# Abstract

The document discusses the Modularized Radio Application reference architecture goals and presents it the in several views: functional, concurrency, deployment and development view. It also briefly describes the Radio Application’s role and placement in the Base Station system.

Contents

[1 Introduction 3](#_Toc38649708)

[1.1 Revision history 3](#_Toc38649709)

[1.2 Purpose 6](#_Toc38649710)

[1.3 Reader’s guidelines 6](#_Toc38649711)

[2 Overview 7](#_Toc38649712)

[3 Scope definition 7](#_Toc38649713)

[4 Radio application in the RBS 7](#_Toc38649714)

[4.1 RBS Overview 7](#_Toc38649715)

[4.2 Radio Sub-System Overview 8](#_Toc38649716)

[4.3 Next Generation Radios (NGR) 11](#_Toc38649717)

[5 Summary of architectural requirements and goals 13](#_Toc38649718)

[5.1 Design goals and approach details 13](#_Toc38649719)

[6 Functional view 17](#_Toc38649720)

[6.1 Functional View Key Principles 17](#_Toc38649721)

[6.2 View model 18](#_Toc38649722)

[6.3 Reference to scenarios 36](#_Toc38649723)

[6.4 O&M layer 36](#_Toc38649724)

[6.5 LRCI/ERCI Layer 41](#_Toc38649725)

[6.6 Common Layer 41](#_Toc38649726)

[6.7 Resource layer 44](#_Toc38649727)

[6.8 Radio Modules 48](#_Toc38649728)

[6.9 Hardware Abstraction Layer 50](#_Toc38649729)

[6.10 The Driver Layer 50](#_Toc38649730)

[7 Concurrency view 55](#_Toc38649731)

[7.1 View specific principles 55](#_Toc38649732)

[7.2 View model 55](#_Toc38649733)

[8 Deployment view 58](#_Toc38649734)

[8.1 View specific principles 58](#_Toc38649735)

[8.2 LMC 58](#_Toc38649736)

[8.3 Flash 61](#_Toc38649737)

[8.4 RAM 64](#_Toc38649738)

[9 Development view 64](#_Toc38649739)

[9.1 View specific principles 64](#_Toc38649740)

[9.2 View model 65](#_Toc38649741)

[9.3 Design rules 65](#_Toc38649742)

[9.4 Source code structure 65](#_Toc38649743)

[9.5 Version handling and branching 68](#_Toc38649744)

[9.6 Test strategy 69](#_Toc38649745)

[10 ETSW 71](#_Toc38649746)

[11 RSWoS 72](#_Toc38649747)

[12 Terminology 72](#_Toc38649748)

[12.1 Service 72](#_Toc38649749)

[12.2 Controller 72](#_Toc38649750)

[12.3 Driver 73](#_Toc38649751)

[12.4 Component 73](#_Toc38649752)

[12.5 Module 73](#_Toc38649753)

[12.6 Functional Block 73](#_Toc38649754)

[13 Abbreviations 74](#_Toc38649755)

[14 References 76](#_Toc38649756)

[15 Appendix 78](#_Toc38649757)

# Introduction

Describe the objectives with the AD.

## Revision history

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev.** | **Date** | **Author** | **Comment** |
| PA1 | 2012-06-11 | EJOJANS | Document template |
| PA2 | 2012-06-14 | ELUDORE | Ch. 4 initially updated. |
| PA3 | 2012-06-21 | ELUDORE | Ch. 4 re-written. |
| PA4 | 2012-06-27 | ELUDORE | Added chapters on Robustness and PIS and references |
| PA5 | 2012-06-28 | ELUDORE | Ch. 7, “Deployment view” skeleton added |
| PA6 | 2012-06-28 | ELUDORE | Modified ch. 47, added ch. 2.1 and sub-chapters in ch. 8 |
| PA7 | 2012-07-03 | ELUDORE | Ch. 5 structure updated, more text in ch. 8 |
| PA8 | 2012-07-10 | ELUDORE | Ch. 8.7 and 9 added, other small corrections. |
| PA9 | 2012-07-16 | ELUDORE | Start writing ch. 5 and 5.6, changes in ch. 4. |
| PA10 | 2012-07-30 | ELUDORE | Added ch. 5.7, other small changes. |
| PA11 | 2012-08-06 | ELUDORE | Added ch. 7.1, other small changes. |
| PA12 | 2012-08-10 | ELUDORE | Added new ch. 4, other small changes. |
| PA13 | 2012-08-17 | ELUDORE | Changes after review of doc structure: ch. on deliveries and labling removed; added ch. 1.2, 8.1 and 8.3; changes in ch. 6.1 and 9.1. |
| PA14 | 2012-09-07 | ELUDORE | Ch. 6.4 added. Changes in view-specific principles. |
| PA15 | 2012-09-13 | ELUDORE | Ch. 8.3 added. More changes in view-specific principles. |
| PA16 | 2012-09-18 | ELUDORE | Ch. 6.5 started. |
| PA17 | 2013-07-23 | eluazal | Introduced Modularization architecture concepts |
| PA18 | 2013-08-09 | eluazal | Editorial and updated figure 8 |
| PA19 | 2013-09-01 | eluazal | Incorporate comments from Ludomir |
| PA20 | 2013-09-20 | eluazal | Incorporate changes from the Architecture WS meeting in Ottawa [Carl Jansson, Johannes Jansson, Luay Zalzalah] |
| PA22 | 2013-11-07 | Rsacjan | Updates to the “Concurrency” chapter. |
| PA25 | 2013-11-07 | eluazal | Update general and chapter 4,5, and 8 comments from formal document review |
| PA27 | 2013-11-27 | Rsacjan | Updates of chapters 6, 7 and 9 after comments from the formal document review. Remaining parts: Figure 7 and chapter 6.9 |
| PA28 | 2013-11-28 | Rsacjan | Further updates forgotten in previous revision. Chapter 6.9 (6.10 in this revision) still outstanding, awaiting input from ealbdun. |
| PA29 | 2013-11-29 | Rsacjan | Update of references and terminology. Driver chapter still outstanding. |
| PA30 | 2013-11-30 | Rsacjan | Fixed a broken reference and did spell checking |
| PB1 | 2014-02-27 | RSACJAN | Updates of chapters: 6.7, 6.10 and 10. New chapter added, 11. |
| PB2 | 2014-04-10 | RSACJAN | Attempt to make the document searchable and change the confidentiality. |
| PB3 | 2014-05-07 | RSACJAN | Updates of the “Resource Layer” chapter after comments from Luay Zalzalah. |
| PB4 | 2014-07-03 | RSACJAN | Updates after review of chapter 6.7 with design community. |
| PB5 | 2014-08-12 | RSACJAN | Changed the legend in Figure 9 (swap of LTE and WCDMA). Small corrections in chapter 6.7. |
| PB6 | 2014-08-13 | RSACJAN | Adding entry for PB5 in the revision history. Correcting references 3 and 4. |
| PB7 | 2014-08-13 | RSACJAN | Added note to chapter 10 to make it preliminary. Not part of review for revision B. |
| PB8 | 2014-08-13 | RSACJAN | Removal of yellow highlight from chapter 6.7 |
| PC1 | 2014-12-10 | eluazal | Update section 6.2.1.1.2 Service layer  Add section 6.2.1.2 Module Fault Handling  Remove sections old 6.2.1.2 Module Supervision, Compensation, and Fault handling, and Section 6.2.1.2.1 Module Diagnosis, Fault identification, and recovery  Change figure O&M layer overview in section 6.4.2  Change section 6.7 Resource layer components to add Fault Manager Interface  Add section 6.7.2 describing Fault Manager Interface |
| PC2 | 2014-12-19 | eluazal | Update to reflect the comments from the review of December 16 2014. For details of the comments please see the IR @ <http://erilink.ericsson.se/eridoc/erl/objectId/09004cff879c18ba?docno=&action=current&format=msw8> |
| PD1 | 2015-05-19 | rsaanze | Update figure 3 (chapter 6.2.1) with the FaultManager located in the resource layer.  Update figure 14 (chapter 6.4.2) to use “Equipment Resource Handler”  Update the document to be consistent in the usage of equipment resource handler. Updates performed on figure 15.  Updates to chapter 6.5 regarding interfaces and signalling.  Update chapter 6.7 with information about the equipment resource handler and the antenna resource handler.  Updates to chapter 6.8.1 describing the antenna module.  Updates to chapter 11. |
| PD3 | 2015 -05- 26 | eluazal | Added section on NGR –section 4.3-  Updated section 6.2.1.1.1 to add the different service control patterns  Updated section 6.2.1.1.2 to include southbound access from the service layer  Updated section 6.4 to include the different Northbound patterns |
| PD4 | 2015-06-03 | eluazal | Update the document with internal review comments for updates to document [PD1-PD3] |
| PD5 | 2015-06-08 | rsaanze | Chapter 6.2.1, figure 4, changed to “Radio Driver Interface (a.k.a HALI1)”  Chapter 6.2.1.1, figure 10, changed figure formatting to be able to print with white background color.  Chapter 6.4.3, figure 18, added RD/RDS interfaces |
| PD6 | 2015-06-09 | eluazal | Update figure 3 for clarity |
| PD7 | 2015-06-16 | rsaanze | Update figure 10 |
| PD8 | 2015-06-17 | eluazal | Update sections 4.3, 6.4, 6.2.1.1.1, 6.4.1 and 6.4.2 based on document formal review 2015-06-16 |
| PD9 | 2015-06-17 | rsaanze | Updates after review comments:  6.2.1 figure 4 updates  6.4.3 changes  6.5 changes |
| PD10 | 2015-06-17 | rsaanze | Updated revision history |
| PD11 | 2015-06-23 | rsaanze | Added “Device Handling Interfaces” and “Device Handling Interfaces Design Rules” to reference list. |
| PE1 | 2015-11-11 | eiandil | Update to include UL Spectrum Analyser additions to the architecture. Includes additions to Figure 4, modifications to section 6.4.1 and associated table, as well as section 6.4.2 and associated Figure 18. |
| PE2 | 2015-11-18 | eiandil | Correct previous revision in history.  Update figure 4 based on initial review comments. |
| PF1 | 2017-08-18 | emndars | ETSW (production test) chapter added. The chapter is a placeholder for ETSW information. |
| PF2 | 2020-04-24 | eraihal | Added a chapter about RSWoS (Radio SW on Stub) |

## Purpose

The purpose of this document is to present the Radio Application reference architecture with the architectural end goals in mind.

The document will also list key references which provide further useful detailed information.

The intended audience of this document is everyone who will benefit from understanding the Radio Application architecture.

The architectural goals are primarily targeting the designers, since this group is most affected by the architecture guidelines and rules -presented in this document-, as part of their task which includes maintaining the architecture.

## Reader’s guidelines

Text added or modified since the previous document version will be marked using MS Word’s change tracking or red color coding.

Note that blue color is used for instructions belonging to the document template.

# Overview

The Primary goals of the modularized Radio architecture are the following:

* Enable parallel development to decrease the communication overhead between different design teams
* Define clear boundaries of responsibility
* Introduce an easier way to scale the software in response to the Radio HW evolution
* Ensure having a one-directional dependency between loosely coupled layers.
* Define guidance and governance for key principals

# Scope definition

List which functional areas, interfaces (ANTPI, R-LRCI, etc…), and deliverables (AUBOOT, etc…) that’s included in the scope.

# Radio application in the RBS

## RBS Overview

The RBS HW may be schematically depicted as the assembly of four sub-systems (cf. ref. [16], [17] and [18]):



Figure : HW subsystems in the RBS

The RBS uses the Transport Network (TN) interface to communicate with other network elements. The 3GPP air interface (Uu) is used to communicate with end-user terminals served by the RBS.

On the downlink path, the Digital Subsystem DU receives information through the Transport Network interface and uses Baseband signal processing to encode the data as IQ modulated digital data to meet the requirements of the antenna interface (AI). The IQ digital data is then sent to the Radio Subsystem (RU) through the CPRI interface. In the RU, the IQ data is used to modulate the downlink carrier to an RF signal, which is filtered, amplified, and forwarded, to the antenna, where it is transmitted through the air interface.

On the uplink path, the RF signal received by the antenna is amplified by the Tower Mounted Amplifier (TMA) and sent to the RU, where it gets filtered, amplified again, demodulated and encoded to the IQ modulated digital signal.

Frequencies outside the range of the configured carriers are filtered out. The IQ data is sent to the Baseband handler in the Digital Subsystem, where the information is decoded, packed and transmitted through the Transport Network interface.

## Radio Sub-System Overview

The Radio Subsystem consists of specialized electronic components to handle the two-way data flows between the CPRI ports and the antenna ports (see e.g. ref. [19]).

The Radio Application is used to configure, control, and manage these components; more specifically to initialize the RU during radio board start-up, to configure them to support the carriers required by the Digital Subsystem, to make adjustment to the configuration [compensation] as result of environmental or carrier power and/or frequency related changes, and to supervise their operation to ensure the health of the RU components and take any needed corrective actions.

The Radio Application also handles measurements functions, e.g. the transmitted and received radio signal power, as well as the supervision of the antenna interface, and control of the antenna devices.

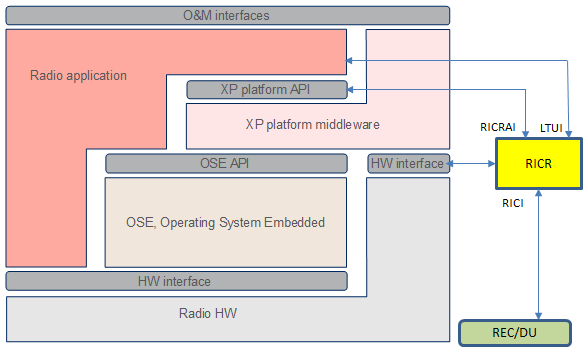


Figure : Radio application, middleware and OS

The Radio Application communicates with external world “*external client”*, typically the RBS control SW on the DU, using a collection of interfaces known as the O&M interfaces, which are implemented as asynchronous signaling interfaces tunneled through the CPRI interface. These interfaces are used to configure, control, and manage the RU entire operation, including carrier set-up, antenna equipment control, compensation, supervision, fault handling, and measurements.

After the completion of the RU start-up cycle, the Radio unit is put into an idle state –from an O&M prospective. Some Supervision activities are carried out in this state- . O&M signaling is required to set the radio to any other state.

The Radio Application uses the middleware (XP platform) API to communicate with the HW devices through the HW buses: I2C, SPI and General I/O, as well as to set up the memory bank registers for access to the memory-mapped devices.

The middleware (XP) API offers a broad list of platform services, such as access to flash and HW log, loading FGPAs, controlling the board’s man-machine interface, and more. The XP API is asynchronous and implemented as both function calls and signals.

The responsibility for the low-level HW control is shared between the Radio Application SW and the XP platform with some functions implemented by both.

Several SPI and General I/O buses are owned and controlled by the Radio Application. These buses are accessed using the memory-mapped I/O rather than XP API.

FPGAs loading have 2 distinct implementations in the XP platform, and the Radio Application.

The Operating System (OSE) API is used for inter-process communication, spawning processes, file system access and other operating system services. The OSE API is implemented as function calls. The procedures are synchronous and often blocking.

