|  |  |  |
| --- | --- | --- |
| Document information | | |
| Info | | Content |
| **Project** | |  |
| **Keywords** | | BRODCAST\_RADIO\_IMPL, ANDROID\_AUTOMOTIVE\_OS, NXP\_MIDDLEWARE |
| **Abstract** | | This document will describe the inner and out workings of the L2/L3 middleware adapter layer used inside the Broadcastradio implementation of Android Automotive OS |

|  |  |  |  |
| --- | --- | --- | --- |
| Revision History | | | |
| Rev | Date | Author | Description |
| 0.1 | 04-12-2020 | Arsalan Anwari | Initial creation of document |
| 0.2 | 07-12-2020 | Arsalan Anwari | Completed document without checking by supervisors |

# Table of Contents

[Table of Contents 3](#_Toc58260322)

[1 Introduction 4](#_Toc58260323)

[1.1 Document Purpose 4](#_Toc58260324)

[1.2 Context 5](#_Toc58260325)

[2 Architecture Overview 7](#_Toc58260326)

[2.1 Approach 8](#_Toc58260327)

[2.2 Subsystem/Component break down 8](#_Toc58260328)

[2.3 Communication mechanism 11](#_Toc58260329)

[3 Architecture Details 13](#_Toc58260330)

[3.1 Execution workflow 15](#_Toc58260331)

[3.2 Function logic 16](#_Toc58260332)

# Introduction

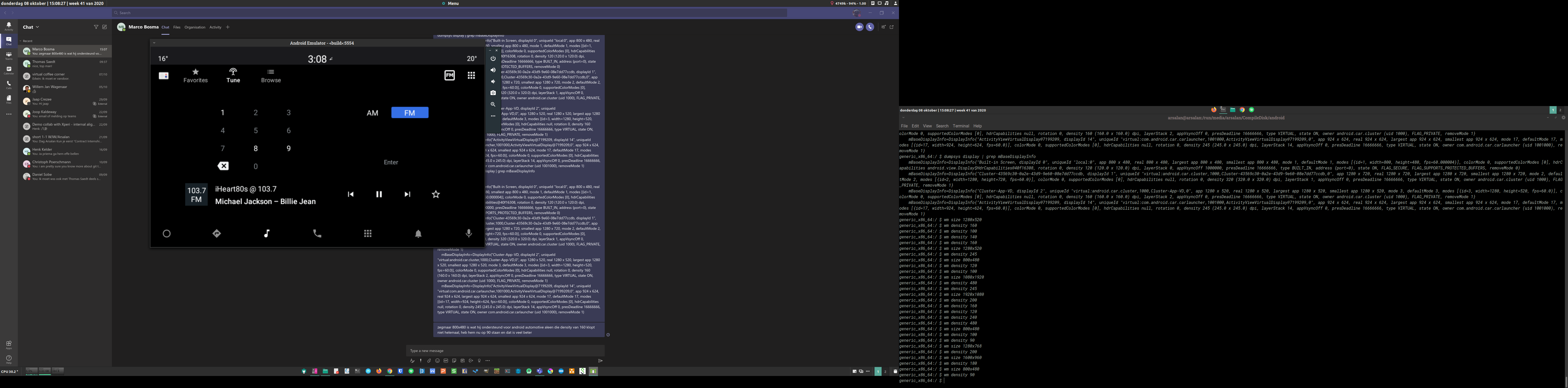
Android Automotive OS is a framework which allows embedded software developers of infotainment systems to generalize their driver level code with a reusable API which can be used by Android application developers in the service level software (for example code regarding the Human Machine Interface). Android Automotive OS is an extension of the existing Android code base named ‘AOSP’ which contain a number of Hardware Abstraction Layers to be used by the embedded software developers. This document only covers one of these HAL’s responsible for digital/analogue radio communication named ‘Broadcastradio’. Here an external Android application can use the functionality of this HAL for service level functionality. Chapter [2](#_Architecture_Overview) and [3](#_Architecture_Details) will explain how this Broadcastradio HAL has been implemented with the L2/L3 middleware of NXP to communicate with the radio hardware on a IMX8xx board. SysML and UML is used as the notation standard. SysML is used describe the ‘system’ in a generic manner whilst UML is used for software and logic specific descriptions.

## Document Purpose

This document provides knowledge about the internal and external communication between the two main pillars that make up the Broadcastradio HAL implementation of Android Automotive OS, namely:

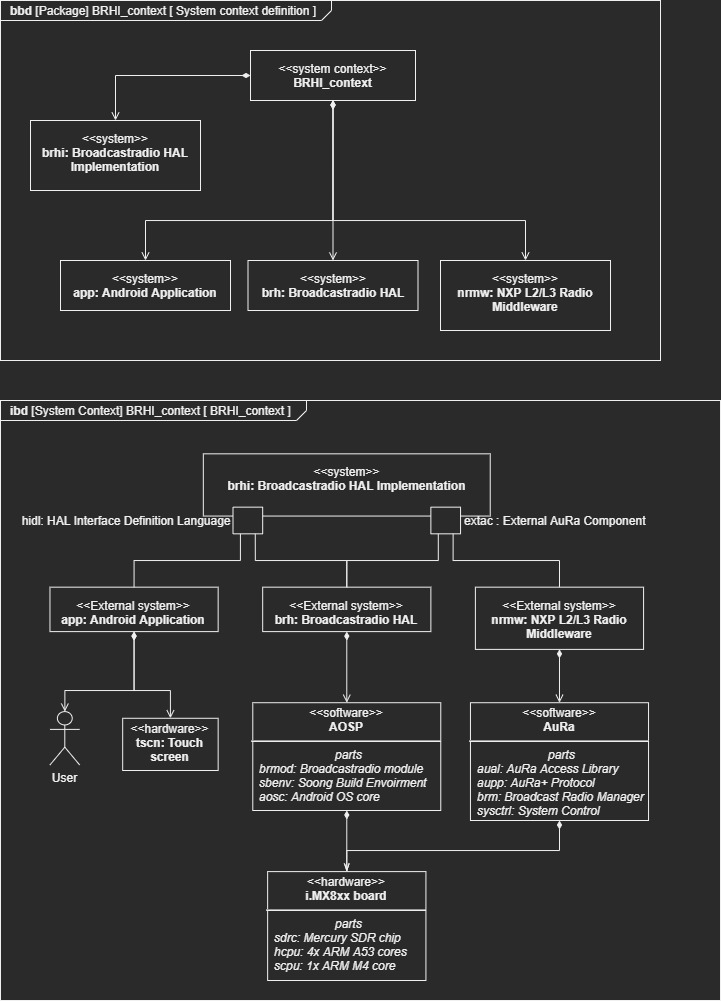
* The communication between the external Android application and the Broadcastradio HAL
* The communication between the Broadcastradio HAL and L2/L3 middleware

This document will only cover the Broadcastradio HAL implementation with analogue FM radio functionality of the L2/L3 middleware. A later revision of this document will cover the remaining radio functionality like digital DAB+, HD Radio and more. By default Android delivers a radio Android Application which you can use to test and debug your Broadcastradio HAL implementation. It is possible to remove this default application and replace it with one of your own, but this will not be discussed in the scope of this document. The default radio application can be seen in figure 1. I will also not go into detail how your test Broadcastradio HAL implementation in the Android Emulator and physical hardware as this document only describes the software architecture of the Broadcastradio HAL implementation and should be used as a guide.



*Figure 1: Default radio application by Google*

## Context



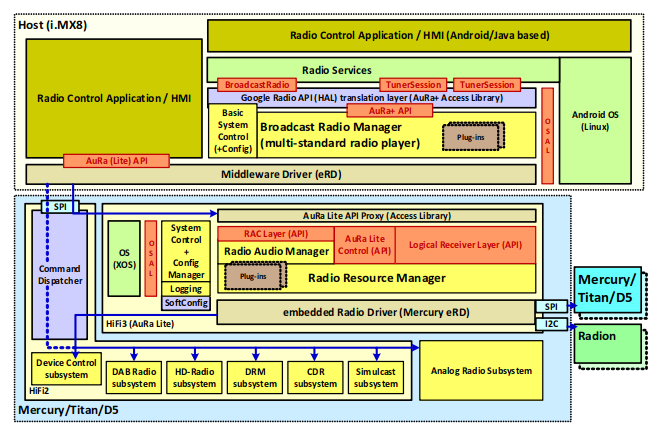
*Figure 2: System Context diagram*

In figure 2 you can see the System context of the Broadcastradio HAL Implementation. Here the user interacts with an Android Application which can access C++ logic of the Broadcastradio HAL through HIDL at compile time. This C++ logic then internally contains code which communicates with the L2/L3 middleware through an external AuRa component. The C++ logic of the Broadcastradio HAL is compiled with the Soong build environment and needs some additional files contained inside the Broadcastradio module and the main Android OS code.

The compilation cycle is bottom up, meaning first the L2/L3 middleware is compiled, then the glue logic of the RadioAdapter class, and lastly the main logic of the Broadcastradio HAL. The C++ logic of the Broadcastradio HAL is compiled as a background service where the Android application can access this logic through a pre-defined API included in the HIDL files.

