OCUDU Dev Guide

Generated 2026-09-05 from 58fa6ca

Developer Zone

Everything you need to contribute to OCUDU: code style, logging conventions, testing requirements, and the full merge request process.

FIRST CONTRIBUTION?

Read the Code Contribution Guide for an overview of the process, then check the GitLab issue tracker for issues labelled `good first issue`.

Architecture

Architecture Guide

Vision, design principles, and implementation patterns for the OCUDU codebase.

Codebase Guide

Module-by-module breakdown of the OCUDU codebase: CU-CP, CU-UP, DU-high, DU-low, MAC, RLC, and the scheduler.

Code Standards

The conventions that keep the OCUDU codebase consistent and readable. Familiarise yourself with these before opening a merge request.

C++ Code Guide

Language choices, library usage, source structure, naming conventions, and commit message formatting.

Logging Guide

Log levels, subsystem tags, and formatting rules for the OCUDU logging framework.

Testing Policy

Test coverage expectations for each type of contribution: unit, integration, and system tests.

Contributing

Code Contribution Guide

Ways to contribute, how to propose ideas, opening merge requests, and licensing requirements.

Docs Contribution Guide

How to add pages, write integration guides, follow the style rules, and submit a documentation merge request.

Using Claude Code

Pre-configured CLAUDE.md files for working on the OCUDU documentation with Claude Code.

Reporting Issues

How to report bugs and feature requests for the OCUDU codebase via GitLab issues.

API Reference

Doxygen

Auto-generated API reference for the OCUDU codebase, built from source documentation.

Software Architecture Guide

This guide describes the software architecture of OCUDU. Reading it before diving into the codebase will help you understand the design decisions, where contributions belong, and what the codebase expects from every contributor.

The guide is structured in three parts:

- **Vision and Goals** - the platform ambition behind OCUDU and the design goals every contribution must respect.
- **Design Principles** - the engineering principles that shape every class and interface:
  - Clean Architecture - layered dependency model; arrows always point inward.
  - Clean Code - SOLID - the five SOLID principles with OCUDU-specific guidance and violation patterns.
  - Object Oriented Programming - encapsulation, polymorphism, composition over inheritance, ownership rules.
- **Realization** - how the principles above are concretely expressed in the implementation:
  - Threading Model - executor-based, fully configurable, decoupled from protocol logic.
  - Asynchronous Programming - coroutine-based async procedures for complex 5G NR flows.
  - Interfaces - stack-layer interfaces, FAPI, and supported functional splits.
  - Metrics Reporting - per-layer KPI and performance metrics via injected reporter interfaces.
  - Real-Time Safety - timing constraints, forbidden operations, and CI enforcement.
  - Use of C++ - C++17/20/23 features, specialised containers, executors, and SIMD.
  - Testing - unit, component, integration, and E2E test strategy.
  - Plugins - extending or replacing OCUDU components with third-party implementations.
- Software Design Patterns - the patterns used consistently across the codebase.
- **References** - books and online resources for further reading on Clean Architecture, SOLID, C++ software design, and design patterns.

Vision and Goals

The platform goals and design ambitions that every OCUDU contribution must respect.

Design Principles

3 items

Realization

9 items

References

Books and online resources for Clean Architecture, SOLID, C++ software design, and design patterns.

Vision and Goals

Vision

OCUDU is built with the ambition of becoming **The Linux of RAN** - a flexible, open platform on which building, extending, and customising for a particular application is easy and efficient.

OCUDU does not aim to be the best RAN software for any single market segment. Instead, it aims to be the platform on which third parties - partners, community contributors, and researchers - can build the best Public Macro, Private 5G, satellite, or defence RAN software, and evolve toward 6G within the same codebase.

This means OCUDU must be:

- easy to understand for new contributors,
- easy to extend without breaking existing functionality,
- deployable across a wide range of processing platforms,
- and reliable enough to operate in production networks.

The design goals below translate this vision into concrete engineering requirements that every contribution should respect.

Design Goals

Readability

Writing easy-to-read and easy-to-understand code is a first-class design goal. Clear naming, consistent style, and thorough documentation are not optional - they are what allow a global contributor community to work on the same codebase without constant coordination overhead.

Modularity

The software is broken down into well-defined components with clean interfaces. A module that respects its interface contract can be replaced, extended, or tested in isolation without affecting the rest of the application. Modularity is the foundation that makes all other goals achievable.