The HW interface to access the HW devices on the Radio board is defined in terms of addresses and data. The HW interface defines what, where and in what sequence to write to the HW to achieve wanted behavior, as well as how to read the HW, to retrieve the required data.

The RICR software is responsible for management of communication links between baseband modules and RE units. An Ericsson proprietary version of the CPRI protocol is used for these links, which is capable of handling O&M control signaling, I/Q data, and synchronization between RE and baseband modules.

RICR supports cascading, time distribution and error handling of the CPRI link.

RICR provides the following SW interfaces:

* RICRAI - the interface towards the Radio Application.
* LTUI – An interface towards the Radio Application for LTU management
* RICI - the signal interface between a REC (for example RICM on DU) and RICR.

RICR interacts with the following hardware:

* COM FPGA
* XCU FPGA

RICR is implemented as a library which is included in RE SW. RE SW initializes RICR when the FPGA is loaded. RICR then starts its own processes.

**NOTE:**

As shown in Figure 2, term *Radio Application* denotes the uppermost SW layer. The term *Radio SW* is used in this document for all the SW on the Radio board, including the middleware (XPP) and Operating System (OSE), RICR, as well as the binaries loaded on the FPGAs.

## Next Generation Radios (NGR)

As part of the Radio product evolution roadmap to address customer needs and provide advanced products the NGR RU program was launched.

NGR RU products have 3 flavors:

NGR LFT: This radio is developed in China to support LTE FDD & TDD. However this radio does not share the global radio architecture and SW.

NGR G1: Same radio application (a.k.a Global Software), middleware and OS structure as seen in Figure 2

NGR G2 and beyond: These radio products use a new enhanced Execution Environment “XCS” and Linux Operating System –see [44] for details. These radios will have same radio application SW (a.k.a Global SW), but different middleware and OS structure as in see in Figure 3.

Key benefits of the new Execution Environment (EE):

* Align and share EE with the RBS Control System
* Better debug and development tools
* Reduce development costs

The new OS “Linux” provides many rich functionalities including:

* Deployment of functional blocks in separate load modules or executables
* Support for shared libraries (Static & Dynamic)

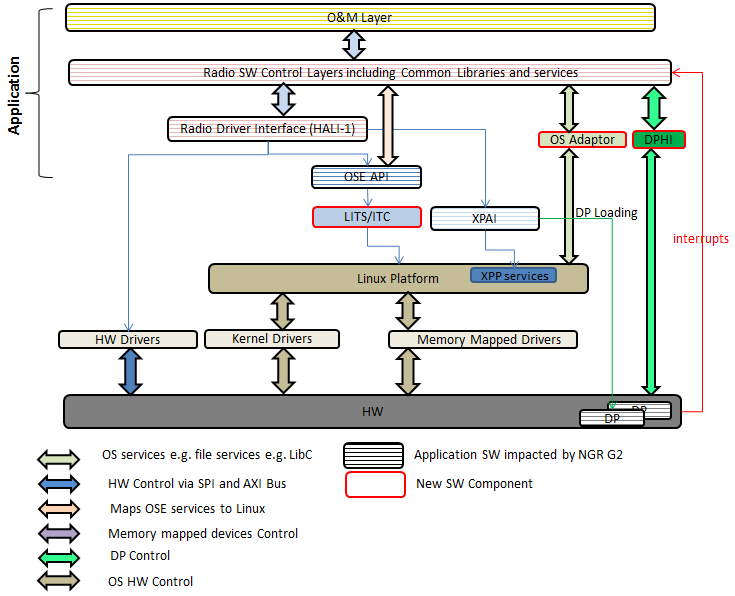


Figure Linux and XCS Based Radio application, middleware and OS

# Summary of architectural requirements and goals

The goal for the Radio Application architecture is to provide efficiency, in terms of RU SW product delivery to satisfy TTM and product cost. The architectural decisions have to embrace the following principles:

* The Radio Application supports a large number of HW variants of:
  + HW platforms
  + Technology platforms, product platforms, and frequency variants [11] -. More than twenty new Radio products, RUS/RRUS/AIR/RIR, are released every year, together with our SW.
  + frequency variants -new variants are added constantly-
* Enable the introduction of new features and functions with relative ease -introduced all the time, to both new and old Radio products to meet customer needs-
* RU SW meets the available HW resources for the different HW variants –Some HW variants have very limited resources-
* Enable parallel SW development - the architecture minimizes the clashes during development activities- through the support of:
* Loosely coupled Layers architecture
  + Functional encapsulation at each layer
  + Clear architectural guidelines governing the scope and granularity of the SW layers and components
* Clear and well defined interface
* Clear data ownership rules
* Scalable -Easy to expand -
* Maintainable
* Testability

## Design goals and approach details

The list of the specific design goals to achieve efficient SW development is detailed below. The main objective for the overall RU SW strategy is to handle the support of many Radio (product) variants using *common code*, and to develop code which is *easily adapted to HW changes*.

### One source code for all HW

To limit the effort for development, maintenance, verification and product release, the use of a single Radio Application source code supporting the active HW variants is deemed most appropriate.

However, there is a trade-off between *the savings* from maintaining a single code base, and *the cost* of verifying all HW, in the scenarios where the SW change is limited in its impact to few products/variants. Creating a separate code base for a group of similar products, e.g. a specific HW platform, may be deemed to be more efficient in such scenarios.

### Code re-use

To keep the Radio Application compact and to ease the SW maintenance, the use of common code for SW functions on similar components on different boards should be strongly considered -replicating code should be avoided-.

See [39] for details.

### Scalability

The architecture should support an approach that enables the support of different RU configurations using different HW variants with different capabilities to include:

* Multi-Banding
* Variant number of TX and Rx branches [0-n]
* Multi-clients.
* Services selection

### Structure for parallel development

To achieve rapid development and short TTM, the Radio Application architecture enables parallel development by different teams. This drives the need to stress importance of adhering to the key architecture requirements:

* Structuring the RU application SW in loosely coupled SW layers
* Clear interfaces between the layers
* Isolation and encapsulation of software functional blocks

### Efficient use of RU resources

The Radio Application is used on different Radio HW platforms with different available computer resources.

In general development usually targets the latest HW generations, but the new features are also made available in the load to customers that have several older versions of the RU HW.

This means that consideration is made to ensure that the new code fits onto the old HW, and complies with its limitations on the CPU memory and flash capacities.

Keeping the memory footprint and the CPU load low is an important design goal in each development activity.

### Testability and observability

Testability of the SW is considered at the very early stages of the design cycle to ensure compliance of the SW with the requirements, and to easily identify possible defects, and observe the execution path/sequence within the application.

### Timely operation

When developing Radio SW, priority of actions handling, and execution time is taken into consideration to ensure that none of the time requirements are compromised.

See ref. [1] and [2] for timing requirements on O&M procedures such as SW loading, board restart and carrier operations. Ref. [5] adds Radio-internal timing requirements related to the properties of the HW, e.g. the need to instantaneously switch off the PA before the board restarts or to complete a supervision loop before it is triggered next time.

### Robustness

Radio Application should adhere to the systemized behavior with emphasis on the systemized abnormal and/or error handling scenarios. Examples of such situations are inconsistent multi-client requests or high temperature scenarios handling.

For non-systemized abnormal conditions, the Radio Application should attempt to recover to normal operation. If recovery and achieving normal operation cannot be achieved or guaranteed, the Radio Application should generate a useful trace log and exit in a controlled manner with a memory dump.

### Isolation of external interfaces

Radio SW’s main interfaces are the interface to the Digital Unit, which encapsulates Radio’s functionality, and the interface to HW, which reflects the implementation on the chip level. These interfaces change independently and in different cadences: new functions are released twice a year, while new HW is released on a weekly basis.

The Radio SW’s architecture ensures that changes to interfaces -especially the HW interfaces-, are contained and applied at the appropriate application layer in accordance of the SW architecture.

### Maintainability

Maintainability and understandability are key design goals to consider when writing the code. Keep the design and implementation as simple as possible and avoid “smart” solutions.

Follow the design rules and document the code using meaningful and useful comments.

# Functional view

The RU software is based on a layered architecture. Layers are arranged in a manner where the most abstract layer is at the top –close to the external interfaces-, and the most concrete layers is at the bottom -close to the hardware-.

## Functional View Key Principles

This section will provide details on the key principles governing the Radio Application Functional view.

**NOTE:**

See ref. [10] for more information about the current/legacy Radio Application functionality, architecture and many practical programming aspects.

See ref. [37] for more information on Radio Application Modularization fundamentals.

See ref. [11] for details on the Driver Layer in context of the Radio HW.

### Controlled inter-layer dependencies

Each layer provides a set of interfaces for the layer above, and only uses the interface published by the layer below. There are no other dependencies between the layers. In the implementation domain this means that a layer only includes the header files belonging to the layer directly below it, and the included header files belonging to this layer’s official interface.

### Clear and well-documented inter-layer interfaces

When developing within a layer, it is important to know the interfaces; both the ones being implemented and realized by the intended implementation and the interfaces the implementation is supposed to use (provided by the lower layer). It is also important to not design against the implementation of the provided interfaces, since then the meaning of the interface is lost. To get there, the interfaces between the layers must be clearly defined and well documented.

### Data ownership and Validation

Data ownership is determined by its primary relevance of the data to the layer and its components.

Configuration data specific to a layer or module within a layer is pushed to that target from the upper OAM layer, and will be validate and owned by the target.

Configuration data is to be maintained by the owner and every effort is made to avoid copying and changing of the configuration data.

For functional modules controllers are initialized with information from the database. Similarly services are initialized with data from the database and information supplied by the service controller. Each Module entity is to only own their initialization data, data that is affected by the service due to change of state, and data needed for connecting with the drivers (internal data is excluded).

### Data-driven operation

To minimize code changes support different HW variants and new features, the Radio Application are highly data-driven with the database having a prominent role in the support of the many HW variants.

Whenever changes in the Radio Application are done, the possibility to implement them through changes of configuration data in the database is considered first. Thinking about data-driven operation, and designing the database accordingly, are important principles of the Radio Application.

## View model

### Loosely coupled Layered architecture

The recommendations for the relationships between the layers are:

* A layer cannot include a header file belonging to a layer above it in the hierarchy.
* Data required by lower layer is made available [pushed] by upper layer.
* Data specific to a layer should be owned and maintained by that layer.
* A lower layer can’t initiate the first contact with a higher layer.
* Common libraries [layer] are situated on the side of the other layers. Other layers can depend on the Common libraries, but the Common libraries shall not depend on the other layers of the Radio Application.

In the current –legacy- SW implementation there are some violations to the above rules. Such violations, however, should be removed as part of the RU SW Modularization activities.

Figure 4 shows the Radio Application layered architecture. This structure is the Radio Application end goal and will be maintained in the future function growth. The layered architecture adheres to the design goal of external interface isolation to keep the SW changes stemming from the interface changes as local as possible.

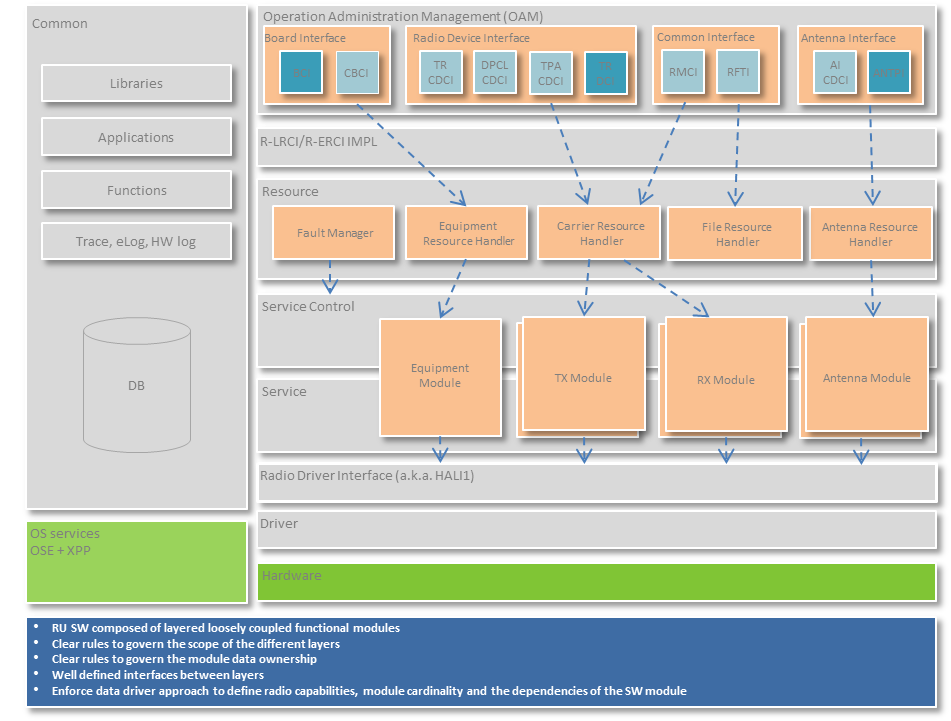


Figure Radio SW layered architecture

#### Functional blocks

The radio software is divided into several Functional and Coordination Blocks.

Please refer to [37] for details.

The Functional Blocks are responsible for configuration, control, and management of the HW blocks determined to be in their scope, and will communicate via the interface HALI1 towards the driver layer, and towards the O&M layer using the R-LRCI and R-ERCI interface. Each Module in the functional block will export interfaces that cover areas in their scope in accordance to the Interface Segregation Principle (“Clients should not be forced to depend upon interfaces that they do not use”).

Coordination blocks are used to manage and control the Functional Blocks and coordinate the common functions of interest to multiple Functional blocks; for an instance, Fault management is done by the Common Functional Block and Carrier Status Coordination, between DL and UL Functional Block is done by the Resource Control Functional Block. The coordination blocks are intended to be functioning in the resource layer (see Figure 4 ).

Functional blocks will adhere to the modularization architecture principles –see Figure 11 -, and each functional block will consist of two layers as follows:

* Service Control Layer
* Service Layer

Functional blocks

* Downlink Functional Block -maps to TX/DL Module-: Contains all functionality related to the TX (downlink) chain control and supervision including:
  + Carrier realization
  + DL HW supervision
  + DL Time alignment
* Uplink Functional Block –maps to RX/UL Module: Contains all functionality related to the RX (uplink) chain control and supervision
* Antenna Functional Block –maps to Antenna Module: Contains all functionality related to the antenna control and supervision
* Equipment Functional Block –maps to Equipment Module: Contains common equipment Control and supervision on the RU board, and includes:
* External Alarm
* Hotspot Reporting
* MMI Handling
* SW Compatibility List Handling
* 48V Supervision

Coordination blocks

* Common Coordination Block
* Resource Control Coordination Block

##### Service Control Layer:

The Service Control Layer contains the Service Controllers which are responsible to implement the module interface –to provide upper layers access to the module- and orchestrate logic that can be associated with certain functionality. An example of such functionality is Carrier configuration event handling, which deals with the allocation and control of radio resources.

The Service Control Layer may also contain sub-controllers that encapsulates the orchestration of specific functionalities in order to off-load the main controller.