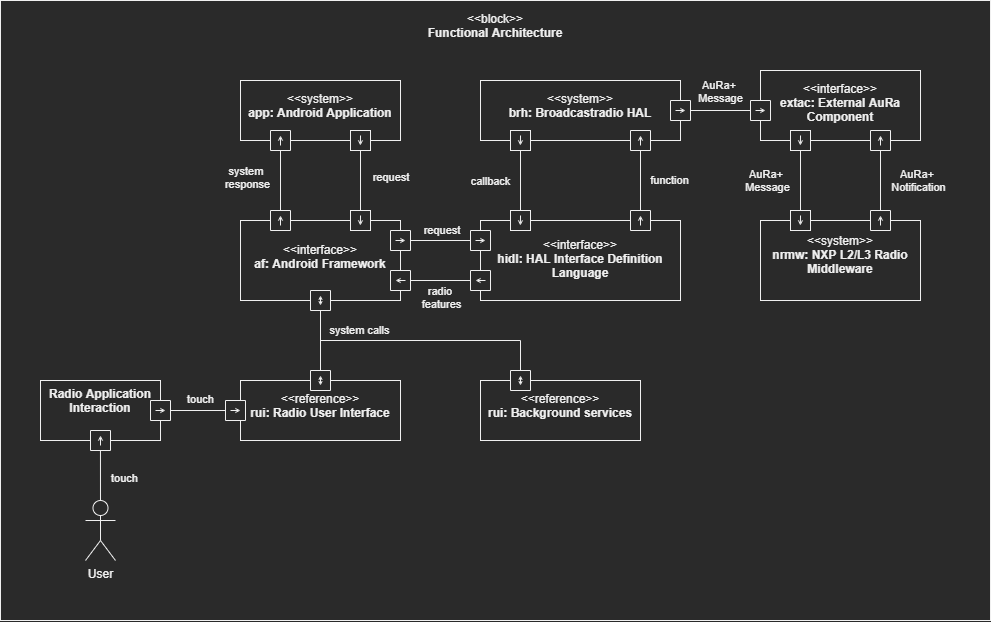
There a 4 parts of the L2/L3 middleware software that need to be used in order to communicate with the radio hardware of the Mercury SDR chip. The L2/L3 middleware consist of different AuRa components. The BroadcastRadioManager AuRa component communicates with the radio hardware. The SystemControl AuRa component can register new AuRa components, initialize them and create QueuePairs between AuRa components. The AuRa Access library is a software library used to handle messages send between AuRa components and the messages are formatted in the AuRa+ protocol.

The Android application is hardware independent and can run on anything that supports Android OS. The Broadcastradio HAL underneath Android OS will be ran on the 4 ARM A53 cores. The L2/L3 middleware however will be running on the Mercury SDR chip where the ARM M4 core is used as a bridge to communicate with the host OS ( in this case Android OS ). See figure 3.



*Figure 3: High level overview of communication between host CPU and Mercury SDR chip.*

# Architecture Overview



*Figure 4: Functional Architecture*

In figure 4 you can see the Functional Architecture of the Broadcastradio HAL implementation. With touch user can interact with the Radio User Interface according to use cases defined in the Radio Application interaction use case diagram (see [chapter 2.3](#_External_communication_mechanism)). The Radio User Interface transceives system calls with the Android Framework which in turn handle majority of the communication between additional background services and the Broadcastradio HAL. The Android application requires access to additional background services to handle things like loading and removing files which will not be discussed in this document.

To communicate with the Broadcastradio HAL the Android Framework uses HIDL files[[1]](#footnote-2) [[2]](#footnote-3) stored in the Broadcastradio module which specifies what classes and functions can be accessed from the Broadcastradio HAL logic by the Android application and how these are declared.

The Broadcastradio HAL communicates with the L2/L3 middleware through an external component

## Approach

The two main requirement of the written software in the Broadcastradio HAL implementation were reusability and simplicity.

* Reusability means any code written for the definition of the Broadcastradio HAL classes and functions should not require changing the original declaration of the classes found in the ‘.h’ files.
* Simplicity means that any manual resource allocation and memory management for the communication with the external AuRa component should be reduced.

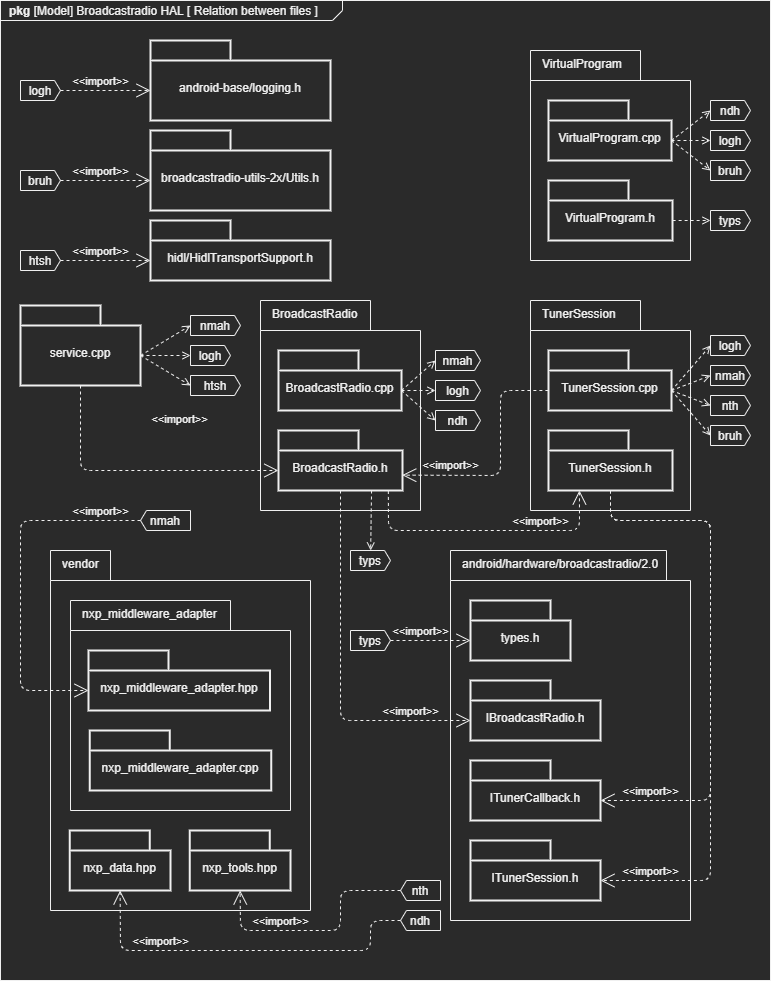
To satisfy both requirements a ‘Singleton’ class is used named ‘RadioAdapter’ which handles things like initializing the external AuRa component, accessing radio features and more. This design pattern is quite controversial and often associated with a “bad software practice”. Although this claim is true 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.

## Subsystem/Component break down

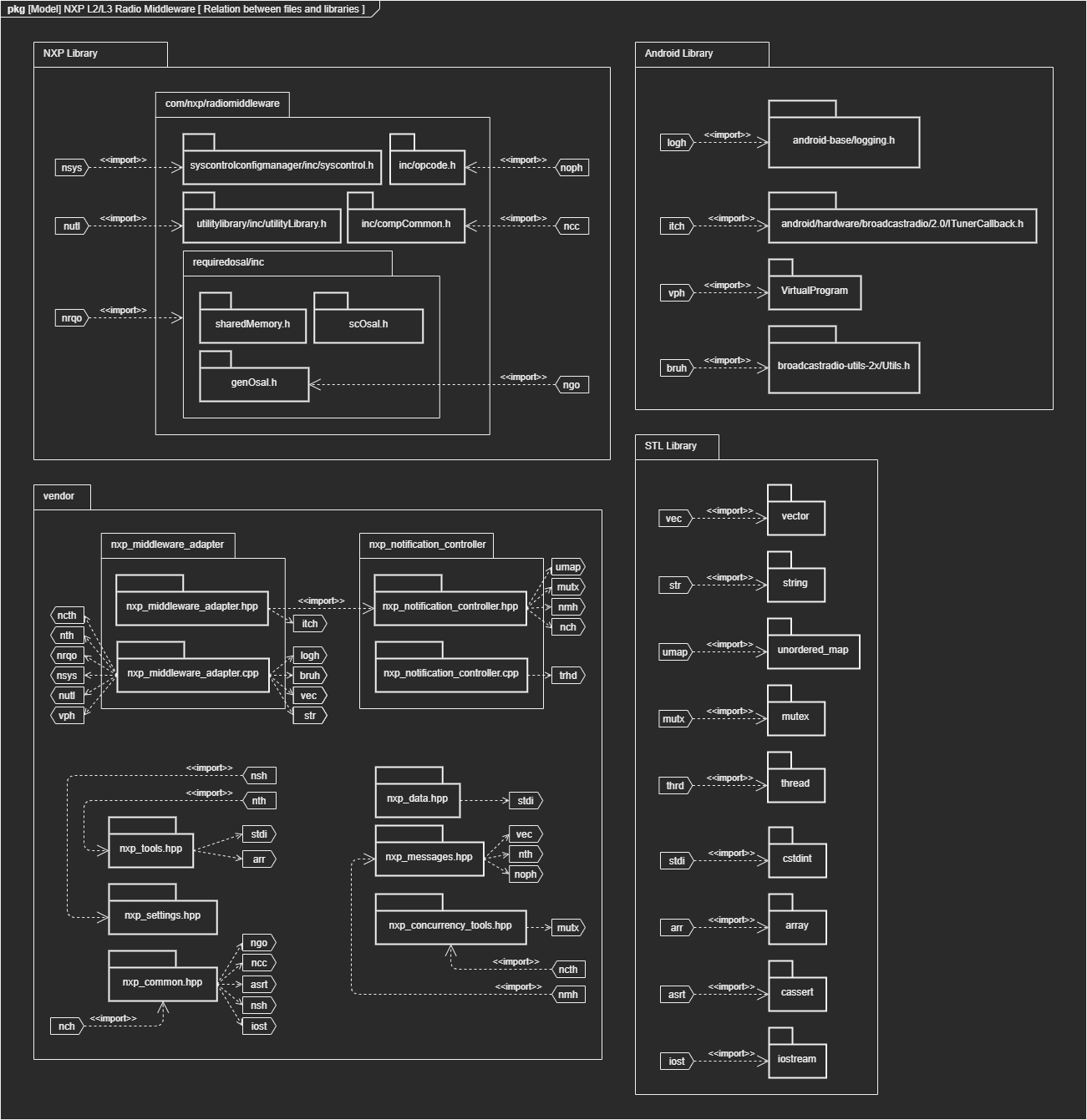
This document will not cover the component breakdown of the Android Application system as this can differ for each different Android application made by the OEM. The package diagrams of the Broadcastradio HAL and External AuRa Component can be found in the two figures below. These package diagrams describe the internal relationship and dependencies of the source files. Here a package can either be:

* A folder or subfolder (example: the package ‘nxp\_data.hpp’ represent the file ‘nxp\_data.hpp’ which resides in the location ‘vendor/nxp\_data.hpp’) .
* An alias for the full file path (example: the file ‘android/hardware/broadcastradio/2.0/types.h’ referred as the package ‘types.h’ in the alias ‘android/hardware/broadcastradio/2.0’) .
* A group of a source and header file where the source file includes the header file (example: the package ‘TunerSession.cpp’ always imports ‘TunerSession.h’, but this does not have to explicitly mentioned as they are contained under the same ‘TunerSession’ group) .
* A source of header file (example: the package ‘service.cpp’) .

Each SysML package contains multiple UML diagrams like class diagrams and concurrency diagrams but can also contain other SySML diagrams like a state machine diagram. These additional diagrams can be found in [chapter 3](#_Architecture_Details)



*Figure 5: Package diagram of Broadcastradio HAL*



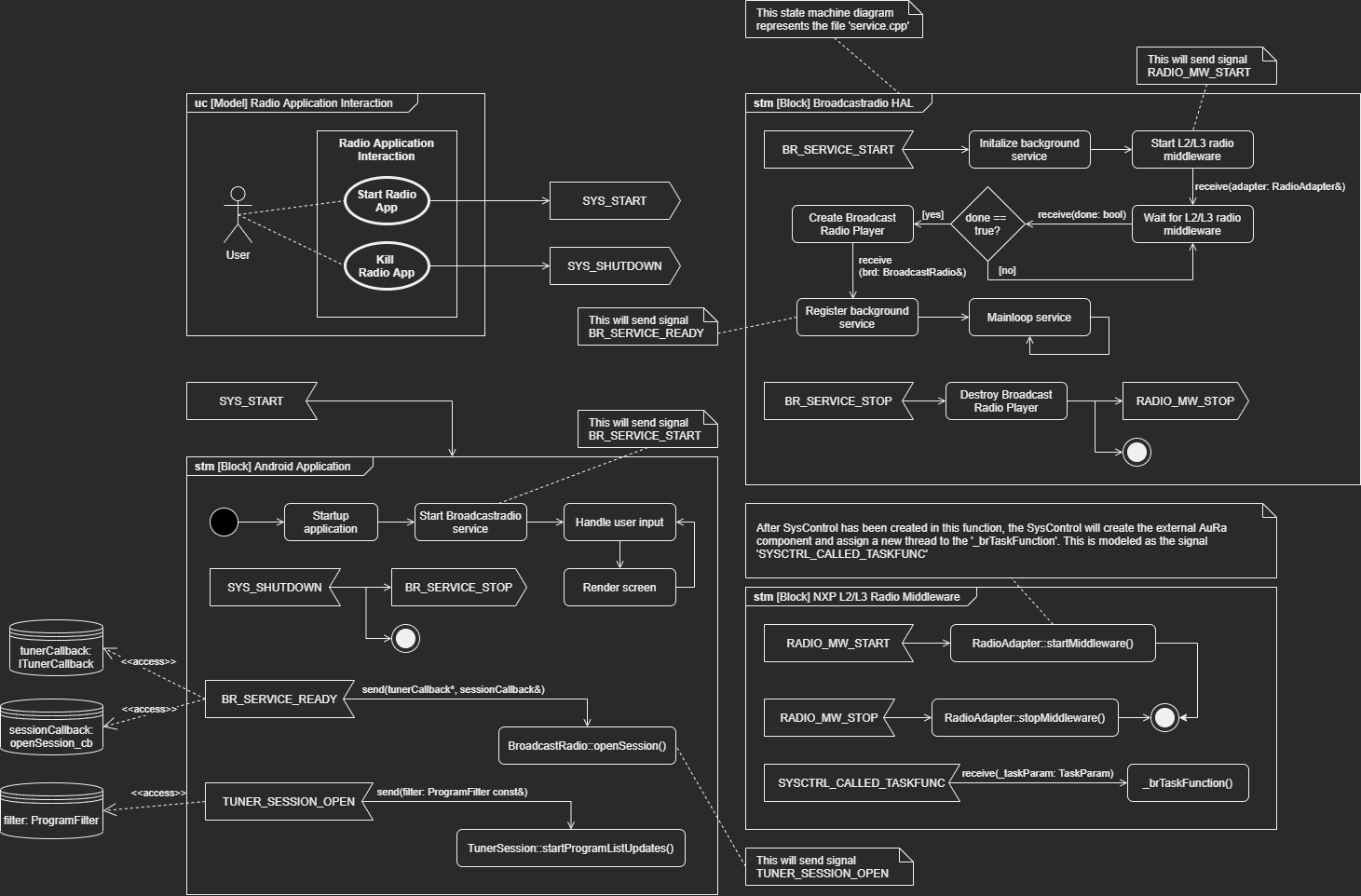
*Figure 6: Package diagram of NXP L2/L3 Radio Middleware*

## Communication mechanism



*Figure 7: Use case diagram of Radio Application Interaction*

In figure 7 you can see a list of actions the user can take in the user interface of the Android application represented as use cases. The Broadcastradio HAL and NXP L2/L3 Radio Middleware is depended on the actions taken by set user as in reality the radio application will just directly call functions of the Broadcastradio HAL through the interface specified by the HIDL files. As this radio application can/will be changed by the Android app development team in the final product I will not be able to describe the external or internal logic, but I will describe what parts of the Broadcastradio HAL could be called in a representation.