Flexibility

RAN functionality must be easy to extend or reconfigure - for instance by swapping a module, re-wiring how modules connect, or introducing a new protocol variant. New deployment scenarios should require touching only the relevant layer, not rewriting the whole stack.

Portability

The software must run on multiple processing platforms: x86, ARM, FPGA, GPU, and future architectures. Hardware-specific code is confined to the lowest abstraction layer so that the protocol logic above it remains platform-independent.

Scalability

Application performance must scale with available processing resources - for example, with the number of CPU cores. The design must support both small, single-board deployments and large, multi-socket server installations without code changes in the business logic.

Reliability

The software must minimise bugs and behave predictably under load. Reliability is achieved through a combination of clean architecture (fewer unintended couplings mean fewer unexpected failures), a multi-level testing strategy (unit, component, integration, and E2E tests), and continuous sanitizer and profiling runs in CI.

Maintainability

The codebase must remain easy to change over time. This means keeping technical debt low, enforcing consistent patterns, and ensuring that fixes and improvements can be made with confidence that they will not silently break unrelated parts of the system. Maintainability is not a one-time effort - it is preserved by applying the design principles described in this guide consistently on every contribution.

Interoperability

OCUDU must be easy to extend to operate with third-party hardware and software - radio units, accelerators, external PHYs, and external libraries. External interfaces, like Open Fronthaul, F1, E2, NGAP, etc., are stable and standard-compliant, so that third-party integrations do not require changes to the core stack.

Design Principles

Every architectural decision in OCUDU flows from a small set of design principles. These principles are not optional guidelines - they are the shared language that allows contributors across the world to work on the same codebase and produce coherent results. Understanding them is the single most important preparation before writing or reviewing OCUDU code.

Clean Architecture

OCUDU structures its codebase as a set of concentric layers. Dependencies always point inward: toward the business logic and protocol entities, never outward toward drivers, frameworks, or I/O. This one rule has far-reaching consequences:

- Protocol logic is testable in isolation, without radio hardware or a running network.
- Adding a new radio or deployment scenario touches only the outermost layer.
- NR protocol layers (MAC, RLC, PDCP, etc.) are completely decoupled from each other.

See Clean Architecture for a full explanation with diagrams.

Clean Code - SOLID

Five principles, one consistent codebase. SOLID shapes every class and interface in OCUDU:

- **Single Responsibility** - one reason to change.
- **Open / Closed** - extend via new types, not edits.
- **Liskov Substitution** - implementations honour the full interface contract.
- **Interface Segregation** - small, focused interfaces; callers depend only on what they use.
- **Dependency Inversion** - depend on abstractions; outer layers are injected into inner layers, never the reverse.

Object Oriented Programming

OCUDU uses OOP as its main paradigm - not just as a language feature, but as the mechanism that makes layering, interface contracts, and testability work in practice. The key disciplines are encapsulation (hide implementation details), composition over inheritance (inject collaborators rather than inheriting behaviour), and polymorphism through pure-virtual interfaces (the concrete mechanism behind every swappable component).

See Object Oriented Programming for the full breakdown including ownership rules.

Clean Architecture

OCUDU is built on Clean Architecture principles introduced by Robert C. Martin. The core idea is to organise code into concentric layers where dependencies always point inward - from low-level infrastructure details toward high-level business rules - and never the other way around.

Clean Code

SOLID is a set of five object-oriented design principles that, taken together, produce code that is easy to understand, extend, and test. Applying them consistently throughout OCUDU is a hard requirement, not a suggestion. Code review will flag violations.

Object Oriented Programming

OCUDU is written in C++ and uses Object Oriented Programming (OOP) as its main programming paradigm. OOP is not just a language feature here - it is the mechanism that makes the layered architecture, interface-based design, and testability strategy work in practice. There are, however, more specialised use cases where other paradigms are more suitable, and these are applied selectively alongside OOP where they improve clarity or performance.

