16: logic of the function ‘TunerSession::tune()’



```
entry / adapter = nxp::RadioAdapter::getInstance()
do / adapter.sendMessage(nxp::message::SEEK_FM_DOWN)
```

```
entry / adapter = nxp::RadioAdapter::getInstance()
do / adapter.sendMessage(nxp::message::SEEK_FM_UP)
```

Figure 17: logic of the function ‘TunerSession::scan()’ in actuality refers to a seek action but ‘scan’ is an older naming convention not to be confused with an actual background scan which sometimes also can be referred to as ‘seeking new stations’ which is not the same as a seek action which in actually means searching for a station up or below the current station.

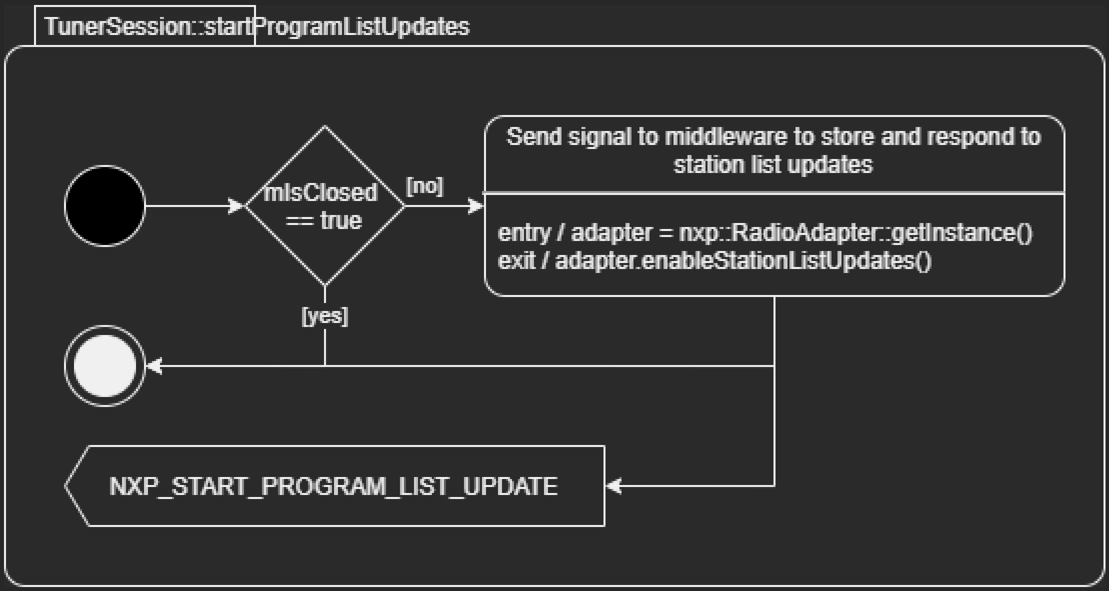

TunerSession::startProgramListUpdates
mIsClosed == true
[no]
[yes]
Send signal to middleware to store and respond to station list updates
entry / adapter = nxp::RadioAdapter::getInstance()
exit / adapter.enableStationListUpdates()
NXP_START_PROGRAM_LIST_UPDATE

Figure 18: logic of the function 'TunerSession::startProgramListUpdates()'

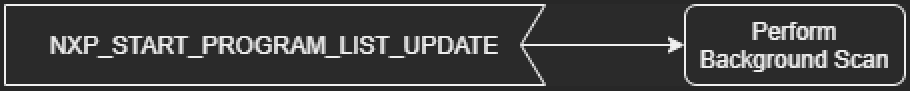


Figure 19: logic of the state 'Background scan thread'

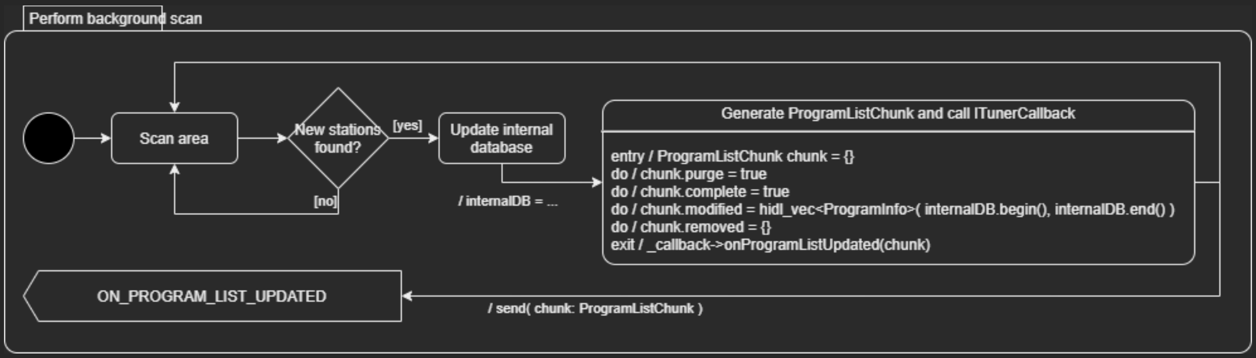

Perform background scan
Scan area
New stations found?
[yes]
[no]
Update internal database
/ internalDB = ...
Generate ProgramListChunk and call ITunerCallback
entry / ProgramListChunk chunk = {}
do / chunk.purge = true
do / chunk.complete = true
do / chunk.modified = hidl_vec<ProgramInfo>(internalDB.begin(), internalDB.end())
do / chunk.removed = {}
exit / _callback->onProgramListUpdated(chunk)
ON_PROGRAM_LIST_UPDATED
/ send(chunk: ProgramListChunk)

Figure 20: logic of the state 'Perform background scan'

Appendices

Appendices can be found in shared folder here

{1}: clean_hal_dir.zip

{2}: software_docs.zip

{3}: PlanofAction.docx