*Figure 8: State machine diagram of communications between three main systems*

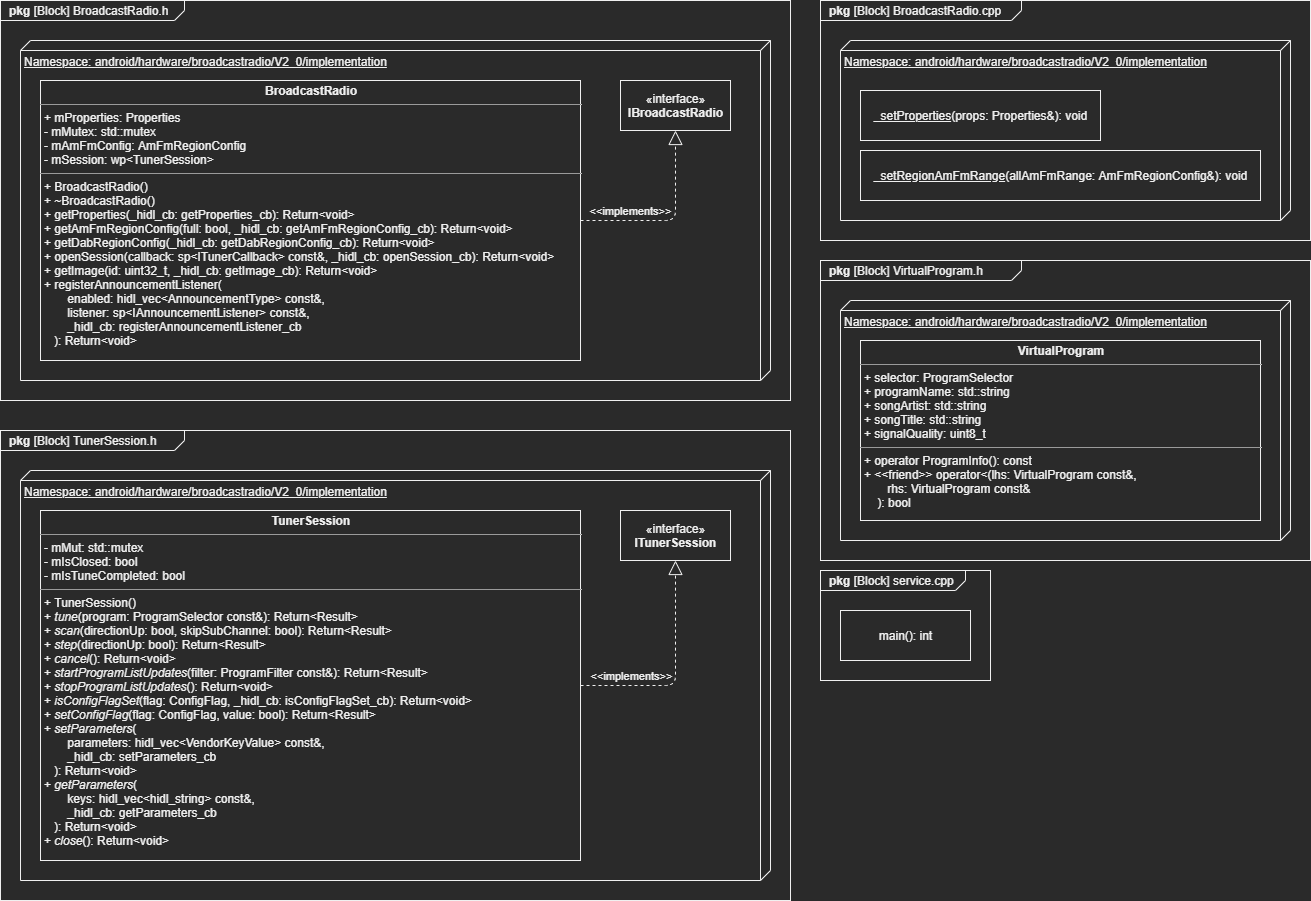
In figure 8 you can see the state machine diagram of the external communication of the three main systems. Like mentioned in previous chapters, the state machine of the Android Application is a representation not the actual logic as this is depended on the Android app development team. Each state in these state machine diagrams can either:

* Be a substate shown in detail in [chapter 3.1](#_Execution_workflow).
* Or be a (member)function based on the structure of the class diagrams shown in figure 8 and 9 where the internal logic is represented as a substate shown in detail in [chapter 3.2](#_Function_logic).

Some state in these state machine diagrams are a ‘black box’ as their internal logic is depended on the Android app development team, the Android Framework or another 3th party. For example the state ‘Render screen’ in the state machine diagram ‘stm [Block] Android Application’ is such a black box.

# Architecture Details

Blocks with the text ‘<name>: <type>’ can be seen a constant variable stored inside the file. Blocks with the text ‘<name>(<name>: <type>, …): <type>’ can be seen a functions with parameters stored inside the file. Blocks with the text ‘<name> -> <type>’ can be seen as a “typedef”.