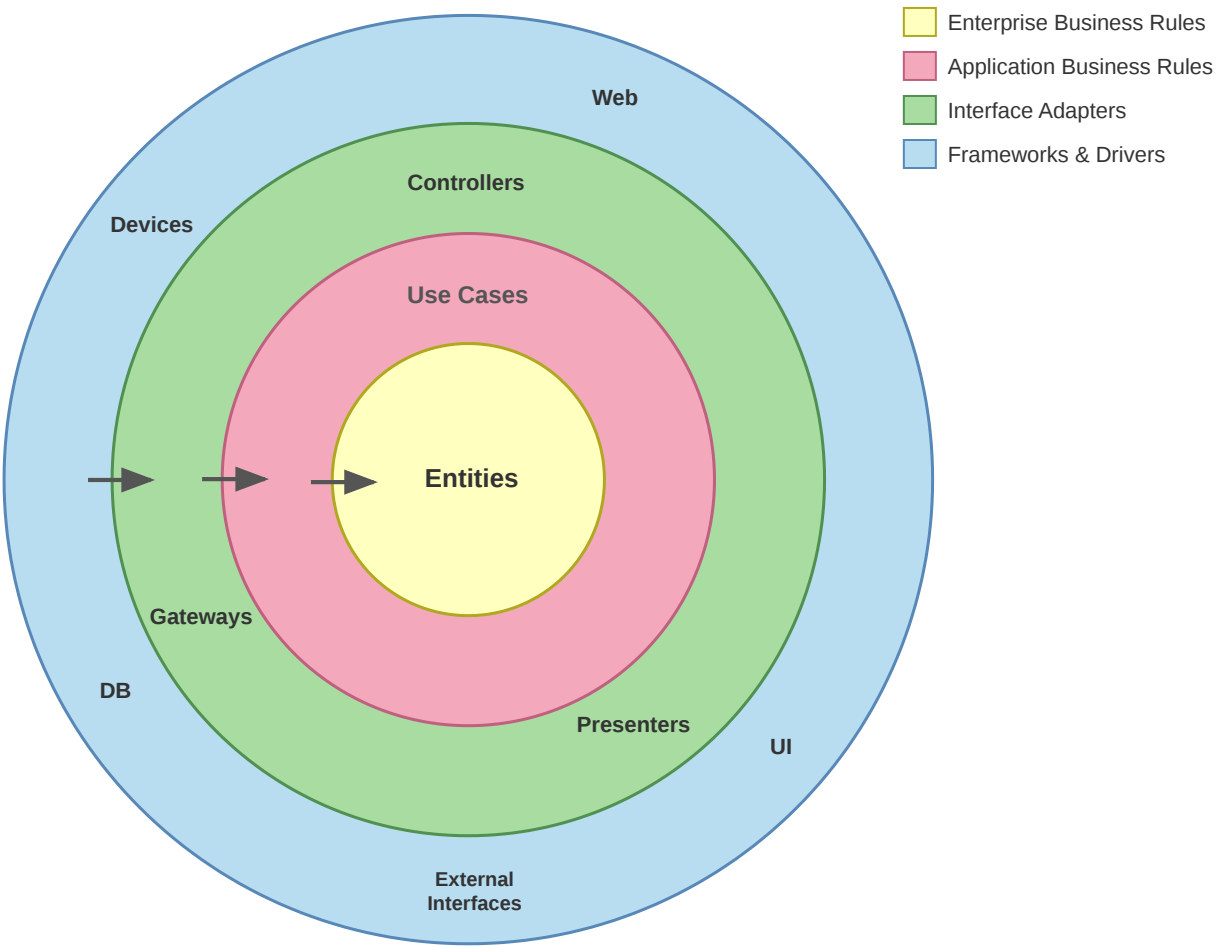
Clean Architecture

OCUDU is built on Clean Architecture principles introduced by Robert C. Martin. The core idea is to organise code into concentric layers where **dependencies always point inward** - from low-level infrastructure details toward high-level business rules - and never the other way around.

The dependency rule

The single most important rule is: **source code dependencies must point inward only**. An inner layer never knows anything about an outer layer. This means:

- High-level business logic does not depend on frameworks, databases, or I/O.
- Business rules can be tested without spinning up any external infrastructure.
- Swapping out an outer-layer component (e.g. a radio driver) does not require touching inner-layer code.



Reproduced from Robert C. Martin, "The Clean Architecture", 2012. The original diagram and article are © Robert C. Martin.