Service Controllers and sub-controllers should not contain logics to control the hardware; this is the task of the services. Controllers and sub-controllers may have multiple functional scopes that can be accesses by groups of clearly defined interfaces. The Service Controllers and sub-controllers orchestrates the services and can look different for different deployments, but the intention is that changes to the service controller mainly shall occur when there is a change in the functionality for a module or when the order of execution of a function changes.

There are several patterns to the realization of controllers and sub-controllers[[1]](#footnote-1) as follows –see Figure 5-:

1. One main controller with one or more sub-controllers: Access to sub-controllers from upper layers will be through the main controller (i.e. main controller manages the coordination and is aware of the sub-controllers).
2. Multiple main controllers with one or more sub-controllers in scope of each main controller
   * + - 1. Main controllers must have no dependencies to each other [no orchestration is needed].
         2. Each main controller will have its specific I/Fs and upper layers can access each of the main controller directly through their specific I/Fs.
         3. Controllers and Sub-controllers cannot share the same service functional scope. Main controllers cannot share the same sub-controller functional scope

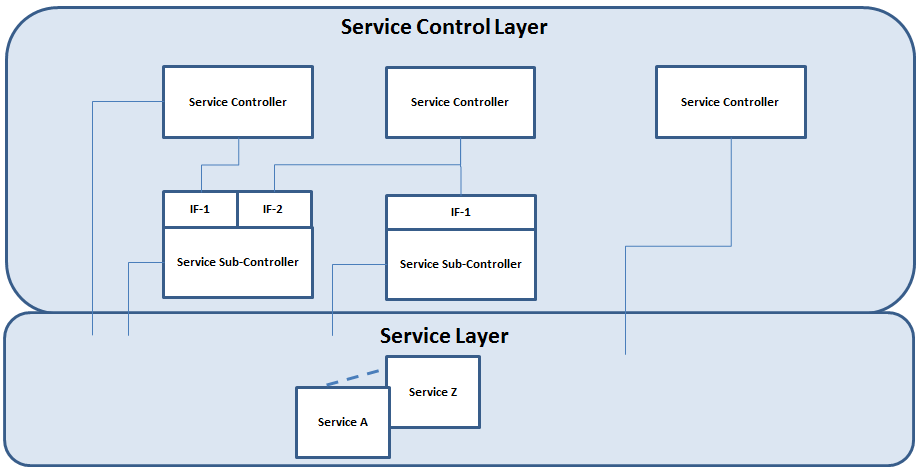


Figure Service Control Layer Patterns

##### Service Layer:

The Service layer contains the services which represents the HW-blocks and are strongly associated with the HW they are executing on. An exception to this is the coordination service, which has no direct association with the hardware.

Services interact with the HW either through HW drivers via HALI-1 interface layer or through southbound client signaling interface (as the case in RDS. See Figure 17.

Services may have different functional scope to include:

* Control: Typical operation would include HW configuration due to a specific carrier event.
* Compensation: Triggered periodically to preform HW configuration compensation due to environmental changes
* Supervision: Triggered periodically to verify the health of the HW.
* Measurement: Triggered on demand to provide performance and operation measurements

Each service functional scope should be encapsulated in its own component and should be in scope of one controller or sub-controller. Interactions between the components serving the Same HW block should be loosely coupled.

To support a HW block multiple functional scopes (such as Control, Compensation, Supervision, and Measurements) the following design implementation patterns can be considered:

1. Include the different HW functional scope support in a single HW block service, and have each functional scope encapsulated in a single sub-component referred to as functional component. The different functional components of the Service should be loosely coupled - data driven-.

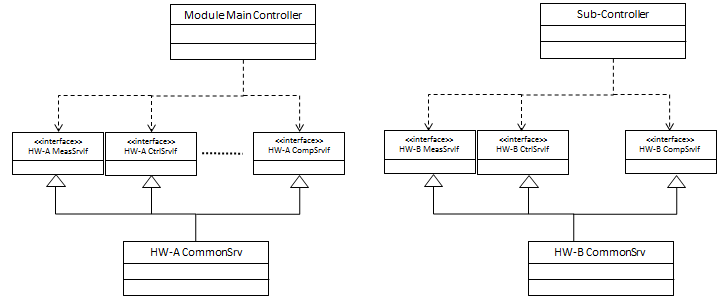


Figure 6 Single HW Block Service Pattern

1. Define distinct service for each HW block functional scope–Control, Supervision, Compensation, Measurement, etc…- Any interaction or dependencies between the different functional scope services for a single HW block should be loosely coupled –data driven-.

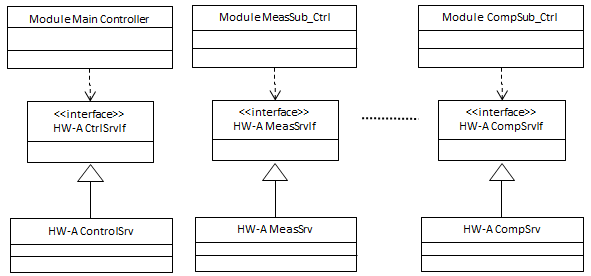


Figure 7 Distinct Service for each HW Block Functional Scope Pattern

1. Define two distinct services for the HW block based on the action type performed –Read or Write-. With this option Control & Compensation will have a single service with 2 functional components, and Supervision and Measurement will be having a single service. Any interaction or dependencies between the different functional scope services for a single HW block should be loosely coupled –data driven-.

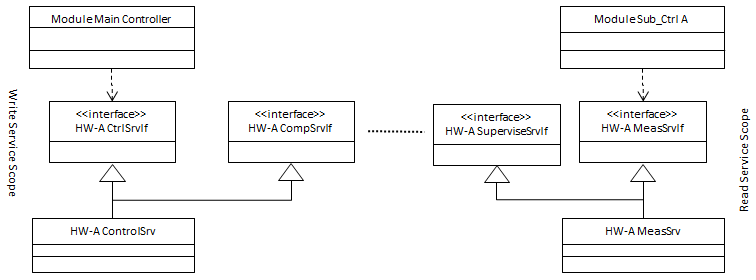


Figure 8 Two HW block Services (Read or Write Only) Pattern

1. Mix of option A and B above, where we have a common service [hiding the different scopes of the service] but have one or more sub-controllers with sub-controllers interacting with a specific functional scope of the service. Any interaction or dependencies between the different functional scope services for a single HW block should be loosely coupled –data driven-.

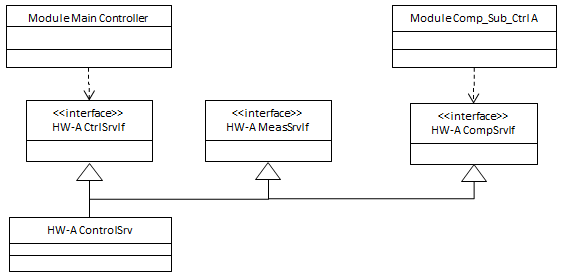


Figure 9 Hybrid HW Block Service Pattern

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Option 1 Single HW block service** | **Option 2 Distinct Service for each Functional Scope** | **Option 3 Two HW block Services (Read Only or Write)** | **Option 4 Hybrid -Previous Options-** |
| **Pros** | Reduced number of services –DB savings- | • Different Functional Scope needs are clearly/easily visible • One pattern to address HW block different functional scope • Prevent the creation of bloated/giant services • Changes in one functional scope doesn’t impact the entire service | • Reduced number of services –DB savings- • One pattern to address HW block different functional scope • Reduce the potential creation of bloated/giant services | • Flexible • Can include the advantages of previous options |
| **Cons** | • Different Functional Scope needs are not clearly/easily visible • Some functional needs of the service are not always within the scope of the same control HW block –e.g. temperature- resulting in the need to use different patterns • Can result in the creation bloated/giant services • Changes in one functional scope impact the entire service | • Increase in number of services –DB size impact- • Requires infrastructure to define/data-drive the interaction dependencies between the different services associated with the HW block | • Different Functional Scope needs are not clearly/easily visible • Changes in one functional scope impact the entire service | • No single pattern to follow • Difficult to enforce rules on how to use or monitor • Can include the disadvantages of previous options |
| **Design Consideration** | Use this pattern as long as the scope is rather small and the size of the combined service is small, this is a nice and easy way to deal with the implementation. If the scope grows, refactoring should be considered and split up the service in separate classes | Use this pattern for medium sized and large services with more than one aspect (control and supervision for example) | Use this pattern when the service will provide either read or write functionality only. | See design considerations for options 1 & 2 but beware of bloating |

Table 1: Service Scope Implementation Patter Comparison

With the aid of the Service Factory Framework, the Services are made exchangeable between different deployments.

Ex. PaSrv for RU configured for TDD will have a different flavor of the service, and will be identified in the service DB.

Interfaces toward the controller is the same but that is not necessary the case towards the drivers.

See Figure 10 how services will be handled on different radios depending on the HW deployment.



Figure Services and blocks

Services associated to a specific HW block do not have knowledge of other services associated with other HW blocks. This means that services shall not communicate with other services.

A service can be shared by multiple modules of the same type; such services are called shared services [e.g. TxLo Service]

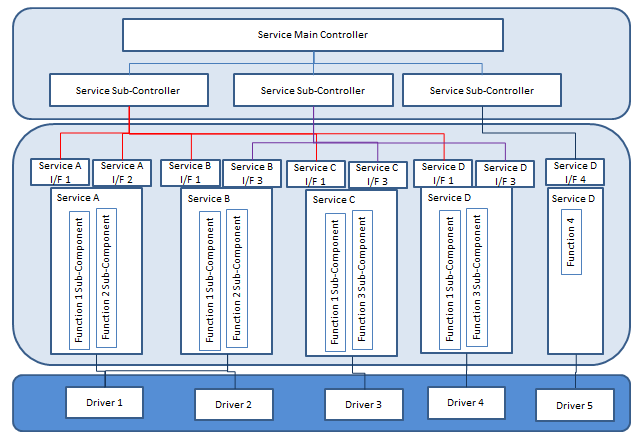


Figure Functional block module architecture

Shared services operate via broadcasting of events to the subscribing controllers -the Controllers subscribe to shared service events at initialization time-.

There are two types of shared services. One to serve a HW block shared between two or more modules of the same type [group of Modules]. And the second type is coordination services to coordinate between multiple modules cycles and prevent deadlocks.

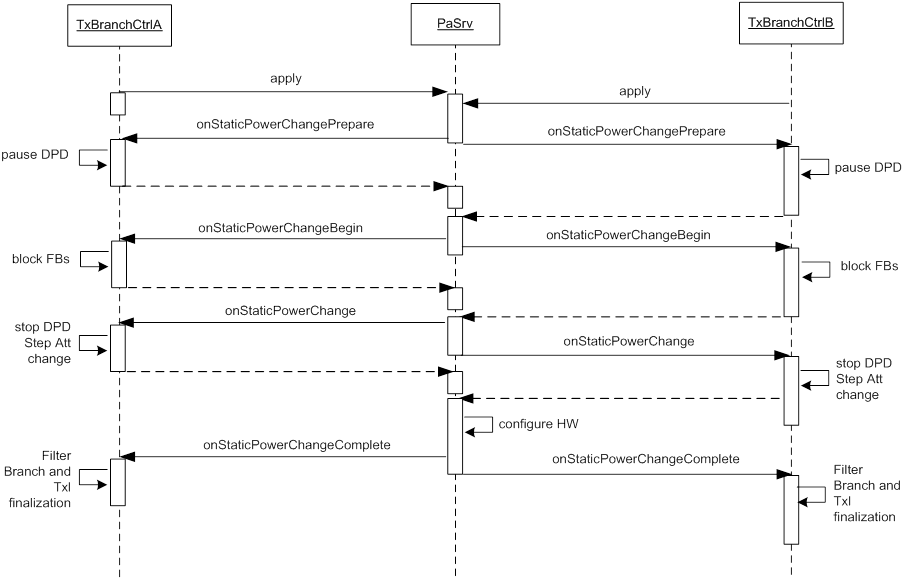
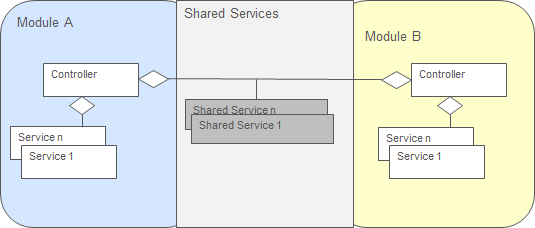


Figure Shared model and sequence diagram example

#### Module Fault Handling, Recovery, and HW Protection

Module may need mechanisms to handle fault triggers from both within the module, or external to the module –e.g. other modules or module types-

There are two types of fault -which the module need to handle- as follows:

1. Recoverable Faults: Faults that require the Module to attempt to take corrective action, and would only be reported if the recovery action fails or the fault occurrence is observed again –defined by rules- e.g. within a certain time.
2. Unrecoverable Faults: Hardware faults that render the Module not operational, and would require the module to take protection action to prevent further damage to the Radio, and take the Module out of service. Unrecoverable Faults are immediately reported.

Module Fault infrastructure main functionalities are:

* Detection
  + Discover all local failures within the Module
* Localization
  + Pin-point the fault source(s) –e.g. Linearization fault can be caused by PA high current in Tx Module-
* Isolation
  + Immediately triggering protective action (e.g. 0-insertion or taking the HW component out of service) towards the module HW component to prevent HW damage.
* Recovery
  + Attempt to recover from the local fault by restarting the Module or part of the Module
* Reporting
  + Inform the Radio Fault Manager about the module local fault

Module Fault infrastructure however is NOT intended to support the following:

* Fault Filtering
  + -Verifying the disturbances before reporting them as faults to the Fault Manager –i.e. maintain fault filtering at the Fault Manager scope-
* Fault Coordination
  + Association and separation of primary local faults from secondary local faults –caused by dependencies between various Radio HW parts
* Fault Suppression
  + Disabling of local faults which depends on other active faults or fulfilled conditions

**Note**: In case of faults detected by a shared services. Multiple faults will be reported to the individual fault advisors. Filtering and consolidation of these faults will be handled by the Fault Manager.