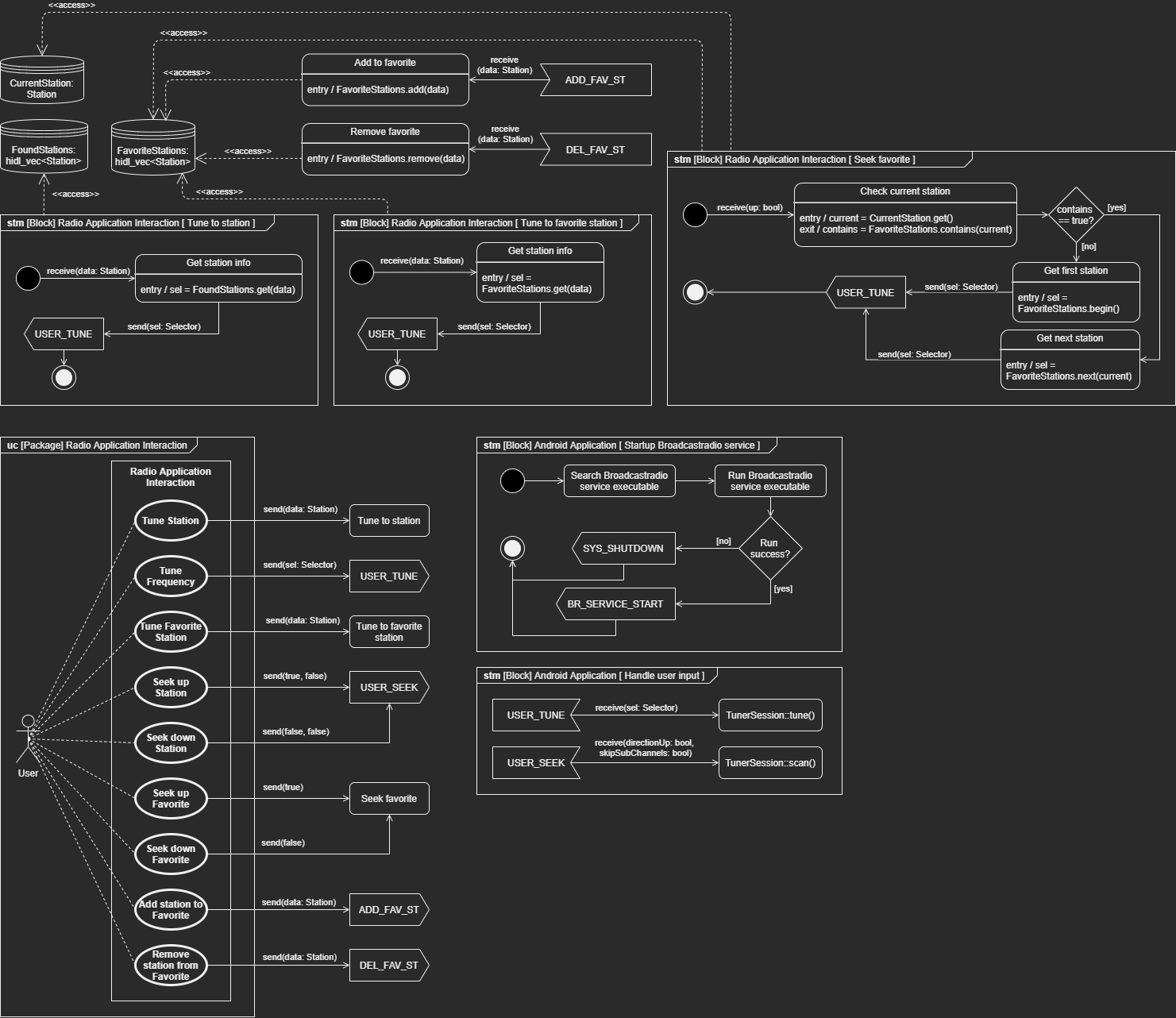


*Figure 9: Class diagram of Broadcastradio HAL*

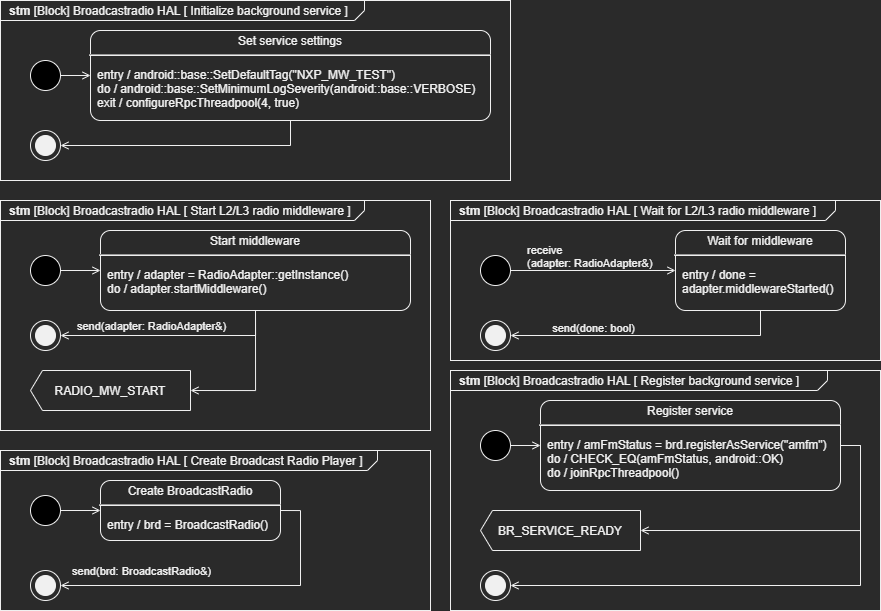


*Figure 10: Class diagram of NXP L2/L3 Radio Middleware*

## Execution workflow



*Figure 11: State machine diagram of Android Application*

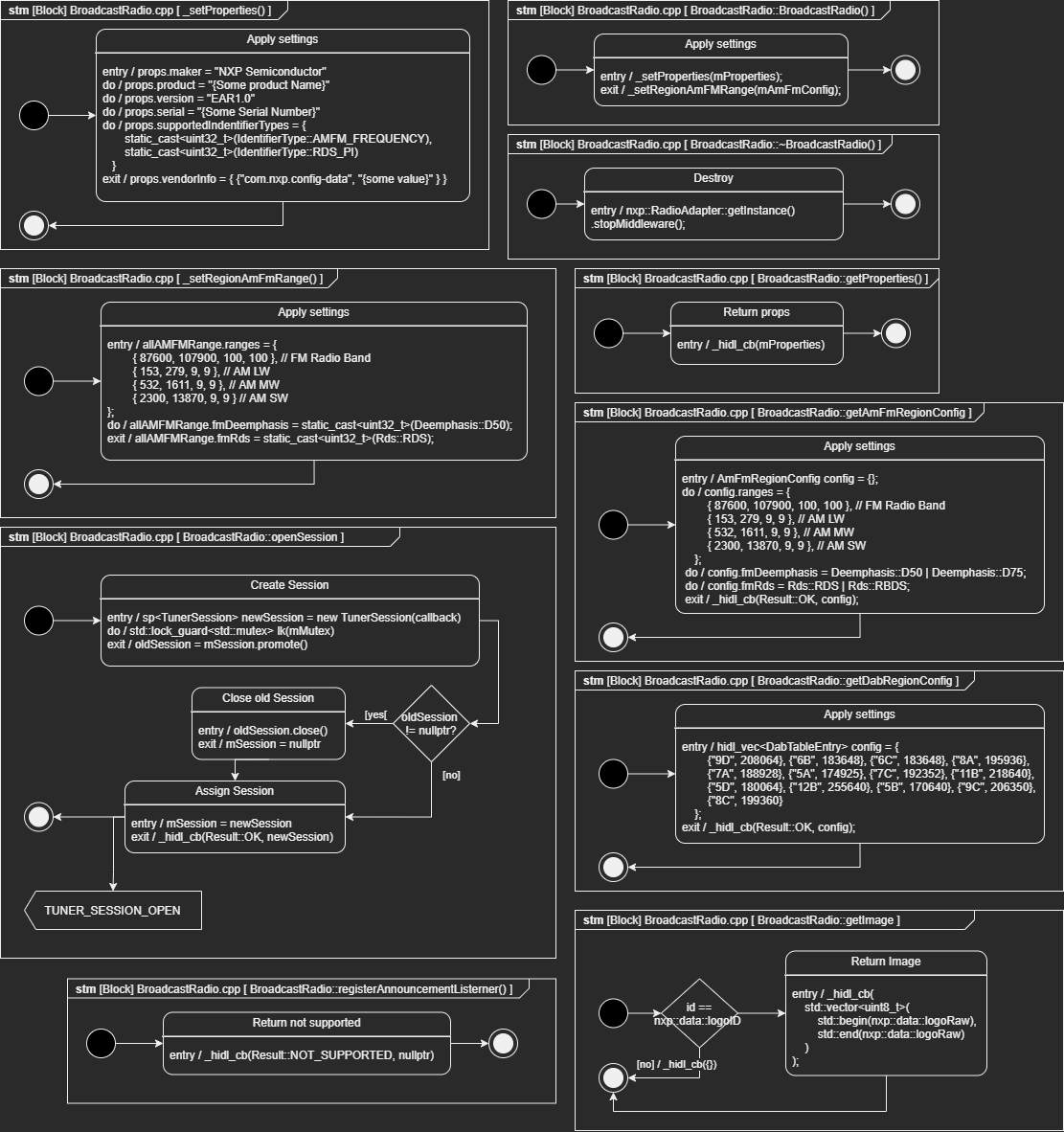


*Figure 12: State machine diagram of Broadcastradio HAL*

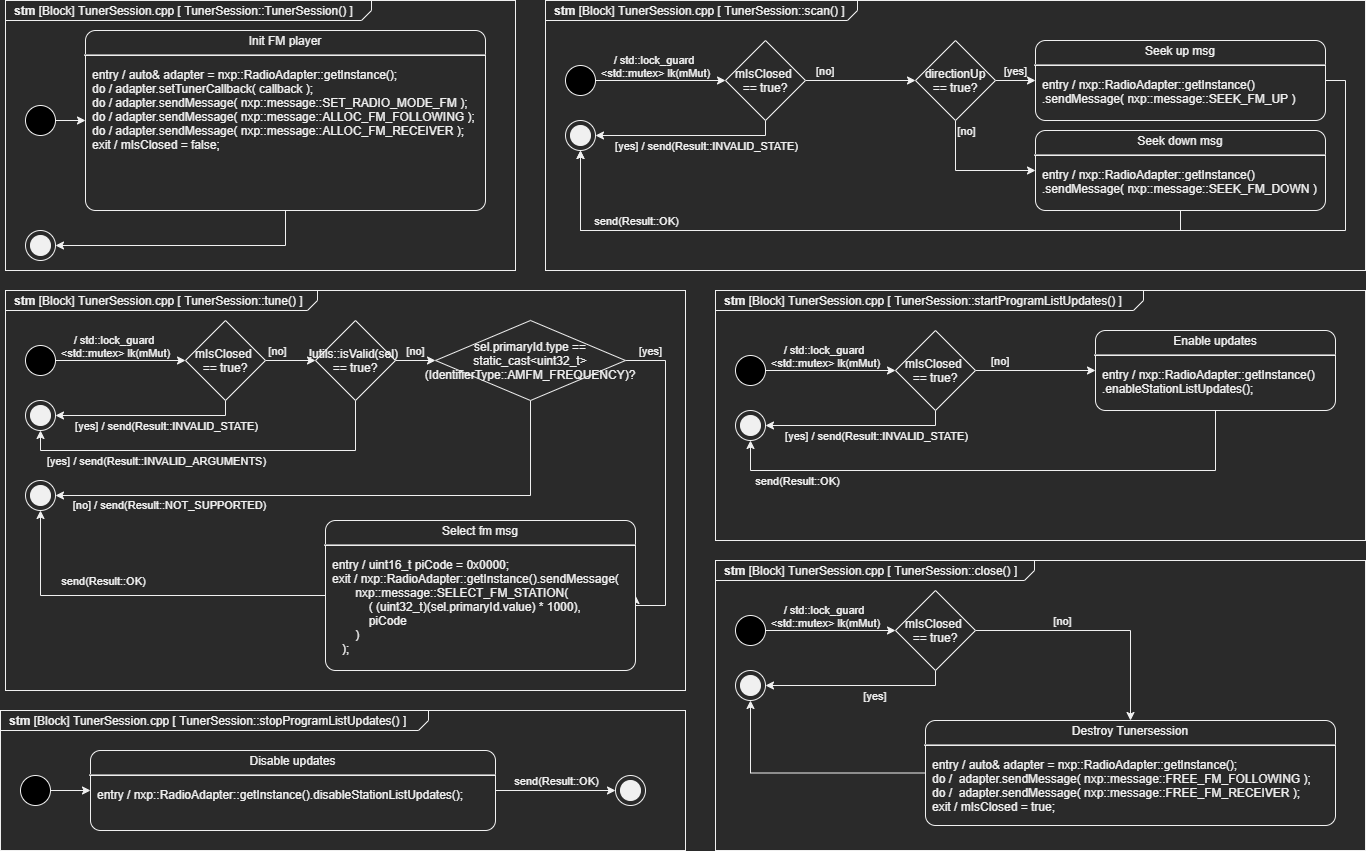
As you can see in figures 11 and 12, not all (member)functions of the BroadcastRadio and TunerSession class are used in these state machine diagrams. This is because some of these functions can be used at different times or not at all depending on how the Android app development team has written the Android application. For example an Android app developer could chose to call the function ‘BroadcastRadio ::getProperties()’ before calling ‘BroadcastRadio::openSession()’, or not call it at all. Same story goes for any (member)function of the TunerSession Class. The only thing you can be entirely sure about is that the Android app developer needs to call the functions ‘BroadcastRadio::openSession()’ and ‘TunerSession::startProgram ListUpdates()’ before he can do anything sensible in the Android application. It can be assumed that the Android app developer will pass a callback function when the function ‘BroadcastRadio::openSession()’ is called where he then call the function ‘TunerSession::startProgramListUpdates()’ inside the callback function. This is represented in the state machine diagram ‘stm [Block] Android Application’ as the signals ‘BR\_SERVICE\_READY’ and ‘TUNER\_SESSION\_OPEN’ in figure 8.

## Function logic

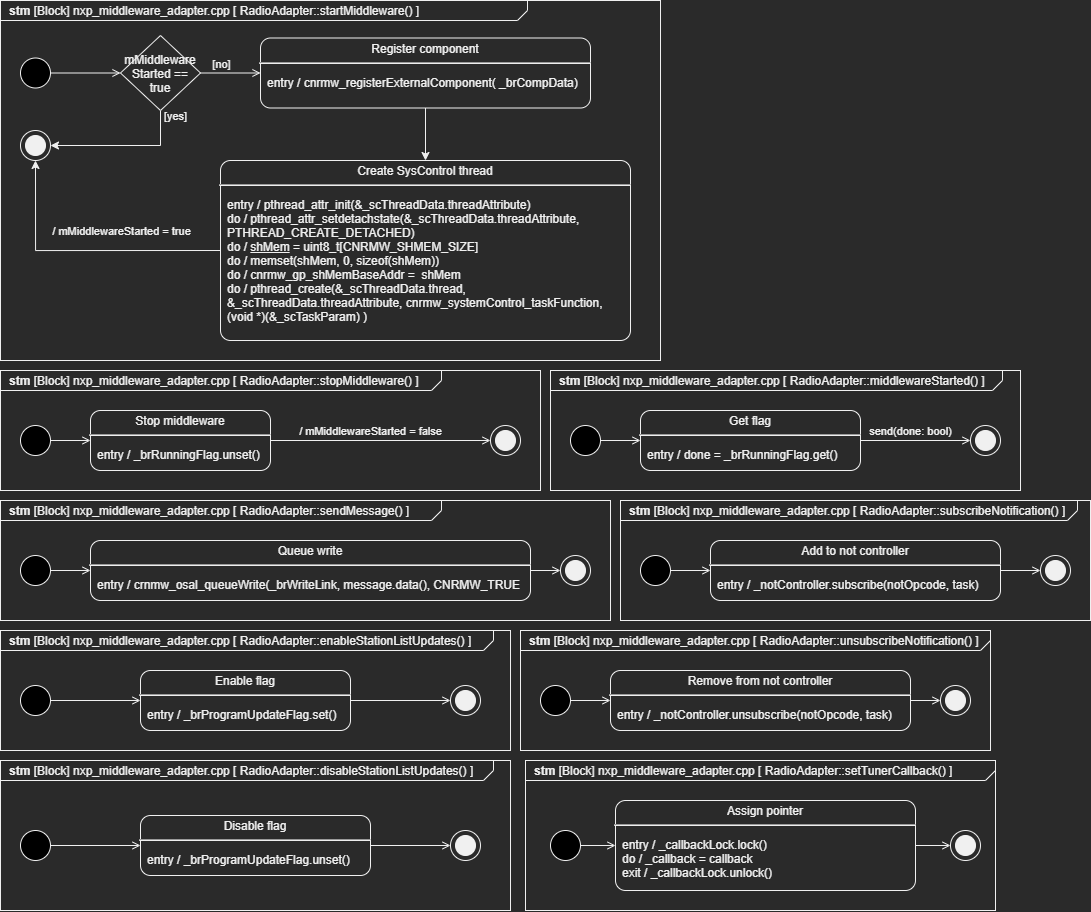
Not all functions of the class diagrams in figure 9 and 10 are modeled into state machine diagrams below. This is because some functions like ‘tools:split16b()’ are pretty straight forwards or some functions are declared with no code inside them (basically meaning they are unused). Therefor only the most important functions are modelled. Most of these functions that are not modeled in the figures below are also very simplistic with descriptive names that already describe what you can expect what will happen.



*Figure 13: State machine diagram of BroadcastRadio functions*



*Figure 14: State machine diagram of TunerSession functions*



*Figure 15: State machine diagram of RadioAdapter functions*

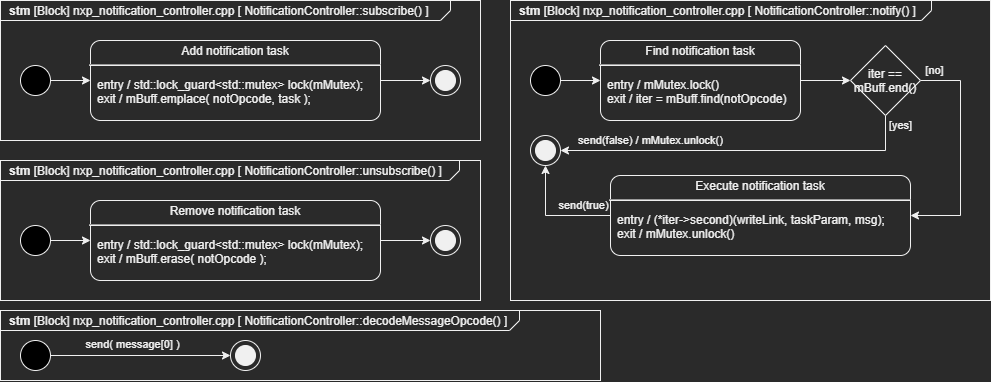
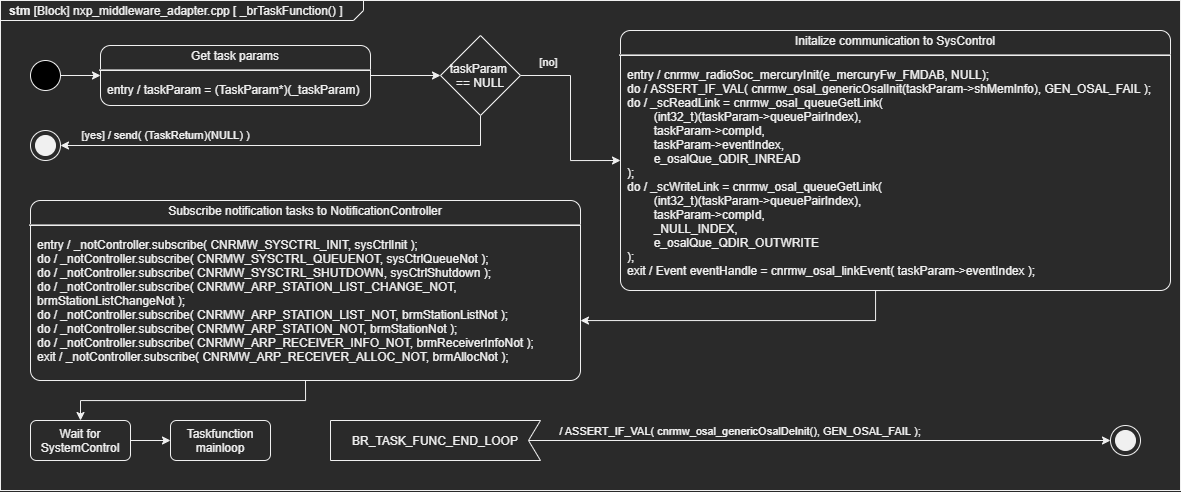
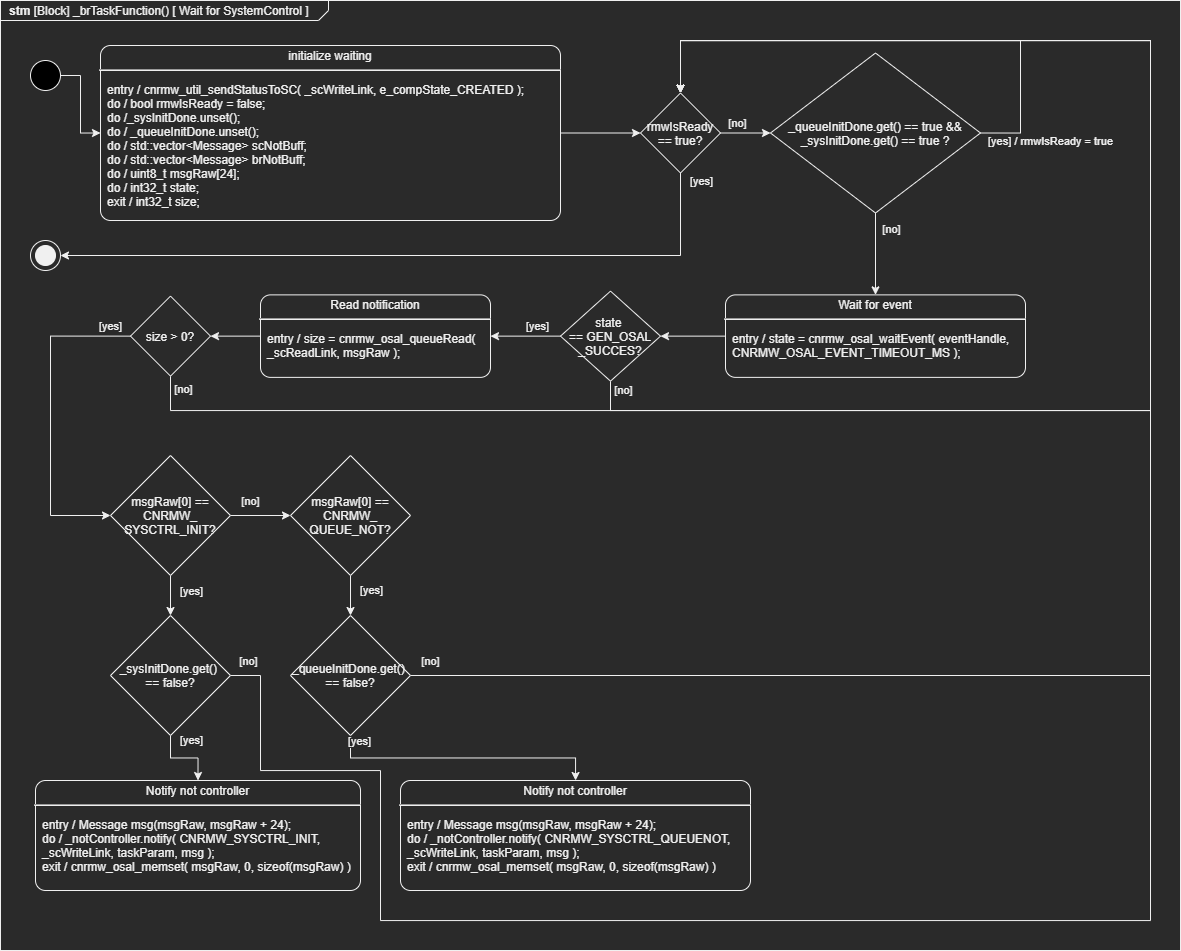