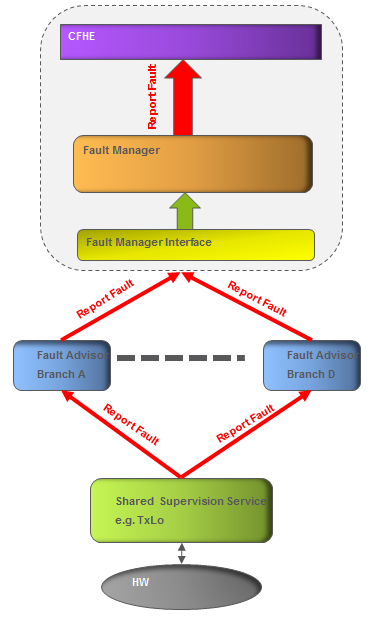


Figure Shared HW Module Fault handling

**Module fault infrastructure includes the following components**:

1. **Fault Advisor**: This is a module sub-controller that provides the following functionalities:
   1. Create and destroy the Fault Object
      1. Only one Fault object for a specific Fault event can exist at time[[2]](#footnote-2)
      2. No new Fault objects are created if the service newly detected fault is deemed related to an already reported fault –reported by either the same service or another service within the module-
   2. Trigger recovery or protection activities based on logic encapsulated within the Fault Object towards the Module Main Controller
   3. Report the Module Faults to the Fault Manager -–Triggered by Fault Object-–using generic fault IDs
   4. Trigger abort recovery cycle request towards the Fault Object
   5. Orchestrate the handling of multiple faults occurring simultaneously.
2. **Fault Object**: An object that is created when a specific fault event is received –the scope of each Fault Object is single fault type-, and is destroyed when fault event is reported by the Fault Advisor to the Fault Manager. A typical Fault Object would encapsulate the following:
   1. Fault ID
   2. Fault type
      1. Recoverable vs. Un-Recoverable
   3. Recovery or Protection Action List to include:
      1. Logic for each action
      2. Number of recovery attempts
      3. History of successful recoveries
   4. Manages the fault Recovery Rules including tracking the number of successful recoveries and timeout triggers –Used to determine if a recoverable fault need to be reported-

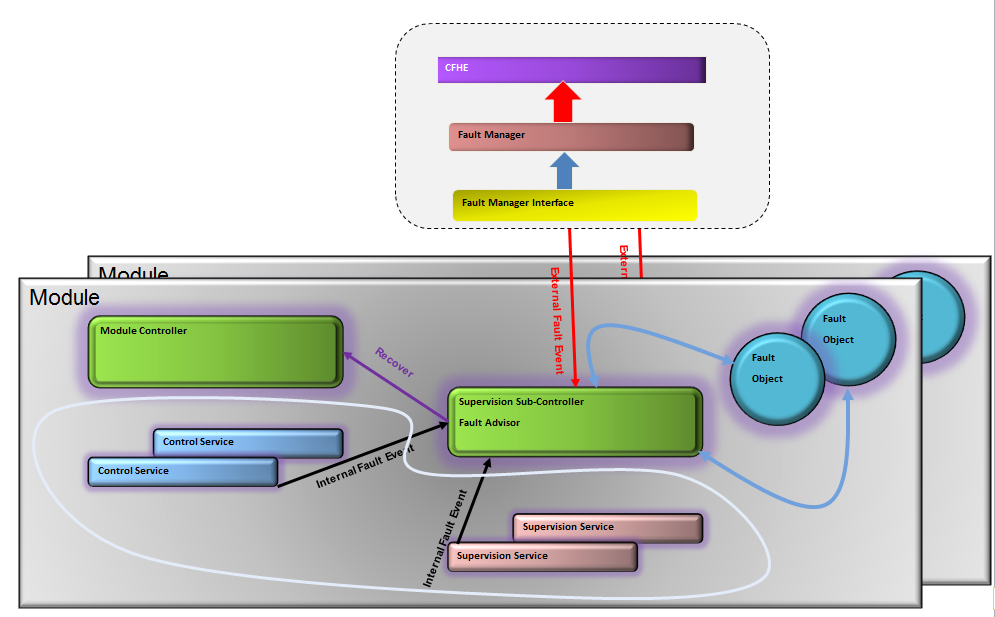


Figure Module Fault Management View

##### Typical Fault Handling examples

##### **Handling of Recoverable Fault**:

Successful recovery of Recoverable Fault will result in the following:

* + - * + Fault is placed in Pending State (Object is not destroyed)
        + Fault is not reported

Fault Object will manage the history of successful recovery and the recovery rules –i.e. reporting the fault if the fault occurs certain number of times with in a specific period of time.

Unsuccessful recovery of Recoverable fault (including retries) will result in fault being reported and Fault Objected destroyed

##### **Handling of Unrecoverable Fault:**

Module performs any necessary protection actions to safeguard the HW blocks in scope of the module, and report the fault.

Figure 15 and Figure 16 present typical high-level view for module fault handling flow

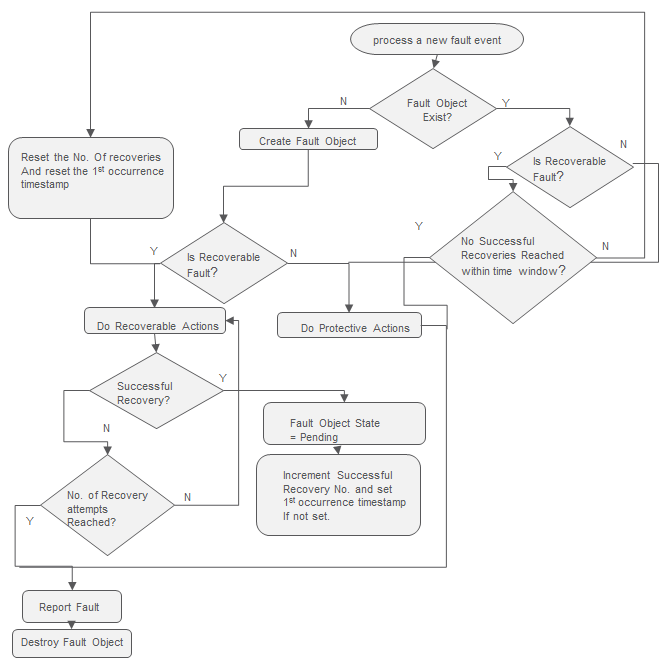


Figure Single Module Fault Flow

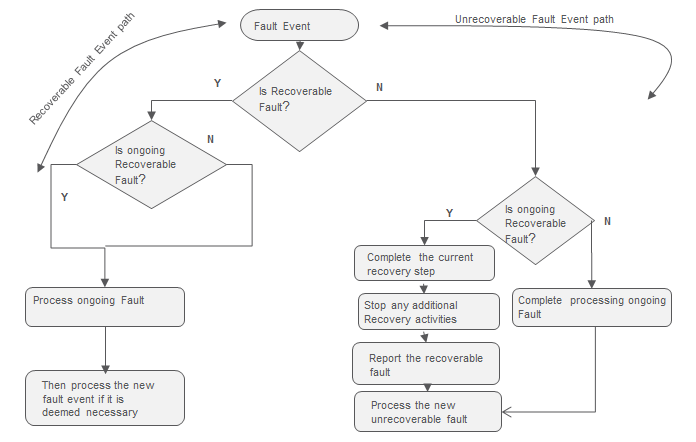


Figure Concurrent Module Fault Flow

### Service Framework

Service framework provides infrastructure that provide a module with the following:

* Service Factory: Provides the service creation framework -data-driven based-
* Service Container: Provides the framework for service registration and lookup

### Service Database

Module services creation are Database driven. Service definitions are located in the serviceSet DB.

The rules regarding database parameters can be found in [38].

/xxx/[[3]](#footnote-3)mmServices key defines the list of the mmModule services for a specific platform

For each service the following DB keys are defined

**/mm/*<service name>/*type:** Service class type (i.e. class name)

**/mm/<service name/txBranchId:** Service branch list (A – H). Empty or missing key indicates branchless service.

1 service instance is created if mmBranchId is empty/missing (instance name = SERVICE ID).

N service instances are created for each branch service (instance name = SERVICE ID + BRANCH ID)

## Reference to scenarios

## O&M layer

As shown in Figure 4, the O&M layer terminates the Northbound O&M interfaces.

There are 2 patterns for O&M Infrastructures as follows:

* Standalone: Infrastructures owned by Application SW –Part of the O&M Layer- and communicates with the layers below it through the LRCI/ERCI interfaces
* Pluggable: An independent –black box and not part of the O&M Layer- Infrastructure provided as binary by other SW group(s) with clearly defined IWDs defining the integration points towards the application. Pluggable O&M components may have associated Southbound components to communicate with external equipment e.g. ATC

One of the key O&M layer objectives is to properly handle the changes in the O&M interfaces, and hides them form the layers below –unless the changes introduce a new function or modify an existing functionality-

The O&M layer as well as pluggable O&M infrastructure publishes the O&M interfaces as the last step in the radio board initialization cycle. This triggers the external clients to establish connection and start sending requests.

More information about the O&M layer may be found in ref. [22] and [23], where the original and the current implementation of the CCIE are shown. Ref. [24] presents the CFHE module.

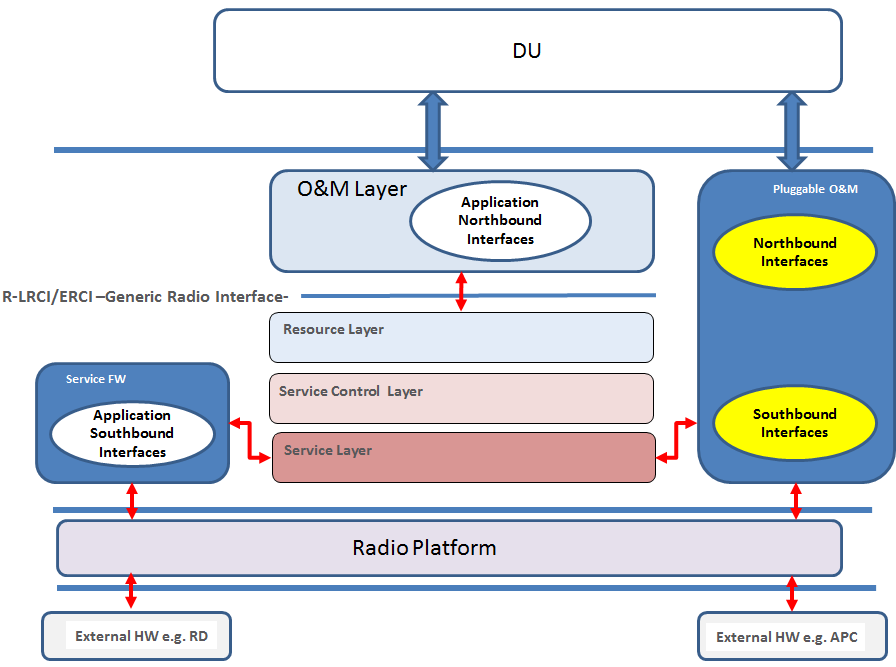


Figure O&M Patterns

### Standalone O&M interfaces

The standalone O&M interfaces consist of the following groups:

* WCDMA Board and Device interfaces
* LTE/GSM/CDMA Board and Device interfaces
* Measurement interfaces
* File transfer interfaces

Board-level interfaces are more general in nature and some functions are not supported by the Radio, but are used by other boards in the RBS, while the Device-level interfaces are radio specific.

|  |  |  |
| --- | --- | --- |
| **Interface name** | **RAN Standard** | **Level** |
| BCI | LTE/GSM/CDMA | Board |
| RRU CBCI | [[4]](#footnote-4)WCDMA | Board |
| RUW CBCI | WCDMA | Board |
| TR DCI | LTE/GSM/CDMA | Device |
| ANTPI | LTE/GSM | Device |
| TR CDCI | WCDMA | Device |
| TPA CDCI | WCDMA | Device |
| AI CDCI | WCDMA | Device |
| DPCL CDCI | WCDMA | Device |
| ALM CDCI | WCDMA/LTE/GSM/CDMA | Device |
| RMCI | WCDMA/LTE | Device |
| RFTI | WCDMA/LTE | Device |

Table OAM Supported Interfaces

The main functions of the Board Control Interface (BCI) are:

* Load module handling
* Board state handling
* Board temperature reporting
* Control and reporting of MMI LEDs and button
* HW log read/write/erase
* Board-level fault reporting
* Triggering of board restarts

The main functions of the Transceiver Device Control Interface (TR DCI) are:

* Radio capability reporting (output power, frequency range, internal bandwidth, number of carriers, etc…)
* Carrier setup and modification including power, frequency, bandwidth, time alignment and I/Q streams settings
* Handling of antenna port characteristics
* Cross-connect port setup and supervision
* Device-level fault reporting
* Configuration of RF-ports

The main functions of the Antenna Port Interface (ANTPI) are:

* Antenna port capability reporting (voltage, supervision, modem, supported antenna line devices)
* Control of voltage and modem
* Connection with antenna line devices TMA or RET
* Sending data to antenna devices
* Antenna system supervision and fault reporting

The functionality of the WCDMA interfaces is similar, but not fully equivalent, and spread over several interfaces –see Table 2. E.g. the antenna port voltage, modem and carrier time alignment are controlled via the board-level interfaces.

The DUS G2 (Generation 2) intends to use the LTE-interfaces as a baseline for the WCDMA O&M. This means that the LTE-interfaces will be used for all RAT’s supported by the Radio Unit. There will still be a need to support the legacy WCDMA interface, though.

The O&M layer also handles the Inter-Device Control Interface (IDCI ROT) for **R**aise **O**ver **T**hermal, used for measurements of the received signal power. This interface is not an O&M interface but it is handled in a similar way.

### Internal structure

The main components of the O&M layer today are shown Figure 18

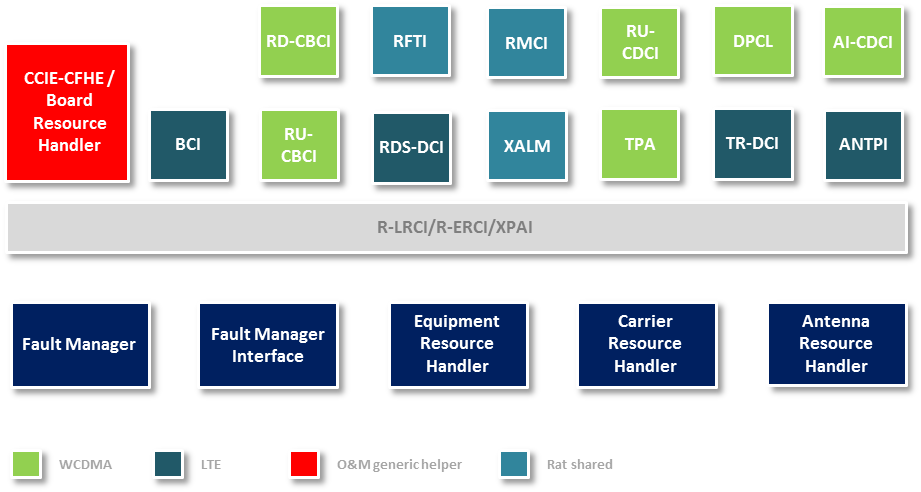


Figure O&M layer overview

The Common Communication Interface Engine [CCIE] contains functionality common to all O&M interfaces such as:

* Publishing the interface
* Protocol revision negotiation
* Sending/receiving client signals (except some indications)
* Maintaining the interface state
* Forwarding fault indication from the CFHE to the subscribing clients
* Translating between the global and the local device ids
* Terminating fault and measurement subscriptions
* Implementing the state machines given by [41]

Some of the client requests are handled internally in the CCIE, e.g. enabling and disabling the interface, audit request, fault subscriptions, and device address configuration. Other requests are passed to the components capable of dealing with these requests, i.e. requests that are to be carried out by the Radio Application software and affect the radio hardware.

CCIE operation is controlled using a table. It defines the signals belonging to the interface, actions to be executed at signal reception and interface state changes at successful completion. CCIE supports two device models with different sets of state machines (see [45] and [46]).

There are several different groups of interface servers; equipment related (RU-CBCI and BCI), carrier related (TR-CDCI, DPCL, TPA for WCDMA and TR-DCI for LTE), antenna related (AI-CDCI for WCDMA and ANTPI for LTE) external alarm, measurement (RMCI), and file transfer (RFTI).