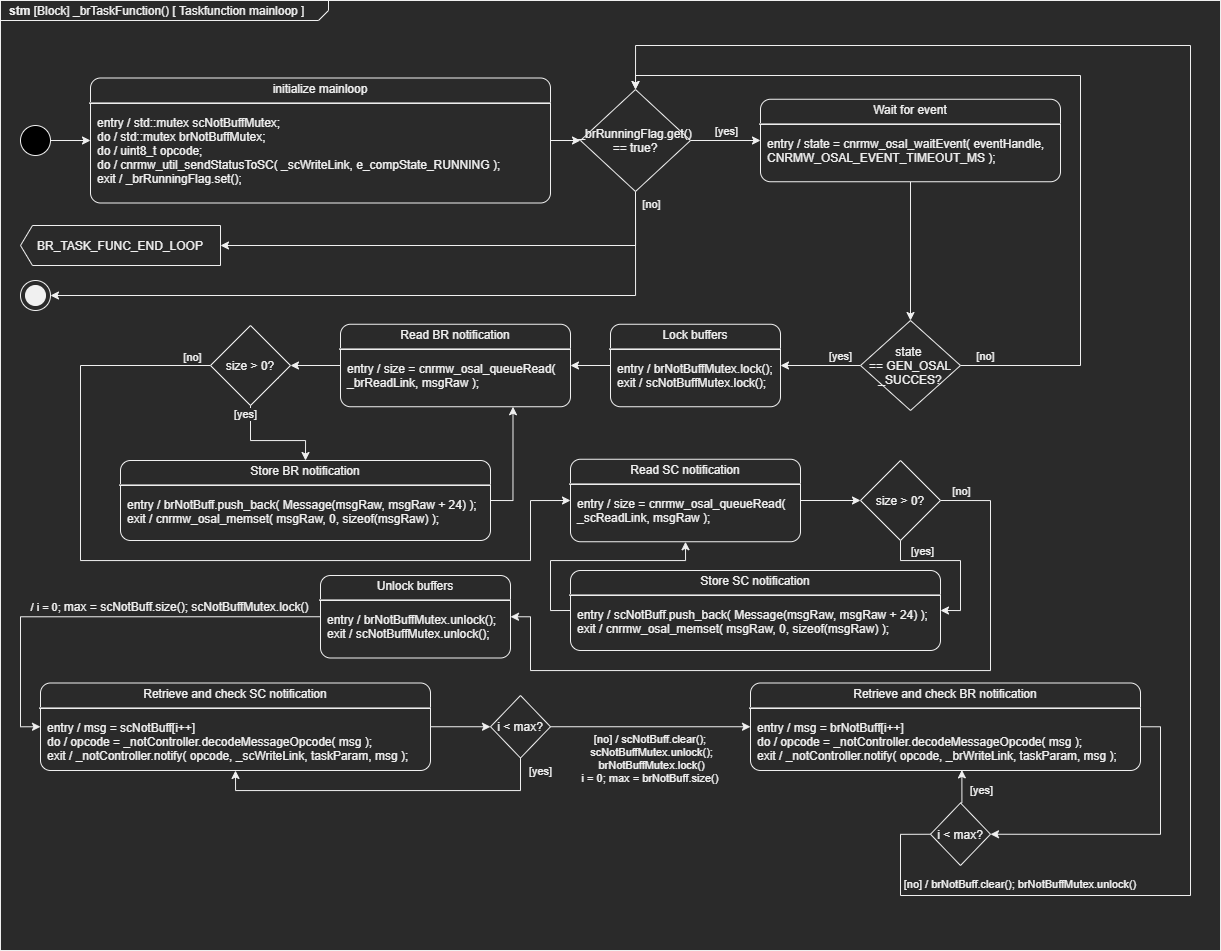
Figure 16: State machine diagram of NotificationController functions



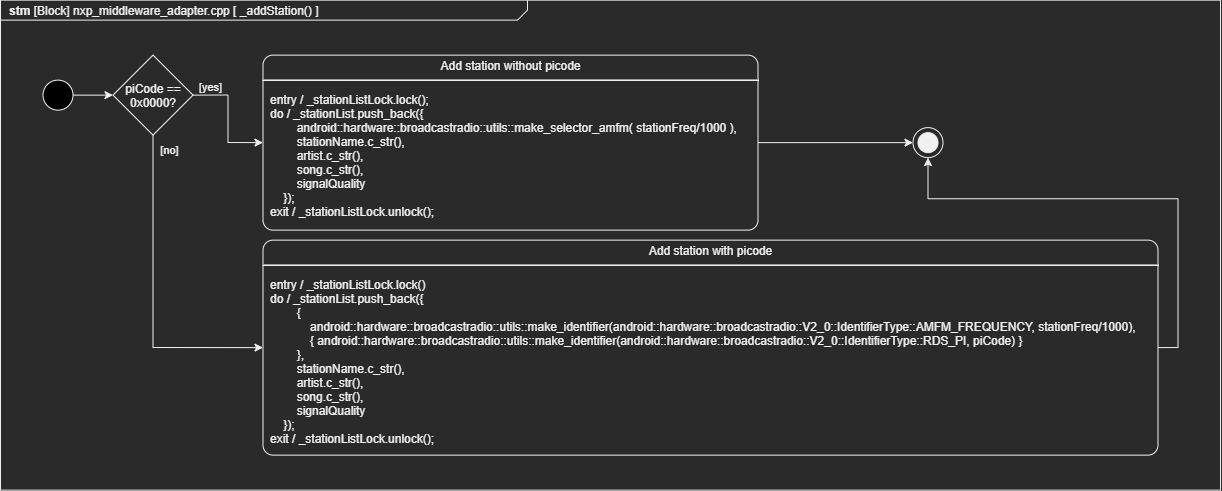
*Figure 17: State machine diagram of overview of \_brTaskFunction() (a.k.a. functions connected to AuRa component)*



*Figure 18: State machine diagram of sub state ‘Wait for SystemControl’ of ‘\_brTaskFunction()’*



*Figure 19: State machine diagram of sub state ‘Taskfunction mainloop’ of ‘\_brTaskFunction()’*

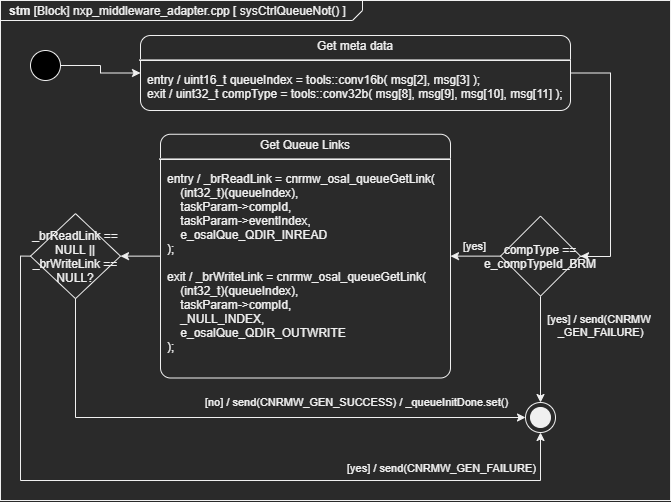


*Figure 20: State machine diagram of \_addStation()*

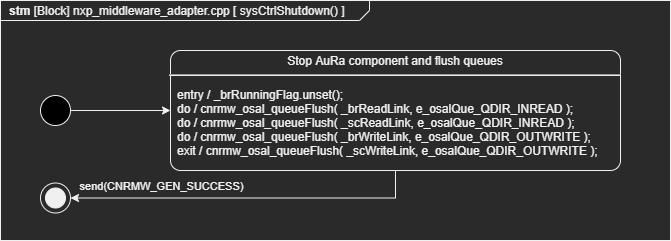
In figure 17 in the substate ‘Subscribe notification tasks to NotificationController’ you can see a series of function pointers being added to a list. In the actual code these function pointers are “lambda functions” stored inside a variable. The following figures below represent the logic inside these function pointers.