The server receives the client requests from the CCIE, performs some sanity checks and converts the request to function calls through the LRCI/ ERCI, depending on the intended target. A single client request may result to a series of LRCI/ERCI calls.

The majority of client requests made through BCI and CBCI interfaces are passed to the Board Resource Handler component in CCIE. Requests with long execution time, especially operations on flash, are handled in separate processes to make the Board Resource Handler available for other requests.

The interface servers operate independently and in parallel, which satisfies the RU SW multi-client support requirement. Possible resource conflicts, especially access to flash, are resolved by the Board Resource Handler. For this purpose, the Board Resource Handler implements the resource lock, which may be acquired by one client at a time, keeping the other clients on hold until the operation is completed. Other resource conflicts, e.g. conflicts associated with carrier resources, are controlled by lower layers.

### Common Fault Handler

The Common Fault Hander keeps the high level fault state of the faults that are sent from the Radio to the clients.

The CFHE receives local fault indications, generates the client fault indications, and sends the indications to the subscribers through the CCIE. The CFHE has the following key responsibilities:

* Fault Translation: Translate local Radio faults to RBS faults. In some cases, this may include mapping several local faults to a single RBS fault and/or the-other way around.
* Fault prioritization: Ensure that only the most severe Radio fault is presented externally. A new local fault with higher severity overrides and masks an existing lower priority fault.
* Generation, logging and sending the fault indications and fault ceased indications matching the active subscriptions.
* Communicating the HW fault status to the CCIE Board Resource Handler for the red LED control.

The CFHE receives the local fault indications through the LRCI/ERCI and XPAI.

Faults detected by various supervision functions in the different modules are reported to CFHE through the LRCI layer. The XPAI conveys to faults detected by the XP platform.

Fault handling is heavily table-driven. The fault table holds the mappings between the internal fault ids and fault types text strings used in the logs, delay values, etc. For every fault, this table also keeps the list of higher-priority faults, which would suppress the fault if reported simultaneously. The fault map describes how Radio faults map onto the signals sent to the clients.

## LRCI/ERCI Layer

The LRCI/ERCI layer is the glue between the O&M layer through which external client requests are received, and the other layers –specifically Functional Blocks Modules- that act on these requests.

The LRCI/ERCI layer provides a collection of interfaces grouped into functional domains (FUDs), see [16]. Each interface is implemented as a C++ class whose methods trigger the associated services.

The LRCI/ERCI layer is only partly implemented. As of the date of the document, some signaling initiated by the WCDMA client requests bypasses the interface.

## Common Layer

The Common SW Block contains functionality and data which are of interest to several modules types in the functional layer. Key components of the common layer include:

* Event Log
* HW Log
* Trace and Error Log
* Common data and data types
* Common functions
* Common libraries
* Database
* Utilities

### Database

The database consists of two main parts the *production database* and the *SW database.*

Production database

The production database is saved during the Radio HW manufacturing process. It contains parameters identifying the HW components and their characteristics measured during the HW calibration, e.g. frequency response curves and temperature limits. This data is used to activate and control relevant parts of the Radio Application, e.g. to start the correct drivers.

The production database is delivered with the Radio HW, partly in flash and partly in EEPROM, and it is sometimes called the *HW database*. This database is released together with the Radio HW and if is modified, the HW R-State is stepped-up.

To generate the production database, the factory uses document “Database Update Description” written for each Radio product during its development. An example of such a document is given in ref. [12].

SW database

This database contains alphanumeric constants used in the Radio Application. The main objective of the SW database is to make the Radio Application more data driven; allowing a broad class of HW variants to be supported solely by adding information to the database, without any SW change. This simplifies the quality assurance and makes the TTM shorter.

The SW database is defined during the SW development and consists of several text files which are concatenated and converted to a binary file during the load module generation. This database is delivered and released together with the Radio Application.

A database entry is a triplet entity (key-type-value), where:

* *key* is a slash-separated list of identifiers. The identifiers are called *segments*.
* *type* is one of char, U8, U16, U32, S8, S16, S32.
* *value* is a character string for type “char” and an array of integers for an integer type.

Below are some examples of database entries fetched from a real Radio product.

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** | **Type** | **Value** | **Comment** |
| /board/powerClassUsage | char | 30W | Output power |
| /board/freqClassUsage | char | BandVIII | 3GPP band number |
| /board/tx/carrierCnt | U16 | 4 | Number of carriers |
| /board/trxDrivers/set | char | 431 | Selects TRX drivers |
| /pa/paDrivers/set | char | 300 | Selects PA drivers |
| /tor/30W/freqCorTable | S32 | 927400, 233, 932500, 268, 937500, 286, 942600, 296, 947600, 300, 952600, 300, 957600, 294 | Parameters for linearization of the PA |
| /431/trxDrivers | char | platform4Csc xppGpio xppGpioResource dcOkSv2 commFpga7 … | List of driver instances to start |
| /hw/xppGpio/type | char | XppGpio | Identifier of the driver (normally class name) to instantiate and start for the driver instance xppGpio |

Table Examples of Database entries

Some keys used to retrieve data from the database are hard-coded in the Radio Application. Other keys, or key segments, are fetched from the database itself because the data is structured using a lot of indirection. An example is the driver data illustrated in the table above:   
- the driver set gives the segment to build the key for the driver list  
- the driver list gives segments to build the keys for individual drivers.  
Only the driver set’s key is hardcoded in the Radio Application.

At the start of the Radio Application the databases are merged, so the distinction between the production database and the SW database is transparent to the application. If a key occurs in the production database and the SW database, the value in the production database takes precedence.

More information about the development process and the contents of the SW database is given in ref. [13].

Ref. [38] presents the coding rules for the database keys and explains how the keys are handled in autoGet, see below.

#### autoGet

autoGet is a system of services to retrieve parameters from the database.

The main advantage of autoGet() is the ability to combine the partial search key, provided by the caller, with additional segments pertinent to the current environment, such as driver set, frequency band or output power, to select the most specific parameter. This releases the programmer from the detailed knowledge of the existing database entries and makes the Radio Application simpler and more re-usable.

The functionally reach autoGet system also provides calls to

* retrieve a single element of an integer array
* retrieve the number of elements in an integer array
* define the key as a sequence of segments which are concatenated in autoGet()
* fetch parameters conditionally if exists

## Resource layer

The Resource layer contains components with hardware independent logic and these components are intended to have a synchronization and orchestration role in the Radio Application. The items to be orchestrated are the different modules, with respect to global events coming from the O&M layer (DU signaling) or to distribute events originating from the modules (fault events as an example).

The main components in the Resource layer are:

* Fault Manager
* Fault Manager Interface
* Carrier Resource Handler
* Equipment Resource Handler
* Antenna Resource Handler

The intention with the Resource layer is to create an inter-module information highway, where the components can pass information between each other and that is needed for the modules.

Since the modules are not allowed to communicate directly, the components of the Resource layer can pass information to each other. To prevent the components to have dependencies on each other, an event passing framework is used, where the components can register themselves and subscribe for events. The owner of this framework is the Carrier Resource Handler, since most events originate from Carrier related functions in the O&M layer (northbound from the Resource layer point of view).

When a component in the Resource layer receives an event, the component will call the associated module or modules, using the module/s interface methods.

The information that will be passed shall be determined using configuration data through the database (both what events shall be passed and the receiver/receivers of the events).



Figure Resource Layer overview

In Figure 19 the overview of the Resource layer can be seen. The thick arrows represent the information flow between the different components. The thin arrows represent the dependency direction from the components in the Resource layer to the modules. Implicitly, this means that the control and data flow between the modules and the Resource layer components (such as function calls from the Resource layer components to the modules or indications sent from the modules to the Resource layer components) is always initiated by the components of the Resource layer (at startup for example). The modules then respond using callbacks for asynchronous calls.

The Resource layer components communicate using a framework derived from the Carrier Subscriber framework. This means that the users (i.e. the components in the Resource layer) of the framework subscribe to certain events and when these events occur, the subscribers are notified.

Table Event Subscriber Framework

|  |  |  |  |
| --- | --- | --- | --- |
| **Event** | **Carrier Resource Handler** | **Equipment Resource Handler** | **Antenna Resource Handler** |
| Baseband setup | Source | Subscriber |  |
| Baseband release | Source | Subscriber |  |
| Carrier setup | Source | Subscriber | Subscriber |
| Carrier release | Source | Subscriber | Subscriber |
| Carrier enable | Source |  | Subscriber |
| Carrier disable | Source |  | Subscriber |
| Carrier shutdown | Source |  | Subscriber |
| Carrier modify | Source | Subscriber | Subscriber |

In Table 4 the different components and their roles can be seen. The Carrier Resource Handler is (currently) the owner of the framework and therefore the other components will subscribe for the events. The rationale behind this is that most event origin from the Carrier domain (RAS).

Fault events, that trigger critical functions in some modules, are sent from the Fault Manager to the applicable modules (e.g. temperature supervision is handled by the Equipment Module. When a temperature fault is reported to the Fault Manager, the Fault Manager will call the applicable function in the Tx and Rx Modules).

### Fault Manager

The Fault Manager is the receiver and distributor of faults and fault triggered events in the Resource Layer. The Fault Manager subscribes for faults from the modules and can trigger events to the other components in the Resource Layer or directly to the modules, if necessary.

The Fault Manager is filtering the faults sent from the modules. This is done to reduce the number of faults reported in the northbound direction; to the O&M layer and to thereby the clients of the Radio Unit and to only send the most significant fault.

To do the filtering, the Fault Manager utilizes a huge set of dependency lists, where all available faults are represented. When a fault occurs, the Fault Manager will look up the list for the specific fault and check if a fault with higher precedence is already active. If so, the fault will be suppressed. If not, the fault will be set to active and reported to the O&M layer.

### Fault Manager Interface

The Fault Manager Interface has two key functionalities:

1. Translate the generic fault IDs events/faults reported by the modules through their Fault Advisor Controller- against their associated HW blocks to the current legacy fault IDs by embedding the module reported fault ID. Used by the different modules –through their Fault Advisor Controller. This approach will address the scalability issues associated support of increased numbers of TX & RX RF ports in the radio products.
2. Manage and coordinate the inter module fault dependencies in cases of a fault reported by one module require actions by another module.

### Carrier Resource Handler

The Carrier Resource Handler handles the configuration and control of the TX and RX modules. The Carrier Resource Handler contains Carrier Control, Carrier List and Carrier Status Coordination components. The Carrier Status Coordinator subscribes for blocking events from the TX and RX modules and can, if triggered, orchestrate blocking of the TX and RX. The Carrier Resource Handler also contains the high level governing component for time alignment of the downlink and uplink (Note! This (TA) is just a proposal. Time Alignment has not been thoroughly investigated yet).

The Carrier List contains references to the resources allocated and committed to by the modules, since the modules are responsible for and owns the resources and data associated with their respective domains. The Carrier List can therefore be seen as an accountant or a cache of the resources the modules contain.

### Equipment Resource Handler

The Equipment Resource Handler (ERH) deals with the Equipment Module. The major responsibilities for the ERH are:

* Single point of access towards the Equipment Module.
* Present interface(s) towards the OAM layer for equipment specific functionality.
* Synchronization between other resource handlers.

ERH controls the Equipment Module via the Equipment Module Interface and is in charge of any access towards the module. Any events (callbacks) from the Equipment Module will be handled by ERH and passed further according to the ERH interface definition.

### Antenna Resource Handler

The Antenna Resource Handler (ARH) deals with the Antenna Module. The major responsibilities for the ARH are:

* Single point of access towards the Antenna Module
* Present interface(s) towards the OAM layer for antenna specific functionality.
* Synchronization between other resource handlers and subscribing to the carrier list in CRH.

ARH controls the Antenna Module via the Antenna Module Interface and is in charge of all access towards the module. Any events (callbacks) from the Antenna Module will be handled by ARH and passed further according to the ARH interface definition.

## Radio Modules

This section describes the structure and scope of the different modules managing and controlling the radio HW blocks.

As per Radio reference architecture all module spans two layers, one is platform-dependent and another less platform-dependent.

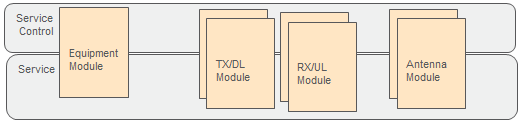


Figure Functional layer main components

The intention with showing multiple instances of the different modules in Figure 20 is to give the reader the vision of that one module represents an antenna branch each.

### Antenna Module

The Antenna module encapsulates the following functionalities:

* Antenna Calibration
* Antenna Feeder Power
* Antenna Power Handling
* Modem Control
* Antenna Supervision
* Antenna Fault Handling
* RETU

The Antenna Module is accessed via the Antenna Module Interface and the uses the Driver Interface (HALI1) to access HW drivers. Detailed description of the Antenna Module will be done in design description for the Antenna domain (which is currently under construction).

### TX/DL Module

The TX module encapsulates the following functionalities:

* Performance control
* Mean power limiter
* Crest factor reduction
* Supervision
* Power measurement
* Trace
* Frequency hopping
* Subband
* Filter
* Data
* Equalizer
* DAC
* TXL
* Clock control
* Time alignment
* PA control and supervision

The TX Module is accessed via the TX Module Interface and the uses the Driver Interface (HALI1) to access HW drivers. Detailed description of the TX Module will be done in design description for the TX domain, see [42].

### RX/UL Module

The RX module encapsulates the following functionalities:

* Time alignment
* Equalizer
* Frequency hopping
* Subband
* Frequency configuration
* Filter
* Interference measurement
* Supervision
* Analog configuration and control
* Clock control

The RX Module is accessed via the RX Module Interface and the uses the Driver Interface (HALI1) to access HW drivers. Detailed description of the RX Module will be done in the design description for the RX domain, see [43].

### Equipment Module

The Equipment SW Block contains functionality which can be associated with the physical unit such as handling of power line disturbances, MMI, external alarm ports, hotspot temperature measurements and SW compatibility lists.

Equipment Module will include:

* Internal DC/DC
* LTU
* Radio Application Startup
* External Alarm
* Hotspot Reporting
* MMI Handling
* SW Compatibility List Handling
* 48V Supervision
* Temperature Supervision
* TDD switching
* Internal link supervision

## Hardware Abstraction Layer

## The Driver Layer

The purpose of the driver layer is to identify the HW, start the correct device drivers and provide access to the HW through a set of well specified interfaces. The driver layer provides many different interfaces. In addition to the SW internal interfaces there are also shell commands and database parameters which serve as interfaces towards users mainly within the HW design areas. Every device on the board shall be properly initialized and preferably be accessible through basic shell commands, regardless of whether the device is in use or not.

The functionality of a device is published as a logical resource. The driver layer hides the actual HW topology from the user who will not need to know what devices there are on the board, which device that implements a specific function and where the devices are connected. The services can focus on configuring and using the resources rather than the devices.

HW changes that we aim to handle completely within the driver layer include: a second source device, a rerouted board or the integration of a function into an ASIC. The driver layer minimizes the SW impact when the HW components change but the functionality remains unchanged.

The driver layer executes in the context of the calling process and the driver interfaces are synchronous method calls. Driver interface operations normally complete quickly.

The driver layer publishes the HW resources that are actually available. There is no intent to hide or fake resources or in some other way distort the reality that would make it difficult for the service layer to manage the resources properly.

### Device Drivers

A device driver represents a physical component and there is a device driver for each specific component - unless a very simple, generic device driver can be used. A device driver is generally platform independent and heavily parameterized to be reusable on any board. The intention is to always strive towards platform independent device drivers to reduce SW complexity and increase SW reuse.

Device drivers handle access to the device including bus configuration, synchronization and device interaction. The device driver shall support multiple users and protect the integrity of the transaction.

There are no empty device drivers or stubs that emulate non-existing HW in order to make the HW look more generic.

### Database Usage

The database plays a significant role for the driver layer. It enables a data driven paradigm where specific information about products and boards is stored in the database instead of being hard-coded in the SW. The database enables reuse of device drivers and simplifies support of new product variants.

The database is used for:

* Selecting the drivers to be started on a specific board.
* Device configuration and initialization.
* Mapping of device addresses and I/Os.
* Mapping of physical devices to logical resources and the publishing of interfaces.
* Storing any other property that makes a device driver reusable.

### Structure

The driver layer is further organized in two tiers as shown in the picture below.

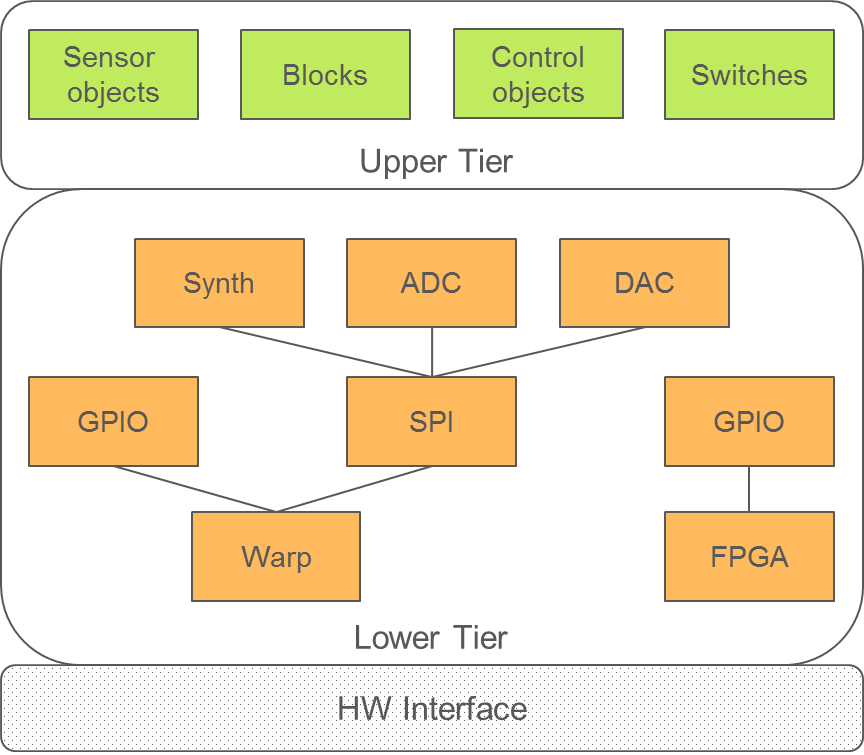


Figure Internal structure of the driver layer

The lower tier is composed of the device drivers, arranged in a dependency hierarchy. At the very bottom, there are drivers for the memory mapped ASICs and FPGAs. Above those drivers there are bus masters (e.g. SPI) and all the slave devices connected to the different buses (e.g. Synth, ADC).

The upper tier contains the mapping of the actual resources (inputs, outputs and blocks) and connects those to the device drivers in the lower tier. It is in the upper tier where RAW values are converted to human readable values and known types. It is also where common names are attached to resources before the resources are published to the upper layers.

### Driver Interfaces

The driver interfaces are a collection of many small interfaces towards different types of logical resources. The purpose of the driver interfaces is not to hide all the HW differences in our radios but rather to provide a stable interface as long as the HW functionality remains the same. New or changed functionality will generally spawn new interfaces and require new logic in the layers above.

The focus is on small interfaces where all functions are always applicable to the underlying HW. Smaller interfaces are more stable over time and the impact is limited when changes are required.

The granularity of the interfaces is that of the underlying HW resources in order to easily support addition or removal of HW resources by just creating the correct number of logical SW resources. Logical resources are divided into the very generic resource types and the more specific types called blocks. The picture below illustrates an example of the resources on a radio.

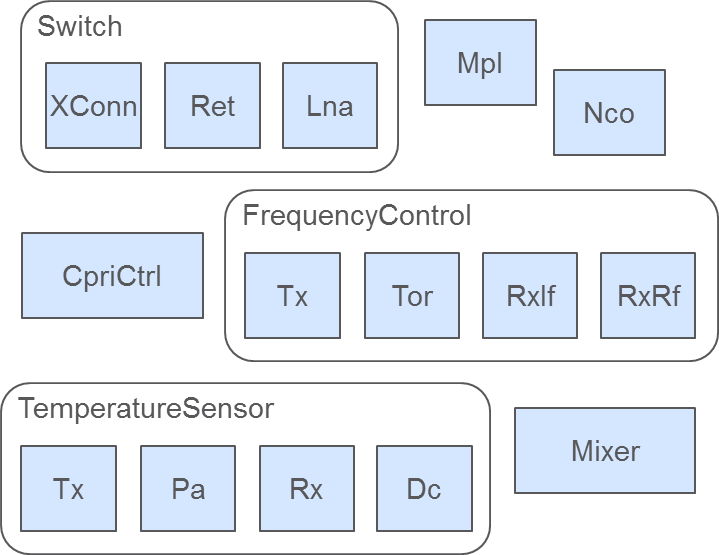


Figure Illustration of logical resources on a radio. Both generic types (Switch, FrequencyControl, TemperatureSensor) and blocks (Mpl, Nco, CpriCtrl, Mixer) are shown.

Resources are accessed by symbolic names that are common between different radio types, see (ref to Driver Interface IWD).

#### Generic Resource Types

Generic resources are highly generalized and have very stable interfaces with a small set of functions. Each resource type also has a set of shell commands that correspond to its SW interface.

The generic resource types are:

* Switches – resembles digital output
* Indicators – resembles digital input
* Sensors – provides measurements   
  e.g. TemperatureSensor, VoltageSensor
* Control objects – sets a value   
  e.g. FrequencyControl, VoltageControl, GainControl

#### Blocks

Blocks represent more complex resources which cannot easily be generalized and therefore have interfaces which are less generic and with more functions. The interfaces to blocks are stable once introduced but new versions of HW blocks are likely to spawn new versions of the SW interfaces as well, if there is a change in functionality.

Rather than extending a block interface with new functionality for new HW, a new version of the interface will be created that is applicable to the new HW. The old version of the block interface remains untouched and will still be used for the HW where it is applicable.

Block interfaces are used for most of the functionality in the radio ASICs and FPGAs, e.g. Mpl, Nco, Mixer.

Due to the nature of the blocks there are no generic shell commands for blocks.

# Concurrency view

This section covers the following:

* Describe which part of the system that can execute concurrently (i.e. maps functional elements to concurrency units)
* Describes Inter Process Communication (IPC) mechanisms including synchronization and coordination.
* Provides a process priority map with rationale.
* Describes system startup.

## View specific principles

The Radio Application compromised of a collection of loosely coupled processes of different priorities that interact with each other using message signals.

Specified timing for RU application initialization and startup, fault reporting, and carrier configuration and management is observed and maintained during all SW development activities.

Careful consideration is given when assigning process priority to ensure that critical Radio actions are handled without delay, at the same time ensuring that other processes are given fair time to run.

“[Thread] priorities are evil, don't mess with them.” / Joe Duffy

## View model

A generic rule of thumb is to consider the tasks that shall be performed by a thread. If the task is short and critical to perform, the priority shall be high. If a task is non-critical and takes long time to perform, the priority shall be low.

### Concurrency in Radio Application

The Radio Application is a concurrent system, where multiple threads execute simultaneously. Since the threads may interact in different ways (using a shared resource for example), caution must be taken to prevent events like deadlocks and starvation from happening. To do this, there are a number of different mechanisms to use.

#### Semaphores

A semaphore is used as a barrier to prevent two of more threads to execute the same code simultaneously. The first thread which enters the code (also known as a critical section) sets the semaphores, continues to execute and releases the semaphore when finished. Any other thread entering the critical section is blocked until the semaphore is released.

An important thing to consider is that the semaphore must be released when the critical section is passed, or the threads waiting to enter will be stuck at the semaphore.

#### Mutexes

A mutex works very much the same as a semaphore, but is thread sentient and can then implement priority inversion safety. This means that if a higher prioritized thread is waiting to grab the mutex, a lower prioritized thread (currently in possession of the mutex) will inherit the waiting threads priority to be able to finish its task and release the mutex.

#### Signaling

Signaling is an asynchronous way of inter-thread or process communication.

When an executing thread receives a signal from the operating system, the thread shall react to that signal in a defined way.

#### Active Objects

The active object is a design pattern to encapsulate thread handling. The active object separates the invocation and execution of the code belonging to the active object. With this means that the calling thread will not execute the code belonging to the object, but the thread within the active object.

### Event Concurrency

Concurrency in the context of RU SW modules refers to the ability of modules to process multiple end-user initiated carrier configuration signals effecting one or more ports in parallel.

The module concurrency mechanism revolves around the support of 2 data sets namely:

* Buffered data: Represents the data stored as events arrive
* Committed data: Represents the data used by the module once a cycle is triggered –this is also referred to as the working data-

In this pattern the Module main controller receives carrier events, and based on these events updates the buffered data as a list of Filter Branch Set (FBS) and will include also the FBS state.

Once the module is ready to consider new data, the committed data is updated, and this data will include both a list of FBS as well as a list of the associated Filter Branches (FBs).

### HW Drivers startup and initialization

The information about which drivers are necessary for a particular HW is available in the database as the combination of parameters *driver set* and *driver list*[[5]](#footnote-5).

The driver set is saved in the production database during the manufacturing process, when all the HW components on the board are determined[[6]](#footnote-6). The driver set is used to identify the driver list, to be started on the board.

During the Radio’s start-up procedure, the driver list is retrieved from the database and the drivers are instantiated and initialized in the order they appear in the list. The order of the drivers’ start-up is important since the HW components are initialized in a predefined sequence and the drivers build up a dependency hierarchy.

Class DrvFactory is used to instantiate the drivers and to maintain the list of all active drivers. If a driver is dependent on another one, it retrieves the reference to it from DrvFactory using its name for the look-up. Further on, the two drivers communicate using the interface published by the provider.

The Channels, Switches and Block Interfaces, in the upper tier, are placed last in the driver list to be started after the drivers in the lower tier have been initialized.

Class DrvFactory is also used to bind the providers in the Driver Layer with the users in the layer above.

### Module Startup and Initialization

Module Startup and Initialization is a onetime activity that takes place during the Radio start-up/restart.

Module startup either passes -which means that all services has successfully started and initialized-, else it is declared failed

Failure of a Module startup causes the failure of the Radio startup (restart in Boot mode)

Module startup and initialization uses the Service Factory Framework to create the services, and uses the Service Container Framework to register the created services.

#### Module Startup and Initialization Steps

* Query the Module service list from DB for running platform
* Create the services
* Register the services in the Service Container
* Pre-initialize each service I
* Post-initialize each service
* Save Module services in the Module
* Register the Module shell commands
* Save the Module Service Container in both mmModCmd & mmModule (used to locate services) -mm is the module type e.g. Tx

# Deployment view

This section covers the following:

* Describes the different deployment forms (e.g. AUBOOT etc.) supported.
* Describes dependencies to Execution environments, Middleware services, and external dependencies including third party software.

## View specific principles

Meeting the execution environment constraints

The Radio Application is deployed on target HW with limited resources in terms of RAM and flash.

The characteristics of the Radio Application deliverable are to meet the HW limitations. The size of the Radio Application deliverable is controlled by including/excluding the FPGA binaries and filtering the database. The required RAM size is limited by linking in a subset of modules.

## LMC

As the final step in the development process, the Radio Application is compiled and linked to create a *load module*, which is suitable for loading into the board processor and executing on the Radio. The load module is formatted according to the rpdout format, see ref. [14].

The Radio Application is packed and delivered, together with several other files, in a SW deployment package known as the Load Module Container, LMC-see ref. [20]-.

The LMC contains all loadable SW necessary to configure and control the Radio board, including the operating system, middleware, the Radio Application and binaries to load onto the FPGA s.

Also included another LMC component which is used to unpack the LMC from flash to RAM and start the execution of the application. This component, known as the XP Program Loader, is included as the first load module in the LMC. It is started from flash by the HW-resident bootstrap IBOOT when the Radio board is booting, to perform its task and disappear.

The LMC contains the components shown in the table below.

|  |  |
| --- | --- |
| **Product name** | **Comment** |
| XPLLM1 | XP Program Loader. Binary image to be started directly from flash. Unpacks the LMC and loads and starts the other executable load modules. |
| XPPLM4 rpdout | Basic XP platform: OSE Delta components (kernel, Link Handler, Real-Time Clock, shell), Memory Handler, Man-Machine Interface, Board Support Package, XPAI server, basic CBCI-AU server, etc. |
| XDPLM2 rpdout | XP support for DP: Link handler, chip select, interrupt control, trace and error, XDAI server, UART support, HDLC primary station, ATFI server. |
| RUS rpdout | Radio application. |
| TrxCtrl | Binary to be loaded on the DPD FPGA. Additionally, a data file used at initialization of the DPD algorithm. |
| ComCtrl | Binary to be loaded on the communication FPGA; support for CPRI and modem. More than one file may occur. |
| Xrus.bin | SW database in binary format |
| Compatibility list | See 8.2.1.1 |

The LMC is currently built to include several different products targeting different HW: FDD Radio Platform 4, FDD Radio Platform 5 and TDD Radio.

Each of these products is further built for three variants: Radio Application –see 8.2.2-, Pre-Installed SW –see 8.2.3-, and AUBoot –see 8.2.4-. Some of these variants may be excluded. E.g. Radio boards without requirement for backward compatibility -i.e. has the ability to be used with old RAN SW- do not need the Pre-Installed SW.

All the builds use the same Radio application code, but will link in only the drivers necessary for the target HW, thus reducing the memory footprint of the application.

The LMC size is controlled by only including the FPGA binaries relevant for the targeted platform.

**The rationale for building so many different LMC forms is to economize on RAM and flash.**

#### The compatibility list

The compatibility list is a list of older LMCs which the current LMC is compatible to and can replace. The list is formatted as an XML text file. The replaced LMCs are shown through the product number and a range of R-states, e.g.:

<ListItem>

<ProdNbr>CXP9013268/6</ProdNbr>

<StartRev>R14ZA01</StartRev>

<StopRev>R48ZD</StopRev>

</ListItem>

<ListItem>

<ProdNbr>CXP9013268/99</ProdNbr>

<StartRev>R14ZA01</StartRev>

<StopRev>R48ZD</StopRev>

</ListItem>

Early during the start-up procedure, the Radio Application collects the compatibility lists from the LMCs residing in different flash slots, including the Pre-Installed SW, concatenates them and transfers to the client. The client compares the list with the LMC available in the RBS Upgrade Package and selects the LMC to be started on the board.