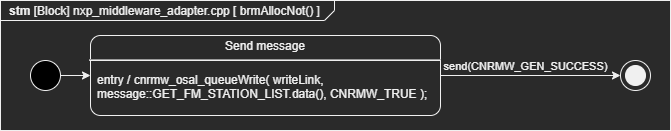
*Figure 21: State machine diagram of function pointer ‘sysCtrlInit’*



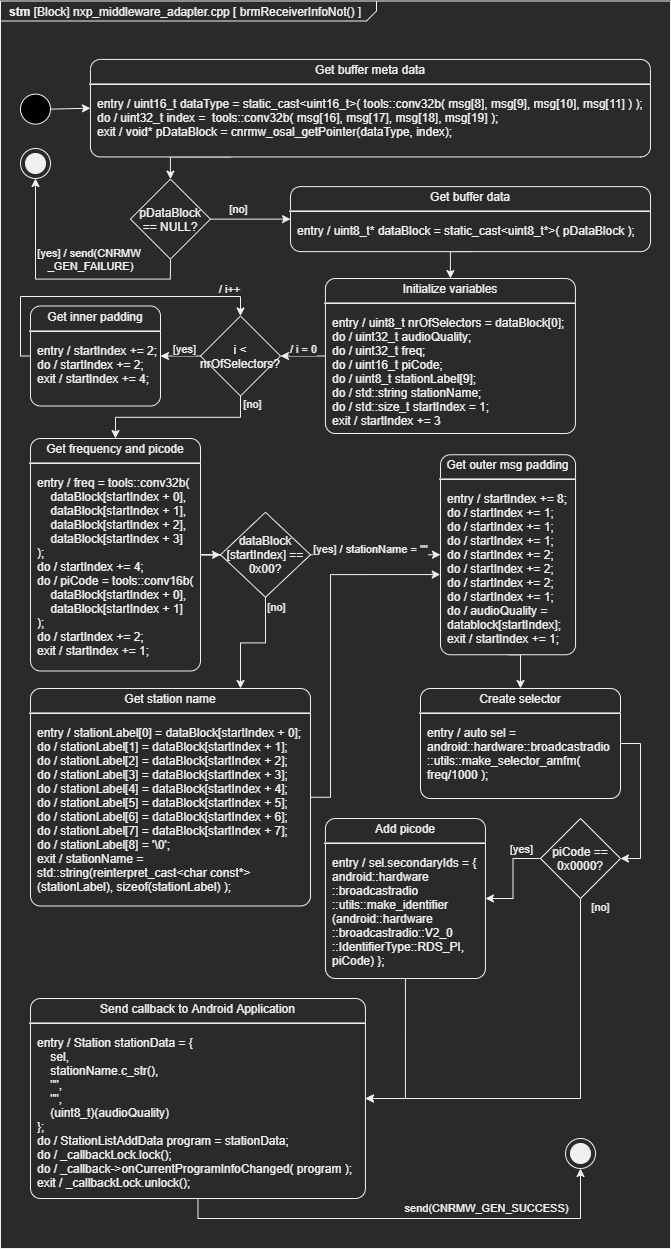
*Figure 22: State machine diagram of function pointer ‘sysCtrlQueueNot’*



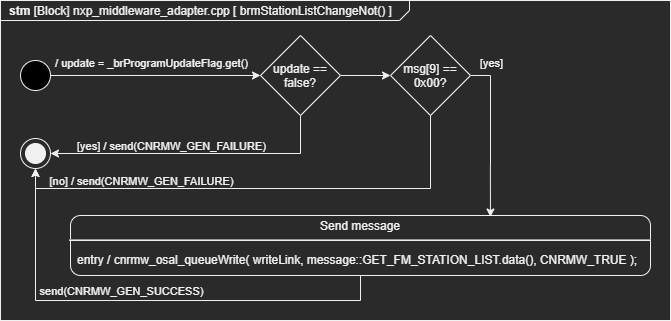
*Figure 23: State machine diagram of function pointer ‘sysCtrlShutdown’*



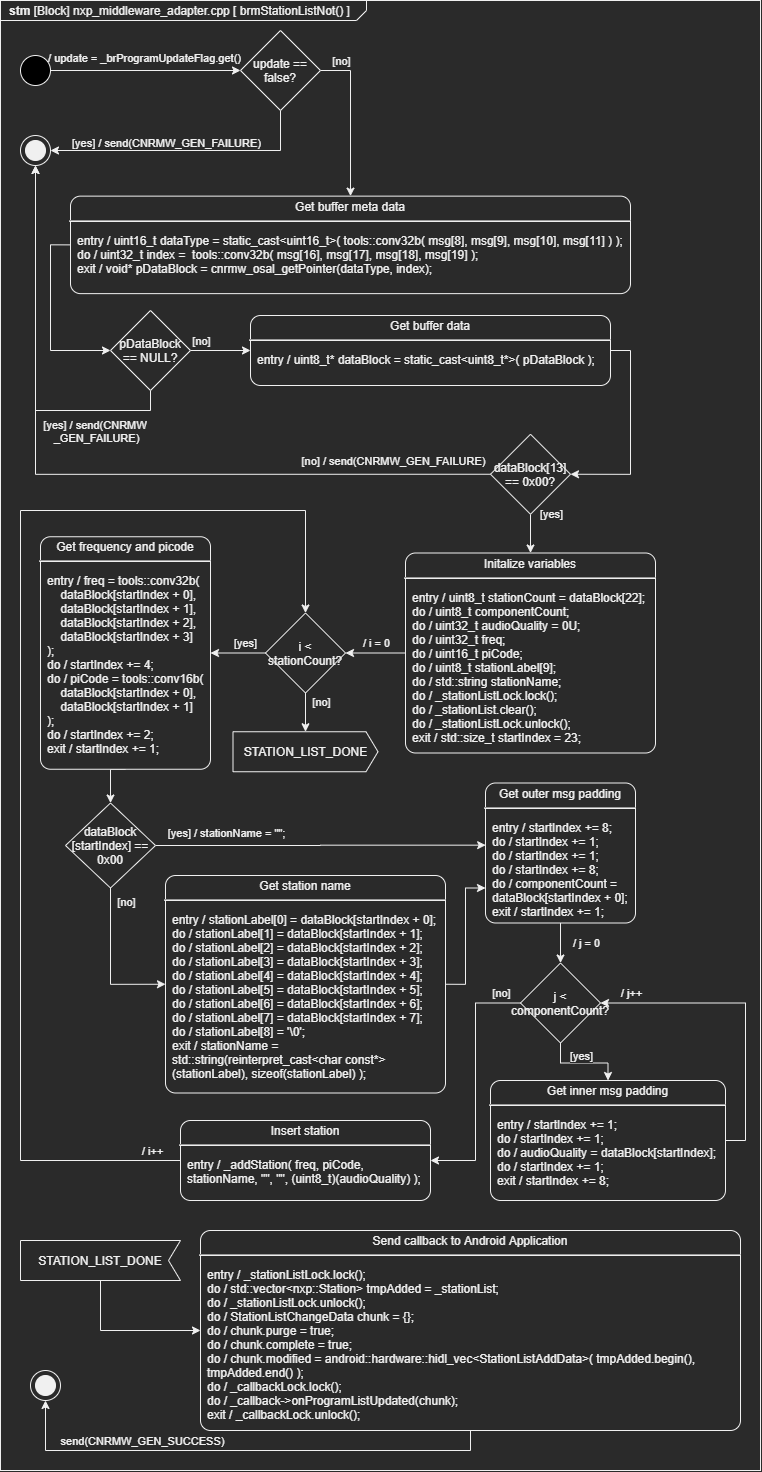
*Figure 24: State machine diagram of function pointer ‘brmAllocNot’*



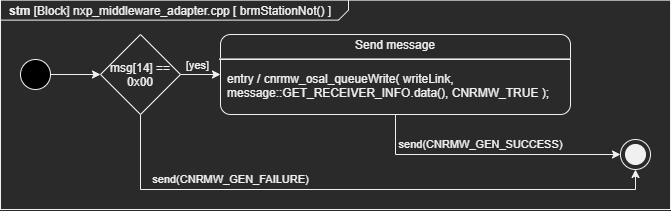
*Figure 25: State machine diagram of function pointer ‘brmReceiverInfoNot’*



*Figure 26: State machine diagram of function pointer ‘brmStationListChangeNot’*



*Figure 27: State machine diagram of function pointer ‘brmStationListNot’*



*Figure 27: State machine diagram of function pointer ‘brmStationNot’*

You can use the RadioAdapter to subscribe to notifications send by the SystemControl and BroadcastRadioManager AuRa components by calling the function RadioAdapter::subscribeNotification(). Here you can supply a function pointer to a lambda or static function you have declared somewhere. Please keep in mind that your function should have a valid structure specified by the ‘NotificationTask’ datatype. You can only subscribe one function to one notification and you cannot subscribe to the following notifications as these are already in used by the RadioAdapter class:

* CNRMW\_SYSCTRL\_INIT
* CNRMW\_SYSCTRL\_QUEUENOT
* CNRMW\_SYSCTRL\_SHUTDOWN
* CNRMW\_ARP\_STATION\_LIST\_CHANGE\_NOT
* CNRMW\_ARP\_STATION\_LIST\_NOT
* CNRMW\_ARP\_STATION\_NOT
* CNRMW\_ARP\_RECEIVER\_INFO\_NOT
* CNRMW\_ARP\_RECEIVER\_ALLOC\_NOT

The RadioAdapter class should be the only class you interact with if you want to expand upon this code, as this is the only class that has access to the External AuRa component. A suggestions has been made by Marco Bosma to change the RadioAdapter to a proxy pattern, but this is still in debate.

1. More information about HIDL files can be found here: <https://source.android.com/devices/architecture/hidl> [↑](#footnote-ref-2)
2. The HIDL files of the Broadcastradio module can be found here as ‘.hal’ files:

   <https://android.googlesource.com/platform/hardware/interfaces/+/master/broadcastradio/2.0> [↑](#footnote-ref-3)