See also refs. [3], [4] and [15] for further details on the compatibility list handling.

### Radio Application

The Radio Application is the regular variant of the Load Module Container. It is the most popular variant in terms of the release frequency. The Radio Application is built for a particular HW platform to minimize the included drivers and FPGA binaries.

The Radio Application is delivered to the RANs: GSM, WCDMA, CDMA, and LTE, where it undergoes thorough testing against each RAN’s own SW. Ultimately it gets released as a part of the RAN’s SW distribution package, which in WCDMA and LTE is called the Upgrade Package, UP.

The Radio Application is delivered to the base station separately from the HW and is loaded onto the Radio board before it can be used. This is deemed the preferred way of distributing the Radio Application , as it decouples the HW and SW deliveries, and makes it possible to implement new functions, deliver error corrections, and deploy them through new SW releases.

### Pre-Installed SW

Technically, the Pre-Installed SW, PIS, is exactly similar to the Radio Application. PIS is, however, distributed as a type 2 SW: it is factory-loaded into slot 1 in flash, made protected, and delivered together with the Radio HW.

PIS is loaded into RAM and started if there is no other suitable Radio Application LMC available in the base station. This happens when a new Radio board is installed as a spare part in an older base station, which lacks newer SW releases. In such a case, the PIS’s backward compatibility is utilized to collaborate with an old client.

Some boards are never used in the above scenario and the PIS is not necessary. In such a case, no PIS is developed and installed and the corresponding verification effort may be saved.

Further details of PIS handling, especially from the client’s perspective, are available in refs. [3], [4] and [15].

### AUBoot

AUBoot is a minimized variant of the LMC whose task is to assist the client with loading and starting the required LMC. AUBoot is delivered together with the HW as a type 2 SW. It is the first SW started on Radio after power-on and cold restart.

AUBoot needs to support a limited subset of O&M procedures including

* Fetching board capabilities
* Acquiring resources (multi-client locking)
* Fetching HW PID
* All procedures concerning the LMC (inventory list, fetching the running LMC identity, loading, selection, deleting, retrieving the compatibility list?)
* MMI/LED handling
* Setting real time clock

Since no radio resources are activated by the AUBoot, only the drivers to load the COM FPGA and to setup the LTU and the synthesizers are linked into this variant of the Radio Application. AUBoot is the smallest LMC and requires the smallest flash slot.

AUBoot is an old concept and originally contained only the XP platform SW. With the advent of PIS, AUBoot might be abandoned, but it requires changes on the client side.

## Flash

Describes parts: IBoot, Board Parameters, User Area (Production DB, Event Log), AUBoot (LMC0), PIS (LMC1), LMC2, LMC3?

Flash is a non-volatile memory that can be erased and reprogrammed. The content of flash is preserved over board restarts and power-off periods. Before a new content can be stored in flash, the old one is erased. Erasing flash is done in blocks of 128 kB and is time-consuming. In the Radio board, flash is used to store the Radio Application and various data.

The size of flash, its usage and split into slots, differ between HW variants, and Radio platforms. The table below shows an example of the flash usage in Radio Platform 5. More information about flash and its usage, including differences between platforms, may be found in ref. [21], [25] and [26].

The both the basic SW necessary to bring the board into operation, and the production database are downloaded into flash during the manufacturing process and made write-protected. The corresponding flash slots are called *Type 2* slots. Modification of such type of slot is treated as a HW change and requires stepping the R-state of the board. Slots which are modified during the normal operation are called *Type 3*.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Provider** | **Comment** |
| LMC0, AUBoot | 2 | Radio appl. | See ch. 8.2 |
| LMC1 - PIS | 2 | Radio appl. | See ch. 8.2 |
| LMC2 – Radio Application 1 | 3 | Radio appl. | See ch. 8.2 |
| LMC3 – Radio Application 2 | 3 | Radio appl. | See ch. 8.2 |
| Event Log | 3 | Radio appl. | See below |
| Production Database | 2 | Factory | See ch. 6.6.1 |
| HW Log | 3 | Radio appl. | See below |
| Board Parameters | 2 | XPP | See ch. 8.3.1 |
| Node Identity | 3 | XPP | See below |
| ABoot | 2 | Radio appl. | See below |
| IBoot | 2 | XPP | See below |

The number and size of slots in flash are controlled by the XP platform, while the Production Database and the Event Log, which belong to the *user area* are controlled by the Radio Application.

Slots LMC2 and LMC3 holds the Radio Application downloaded from the DU. One of these slots is overwritten with the new SW during the RBS upgrade process while the other one is preserved as a fallback in case the RBS upgrade fails.

**The Event Log** contains persistent trace of important events in the unit such as board restart, client connection or rejecting a client request. It is mainly used for trouble-shooting. The Event Log is organized as a circular buffer with new entries overwriting the oldest ones when the log gets full.

**The HW Log** contains information about HW faults detected by the board’s supervision functions. After detection of a HW fault, the board is sent to the factory for repair where the HW Log is used to identify the malfunctioning components.

The Node Identity is used by WCDMA to store the RBS’s unique identity. It is distributed to all replaceable units within the node to allow for replacement of one or several units without losing the identity. The identity is used in the licensing system: the licenses provided by Ericsson are tied to an individual RBS using its unique id.

The IBoot (Initial Boot) is the bootstrap loader initiated at start/restart of the Radio board. It contains the executable binary image, hard-wired at the CPU’s reset vector. IBoot locates and starts one of the LMCs in flash. After cold restart it is LMC0 and after warm restarts one of the others LMCs, selected by the client.

The ABoot (Application Boot) if exists, is called from the IBoot early during the board start-up. It is used to initialize HW components controlled by the Radio Application. The ABoot is seldom used because the HW reset is distributed to all the components and sets them into idle states. Sometimes, however, early initialization using SW is necessary, e.g. to avoid spurious transmission, and it is implemented with ABoot.

### Board parameters

The board parameters is a collection of persistent, board-specific parameters maintained by the XP platform in a dedicated flash area, see ref. [21]. The parameters are read by the XP platform and presented to the Radio Application as environment variables. Board parameters play a similar role for the XP platform as the database plays for the Radio Application.

The XP platform uses the majority of the board parameters to control the communication buses General I/O, HDLC and I2C. Other parameters describe the layout of flash. The Radio Application uses only the address of the user area in flash from this group.

Another group of parameters maintains the board’s product number, R-state, name, production date and serial number. The Radio Application passes these parameters to the client during the start-up handshake and uses them in traces.

## RAM

Describes organization of RAM: XPP+RICR (5MB), Radio Application (heap, pool, extended pool? etc.), Memory-Mapped I/O

# Development view

This section covers the following:

* Describes the code-line organization and module organization.
* Describe the source code version handling including track strategy.
* Standardization of design testing, i.e. what type of tests to implement and the principles behind them.
* Reference to design rules.

References [6] and [7] give more details in some areas covered by this chapter.

## View specific principles

One source for all HW

The Radio Application consists of one source code base, able to support all past and present variants available of the global Radio HW. Depending on the changes made to the hardware, the need for change is different for the Radio Application. Very small changes, for example pure parameter changes, are dealt with through the database. Bigger changes require changes in the source code, which may span from the drivers up to O&M layer, but the intention should be to minimize the scope of the changes. The Radio Application is built up with the help a number of frameworks, which are there to support the possibility to add new features in an as non-invasive way as possible. Instead of just modify existing classes and components, the designer should seek to investigate if the code base instead could be extended, to accommodate the wanted changes.

Structured code

The Radio Application code shall be divided into blocks with well-defined interfaces. A block interface shall be exposed as a collection of header files placed in a separate directory. Blocks are only allowed to access each other’s interface directory; there shall be no other dependencies. In this way, the dependencies may be controlled and limited, making the blocks suitable for parallel development.

Object-oriented development with C++

The Radio Application shall be developed using C++ and object-oriented programming. This gives access to both high-level design concepts, like design patterns and inheritance, and low-level HW access, necessary in embedded SW. Other important advantages of C++ are high execution performance and the popularity in the programmer community.

The test suits shall be developed using languages suitable for the verification domain. These languages shall facilitate easy programming of event sequences.

Maintainability

The Radio Application code shall ensure good maintainability. The code constructs shall be simple and clear. The implemented functionality shall be based on identifiable requirements. The code shall be well-commented to ease its understanding for a new designer.

Testability and observability

Each class, or a collection of classes, shall be accompanied by test code to execute the basic test. The code shall be instrumented with traces so that the internal execution sequences may be observed and recorded.

## View model

## Design rules

The Radio Application shall be implemented with respect to the design rules on the LMR System baseline (which can be found in [40]) and the RA Radio C++ coding rules [39] .

## Source code structure

The Radio Application consists of the code owned by the RA Radio and external components owned by other units.

### RA Radio source code

The RA Radio SW resides in a repository and is version handled. The structure reflects the static architecture, with folders for every major component. The components are:

* O&M
* TX
* RX
* Equipment
* Antenna
* Drv
* CommonFW
* ResourceControl

Each SW block is further divided into smaller units, possibly in several levels. Every unit contains directory “inc” for header files, “src” for the implementation code and “unitTest” for the test code.

Each SW block has a top-level subdirectory “if” containing its public interface. The directory contains the header files to be used in other blocks. The blocks are only allowed to access each other’s directory “if”. This gives control over the interdependencies between the blocks.

Figure Source code directory structure



The goal of the directory structure is to organize the code in blocks corresponding to the application layers. In this way the code is split into relatively independent parts, to facilitate parallel development. The existing code occasionally breaks this principle as a result insufficient SW architecture maintenance and imperfect understanding of the concept.

#### Module Directory Structure

Module directory structure follows the Radio Application directory structure and naming convention and includes the following directories: if, inc, src, unitTest, Makefiles, etc…

Module directory is located off the main application directory (sw/app) and it includes all Module source code, test code, and documentation related to the module.

The Service directory naming convention follows the following pattern: ***<service name abbreviation>*Service (e.g. mplService)**

Each module defines two External Interfaces as follows:

*a. module*Adapter: Provide a link/glue between legacy Carrier Control and the Module main controller. The module adaptor is responsible for the registration to Carrier Control configuration events, and passes these events to the module main controller through the module interface.

The adaptor is also responsible for passing back calls and indication signals to the upper legacy layers that are initiated by the module controller.

This interface is deemed temporary and is expected to be removed once the RU SW modularization exercise is fully completed.

*b. module*ModuleIf: Provides the north-bound interface towards the Module. Upper layers use this interface to communicate with the Module control service

### External components

The Radio Application is dependent on several external SW components and interfaces residing in different repositories.

WCDMA O&M interfaces:

* CBCI RU, RRU, RUW, FU
* CDCI TR, TPA, DPCL, AI, ALM
* BLIB (I/F components and signal base

LTE, GSM, CDMA O&M interfaces:

* TR DCI, ANTPI
* BCI, I/F components and signal base

Inter-device interfaces:

* IDCI (WCDMA interface for ROT measurements)

Platform components and interfaces:

* XDAI, XPAI
* A2CI, ATFI
* FPGA I/F
* LTUI
* RICRAI
* EUL ROT estimator
* C/C++ library, STL, Execution Environment

## Version handling and branching

Optimally, the Radio Application should be developed in a single track, because maintaining every additional track involves considerable resources. In such a concept, all development and TR corrections, and all deliveries, occur at the track head. This strategy has proven unrealistic because the development at the track head is dynamic and it is difficult to make this code fully stable.

Therefore, Radio Application is maintained in several parallel tracks. All new development is done in the *main track*. In time before a product release, a *stabilization track* is branched out. This track is subject to extensive testing and error corrections until it reaches the quality sufficient for the customer delivery.

Defects detected in a stabilization track are also corrected in the main track and other stabilization tracks.

The stabilization tracks are not intended for function growth, only for the quality assurance activities. However, in urgent cases new functions are added to these tracks because branching out a new track from main, and stabilizing it, would take too long time.

A stabilization track should be closed when the following stabilization track reaches release quality. In practice, stabilization tracks are maintained longer, e.g. to tailor deliveries for prioritized customers.

Apart from the main track and stabilization tracks, teams and individual designers use branching code for development activities, e.g. to facilitate parallel development of multiple features in the same code base. These tracks are normally short and ultimately merged to the main track. Currently, there are no general rules on how team branching should be handled. Some teams make local copies of the SW, carry out the development on them and, finally, merge the modified files back to the main track

## Test strategy

The test strategy on RA Radio can be summarized to the following:

* Focus on hardware generic and standard independent tests, as the RA is responsible for radio software development and multi standard RBS features.
* Use of test driven development to facilitate finding errors in early phases to ensure efficiency and fair cost of testing
* All testing shall be integrated part of the feature development in the cross-functional team to reduce hand overs.
* A fully automated legacy feedback loop shall be maintained on radio subsystem level to constantly assure quality.

More information about the RA Radio test strategy may be found in ref. [8]. The test strategy on the PDU level is described in ref. [9].

### Verification levels

The verification is done on the three levels, where each level targets different concerns:

* Radio Application design level
* Radio subsystem level
* Node level

#### Radio Application design level

Principles valid for this level:

* Full automation and ease of use to make daily work more efficient
* Development of the RU SW is based on design input specifications (DS, IWD etc.) thus the testing shall also be based thereof
* Use of software quality rank (SQR) to get measurable quality.

Types of tests included:

* + Static test (reviews, static analysis, e.g. with Coverity)
  + Component test (a.k.a unit test), a white-box test run on host.
  + Radio subsystem integration, executed on target, on a wide selection of HW to cover all platforms, driver sets, frequency bands etc.

The Radio simulator may be used in the test activities, especially for debugging.

Apart from functional tests, the design level tests also cover heap, pool and CPU usage. A tool is used to detect memory access errors.

#### Radio subsystem level

Principles valid for this level:

* Testing shall be done from a black-box perspective to make sure the product black-box requirements are met.
* Cover all radio sub-system functional and non-functional requirements (100% sub-system requirements coverage)
* Execution shall be fully automated for efficiency

Types of tests included:

* + Functional verification
  + Radio performance verification
  + Characteristics’ verification
  + Regression test of legacy functionality

The test environment for these tests contains the target Radio HW and the target antenna HW, and simulated Digital Unit.

#### Node level

Principles valid for this level:

* Testing shall be based on operator procedures, including planning, preparation, and deployment of the test configuration.
* Execution of test cases and monitoring of the multi standard RBS as a black box
* Built in observability mechanisms shall be used

Types of tests included:

* + Target errors that are difficult to predict and that allows the system to be perceived as unstable e.g. real-time effects and interference between different parts of the system which may cause hanging resources, degradation of services, intermittent alarm, and system downtime.
  + All impacted RAN standards (C/G/W/L) are covered including both single and multi-standard aspects. With ‘L’ is meant both TDD-LTE and FDD-LTE.

The node level tests are executed on the RBS target HW with simulated neighboring nodes: Ue, RNC, BSC, CN nodes, etc. The RBS configurations are defined by CIRV and consist of multi-standard RBSs equipped with a selection of Radio and antenna unit variants. The selected test cases correspond to the RBS use cases dependent on the Radio functionality.

After this test level, the Radio Application is delivered to CIRV where further tests are performed, e.g. interoperability tests, where other nodes are added to the test environment, and all tests requiring the real traffic load. The long term goal of node level tests is to confirm that all faults are found in earlier phases.

### Test environment and automated tests

The test environment consists of:

* The Radio simulator to run tests on host (primarily for white box tests).
* A selection of Radio boards connected to a controlling unit, to run tests on stand-alone Radio target HW (for white and black box tests).
* A selection of RBS configurations equipped with Radio HW, to run the node-level tests (primarily for black box tests).

Automated tests may be pre-scheduled or run on demand. For the scheduled tests, the Radio Application build starts at fixed intervals and captures the earlier SW deliveries. A successful build is followed by a test suit, whose results are stored for a later analysis. When debugging, a similar sequence, but with customized test cases and traces, is executed on demand.

# ETSW

It is possible to build a special version of the Radio Application for production test. This chapter will contain information about the production test software ETSW.

# RSWoS

The Radio SW on Stub (RSWoS) consists of the higher layers in Radio Application stubbed at the Service Control layer, XCS, and simulated RICR. Within Radio Application, it stubs HW specific parts.

RSWoS is used with C2RS (C2 Radio Simulator) to perform Baseband and node tests without radio boards.

RSWoS must use identical code to the HW agnostic part of Radio Application (OM, Resource, and Common layers). There should be no #ifdef / #ifndef related to RSWoS in this code.

At startup, Radio Application reads the database to decide if it must instantiate stubbed service controllers and services instead of real ones.

# Terminology

***Please note that this chapter is preliminary! It has not been reviewed and should therefore not be seen as canon.***

## Service

A *Service* in the context of the Radio Application Software is one part of the software representation of a certain *Functional Block*. The *Functional Block* may be represented by both hardware and software, i.e. they form the *Functional Block* together. A *Service* shall be hardware aware in the sense that it must know how the hardware works as the *Service* implements the behavior of the *Functional Block*. The access to the hardware, from the *Service* point of view, is abstracted by the *Driver*. The *Driver* provides a Resource Handler or Resource Object the *Service* will use to perform the behavioral control of the hardware.

A *Service* shall implement one or more interfaces which present the methods available for the *Controller* to use. The methods shall be separated and put into different interfaces according to what type of functionality they present, i.e. control and supervision.

A *Service* may be composed by one or more classes and can be referred to as a *Component*.

## Controller

A *Controller* is a *Component* which has the responsibility to orchestrate and control *Services*. The *Controller* shall not implement logics that should reside in a *Service*, hence the *Controller* is more decoupled with the hardware but may still be depending on the hardware it runs on (for example, the *Controllers* between two different hardware deployments could differ in what type of *Services* the hardware presents).

## Driver

A *Driver* provides the *Service* with the means of accessing a hardware component. The *Driver* shall not contain business logic; that is being done by the *Service*. One reason for decoupling the access from the logic is to be able to offer early access to the hardware design verification.

## Component

A Component is class or a set of classes with high coherency. Instead of introducing yet another definition for “component of components”, such as “Package”, the component can be comprised by *Components* (or sub-*Components*).

## Module

A *Module* is a set of *Components*; *Services* and *Controllers*. A *Module* is a governing instance for a major part of the hardware and is the level of granularity chosen to present the functional abstraction of a radio.

The Radio Application contains, but is not restricted to, the following *Modules*:

* TX Module
* RX Module
* Equipment Module
* Antenna Module

Every *Module* is responsible for a certain part of the Radio.

A Module consists of one or more *Controllers* and the *Services* needed to control the *Functional Blocks* associated with the specific domain.

## Functional Block

A *Functional Block* can be comprised by software or hardware only, but is most often an aggregation of hardware and software and consists of four parts; hardware component, software *Driver*, software *Service* and configuration data (in the database). If the *Functional Block* is implemented on a single device, the *Driver* and the *Service* will reside in the same domain. In a distributed system (such as the RDS), the *Functional Block* may be distributed between the different systems, for example the Service may reside on one part of the system, while the *Driver* and hardware component may reside on the other part of the system.

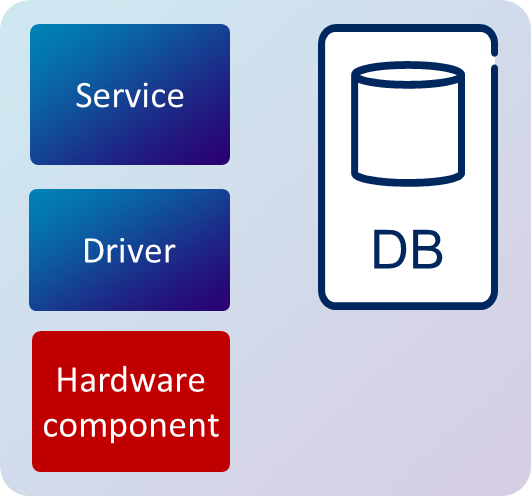


Figure Functional Block concept

The granularity of a *Functional Block* is mostly related to a certain hardware component, e.g. a local oscillator, a group of hardware components implementing a certain function, e.g. a power amplifier or, for ASICs and FPGAs, a function the ASIC or FPGA implements. In the case of the *Functional Block* “local oscillator”, the *Functional Block* consists of a hardware component (the physical oscillator), a software Driver (providing access to the physical oscillator) and a software *Service* (defining the behavior of the physical oscillator).

# Abbreviations

3GPP 3rd Generation Partnership Project

A2CI Auxiliary unit class 2 Communication Interface

ADC Analog to Digital Converter

AI Antenna Interface

ALM Alarm

ANTPI Antenna Port Interface

APC Alarm, Power, Climate

ARH Antenna Resource Handler

ASIC Application-Specific Integrated Circuit

ATFI Auxiliary Transport Function Interface

BCI Board Control Interface

BLIB Base Station Library

BSC Base Station Controller

CBC Central Board Control

CBCI Central Board Control Interface

CCIE Common Communication Interface Engine

CDCI Central Device Control Interface

CDMA Code-Division Multiple Access

CFHE Common Fault Handling Engine

CIRV Continues Integration and Release Verification

CN Core Network

CPRI Common Public Radio Interface

CRH Carrier Resource Handler

DAC Digital to Analog Converter

DCI Device Control Interface

DP Device Processor

DPD Digital Pre-Distortion

DPCL Downlink Power Clipping

DU Digital Unit

ERCI Equipment Resource Control Interface

ERH Equipment Resource Handler

EUL Enhanced Uplink

EVM Error Vector Magnitude  
FDD Frequency-Division Duplex

FPGA Field-Programmable Gate Array

FM Fault Manager

Git code version handling system (not an abbreviation)

GSM Global System for Mobile Communications

HAL Hardware Abstraction Layer

HDLC High-level Data Link Control

I2C Inter-Integrated Circuit interface

IC Inter-Connect

IDCI Inter-Device Communication Interface

LED Light-Emitting Diode

LMC Load Module Container

LRCI Logical Resource Control Interface

LTE Long-Term Evolution

LTU Local Timing Unit

LTUI Local Timing Unit Interface

MBSE Model Based System Engineering

MMI Man-Machine Interface

NCI Node Control Infrastructure

O&M Operation and Maintenance

OSE Operating System Embedded

PA Power Amplifier

PIS Pre-Installed Software

RAN Radio Access Network

RAS Radio and Sector  
RBS Radio Base Station

RF Radio Frequency

RIC Radio Interface Control

RICRAI Radio Interface Control Radio Application Interface

RNC Radio Network Controller

ROT Raise Over Thermal

SPI Serial Peripheral Interface

STL Standard Template Library

TBD To Be Defined

TN Transport Network

TDD Time-Division Duplex

TMA Tower Mounted Amplifier

TPA Transmit Power Amplifier

TR Transceiver

TRX Transceiver

TTM Time To market

UART Universal Asynchronous Receiver/Transmitter

Ue User equipment

VSWR Voltage Standing Wave Ratio

WCDMA Wideband Code-Division Multiple Access

XDAI XP with DP support Application Interface

XP Auxiliary Processor

XPAI XP Platform Application Interface

XPP XP Platform

# References

1. Characteristics Requirement Specification for RBS, 24/1056-HRB105102/1
2. O&M timing requirements overview, 15/0363-FCP105361
3. Use Case Realization Equipment Handling, 1/155 60-CSH 103 701
4. Capability Realization: ICBR Equipment Startup and Initiation, 534/155 16-HRB 105 102/1 (N.B. This document does not exist in eridoc. It has to be generated from the MBSE model of the RBS.)
5. RU SW Real time requirements, 10/0363-FCP 105 361
6. RU Software Description/Handbook, 1/1551-CAH 161 1490
7. RU Software Makefile, EAB/FJR-11:0243
8. RA Radio Test Strategy, 1/1524-FCP 105 361
9. PDU LTE & MS RBS Test Frame Work, 9/1570-5/FEA 101 8211
10. RU Software Description/Handbook, 1/1551-CAH 161 1490
11. Driver Interface Intro, [ericoll link](https://lte-wiki.rnd.ki.sw.ericsson.se/mediawiki/index.php/File:Board_level_Radio_SW.ppt)
12. Database Update Description 850, RRUS12 B5, 152 12-KRC 161 321/1
13. WoW: RU External Database, Handling and Responsibility, EAB/FJB-12:0087
14. Loading Format for C Programs in RP/EMRP
15. FS Multi-Client handling of Radio, 6/155 17-HRB 105 602
16. Target Multistandard RBS System Architecture Description, 1/1551-CSH 103 6000
17. LTE RAN System Architecture, 1551-HRB 105 500
18. WCDMA RBS Node, System Description, 1551-HRB 105 102/1
19. RUS Functional DS Overview, 2/102 62 CNH 160 6510
20. Product structure SubSys Radio SW, 1315-CRH 109 0526
21. XP Application Dependencies, 8/155 19-CEH 101 90/1
22. Common Communication Interface Engine, 102 62-CAH 161 1437/2
23. Implementation Proposal for Radio SW, CCIE modules, 7/159 41-FCP 105 360
24. Fault Handling – Above R-ERCI/R-LRCI, 37/102 02-CNH 160 6510
25. Type2 SW, 60/102 62-CNH 160 6510
26. Event Log DS, 76/102 62-CNH 160 6510
27. DL time alignment for RU, 38/102 62-CNH 160 6510
28. Mean Power Limiter, 28/102 62-CNH 160 6510
29. DL Equalizer, 44/102 62-CNH 160 6510
30. DL Data, 20/102 62-CNH 160 6510
31. DL Filter, 62/102 62-CNH 160 6510
32. RUS Functional DS overview, 2/102 62-CNH 160 6510
33. RU SW Modularization Fundamentals, 1/102 68-FCP 120 6753
34. [Tx Branch Control Overview](https://ericoll2.internal.ericsson.com/sites/radio_01/ra_radio/Training/_layouts/PowerPoint.aspx?PowerPointView=ReadingView&PresentationId=/sites/radio_01/ra_radio/Training/Documents/TX%20module%20(WP%201186)/Tx-Branch-Control-Overview.ppt&Source=https%3A%2F%2Fericoll2%2Einternal%2Eericsson%2Ecom%2Fsites%2Fradio%5F01%2Fra%5Fradio%2FTraining%2FDocuments%2FForms%2FAllItems%2Easpx%3FRootFolder%3D%252Fsites%252Fradio%255F01%252Fra%255Fradio%252FTraining%252FDocuments%252FTX%2520module%2520%2528WP%25201186%2529%26FolderCTID%3D0x012000140AE022FC5E674DB9E0B3E1FE734799%26View%3D%257b669DE334%2DED73%2D4423%2D9005%2DB4E974B874EE%257d&DefaultItemOpen=1)
35. DL Frequency Hop 56/102 62-CNH 160 6510
36. DS TX Frequency Configuration 26/102 62-CNH 160 6510
37. NDS for Radio Software Modularization 1/102 68-FCP 120 6753
38. RU DB Design Rules, [3/0363-FCP 103 99 83, PA4](http://erilink.ericsson.se/eridoc/erl/objectId/09004cff875e24ad?docno=3/0363-FCP1039983Uen&action=current&format=msw8)
39. C++ Code design rules for Radio SW, [EAB/FJB-07:0091](http://erilink.ericsson.se/eridoc/erl/objectId/09004cff8746886f?docno=EAB/FJB-07:0091Uen&action=approved&format=msw8)
40. Radio SW Baseline, [1/109 03-CRH 109 0536](http://cdmweb.ericsson.se/WEBLINK/ViewDocs?DocumentName=1%2F10903-CRH1090526&Latest=true)
41. Device Handling Interfaces, [2/102 60-CSH 109 999](http://erilink.ericsson.se/eridoc/erl/objectId/09004cff86a78d8b?docno=2/10260-CSH109999Uen&action=approved&format=msw8)
42. TX Module Design Description, [4/159 61-CNH 160 6510](https://erilink.ericsson.se/eridoc/erl/objectId/09004cff88b5b867?docno=4/15961-CNH1606510Uen&action=current&format=msw12)
43. DD Radio SW RX Module, [3/159 61-CNH 160 6510](https://erilink.ericsson.se/eridoc/erl/objectId/09004cff88a9b0d1?docno=3/15961-CNH1606510Uen&action=current&format=msw12me)
44. Impact of NGR G2 on Radio SW Architecture [1/1551-CRH 109 0526](https://erilink.ericsson.se/eridoc/erl/objectId/09004cff8912eb92?docno=24/0363-100/FCP1039665Uen&action=current&format=msw8)
45. Device Handling Interfaces Design Rules, [2/10260-HRB 105 102/1](https://erilink.ericsson.se/eridoc/erl/objectId/09004cff872eb286?docno=2/10260-HRB105102/1Uen&action=approved&format=filemakerpro)
46. Device Handling Interfaces, [4/10260-HRB 105 700](https://erilink.ericsson.se/eridoc/erl/objectId/09004cff86a80b0d?docno=4/10260-HRB105700Uen&action=current&format=msw8)

# Appendix

\* List of stakeholders (defining key stakeholders and their key concerns)

\* SoC with architecture mapping?

1. See 6.2.1.1.1 for details of controllers and sub-controllers connections to services [↑](#footnote-ref-1)
2. de-bouncing of the same fault belongs to the supervision service/sub-component [↑](#footnote-ref-2)
3. mm refers to the module type e.g. tx, rx, etc… [↑](#footnote-ref-3)
4. WCDMA in table 2 reflect the current [a.k.a legacy] WCDMA interfaces. For DUS G2 BCI & TR-DCI will be used for WCDMA [↑](#footnote-ref-4)
5. Strictly speaking, the driver set is given by two database parameters named “/board/trxDrivers/set” and “/board/paDrivers/set”. The *values* of these parameters give names of the corresponding driver lists. [↑](#footnote-ref-5)
6. The factory selects among equivalent HW components based on the supply. Generally, two equivalent HW components require different drivers. [↑](#footnote-ref-6)