Layers in OCUDU

OCUDU maps the Clean Architecture rings directly onto the 5G NR protocol stack, from highest to lowest abstraction:

Abstraction	Layer	What lives here	OCUDU folders
Highest	**Protocol entities**	Pure 3GPP TS implementations: protocol state machines, data plane processing, radio resource scheduling	`mac/`, `rlc/`, `pdcp/`, `rrc/`, `sdap/`, `scheduler/`, upper `phy/`
↑	**Functional units**	gNB component orchestration: coordinates protocol entities, manages UE contexts, drives cell bring-up and tear-down	`cu_cp/`, `cu_up/`, `du/`
↓	**Interface adaptors**	3GPP signalling stacks and boundary glue: adapts internal interfaces to wire protocols and external systems	`f1ap/`, `ngap/`, `e1ap/`, `e2/`, `fapi_adaptor/`, `gateways/`, `ofh/`
Lowest	**Infrastructure**	Platform utilities, radio drivers, DSP primitives, observability: everything the upper layers are built on but do not depend on directly	`adt/`, `support/`, `ran/`, `hal/`, `radio/`, `ocuduvec/`, `ocudulog/`, `asn1/`, `security/`, `pcap/`

`apps/` sits above all layers as the composition root — it creates and wires the functional units, adaptors, and infrastructure together at startup.

The key constraint: **lower abstraction modules depend on higher abstraction modules, not the reverse**.

Separation of concerns

Mixing code from different abstraction layers (and also between blocks within the same abstraction layer) is actively avoided:

- Keep abstraction layer boundaries and interfaces clean and narrow.
- Minimise coupling between modules at different abstraction levels.
- Use interfaces (abstract C++ classes) as the contract at every boundary. This decouples data from behaviour, enables mocking in tests, and allows CPU-optimised implementations to be swapped in without changing business logic.

Dependency inversion at the boundary

When the direction of control flow would force an inner layer to call outward, the **Dependency Inversion Principle** is applied. The inner layer defines an interface; the outer layer implements it. This keeps the dependency arrow pointing inward even when control flows outward.

A concrete example from OCUDU: when the RLC AM entity exhausts its retransmission budget (`max_retx`), it must tell the DU-high to initiate a UE release. RLC is an inner layer; DU-high is an outer layer. RLC must never hold a pointer to DU-high — that would be an outward dependency.

Instead, RLC defines the `rlc_tx_upper_layer_control_notifier` interface (in `include/ocudu/rlc/rlc_tx.h`) with an `on_max_retx()` method. The DU-high implements this interface through `rlc_tx_control_notifier` (in `du/du_high/du_manager/du_ue/du_ue_adapters.h`) and passes it into the RLC entity at construction time.

At runtime, control flows inward-to-outward: RLC → `on_max_retx()` → DU-high. At compile time, the dependency arrow still points inward: DU-high includes the RLC header to implement the interface; RLC includes nothing from DU-high.

What this means for contributors

When adding a feature, decide which layer it belongs to before writing any code:

- **New radio hardware** → touch only the infrastructure layer (`hal/`, `radio/`, `ofh/`). The protocol stack above never changes.
- **New full L1 (third-party or custom)** → implement the `split6_o_du_low_plugin` interface (`apps/units/flexible_o_du/split_6/o_du_low/split6_o_du_low_plugin.h`). Your L1 exposes FAPI P5/P7 adaptors; OCUDU's DU-high connects to them without knowing what is behind the interface. No changes to MAC or above.
- **New PHY algorithm (e.g. channel estimator)** → implement the `port_channel_estimator` or `dmrs_pusch_estimator` interface (`include/ocudu/phy/upper/signal_processors/channel_estimator/`), register it via the factory, and add your implementation under `lib/phy/upper/signal_processors/channel_estimator/`. The PUSCH pipeline that calls `compute()` is unchanged.
- **New protocol behaviour** → touch only the protocol entities or functional units (`mac/`, `rlc/`, `cu_cp/`, etc.). Infrastructure and adaptors are unaffected.

Keeping changes scoped to the correct layer is what allows the codebase to stay portable, testable, and maintainable as it grows.

Benefits at a glance

- **Portability**: C++ codebase runs across different platforms; hardware dependencies are isolated to the lowest layer.

- **Testability**: Business rules are tested without external components by mocking interface implementations.
- **Decoupled NR layers**: MAC, RLC, PDCP, and other NR layers are completely decoupled from each other through interfaces.
- **3rd-party integration**: External PHYs connect via FAPI; the rest of the stack is unaffected.
- **Flexible execution**: Memory and threading resources are configurable without touching business logic.
- **CPU optimisation**: Interface-based design allows swapping a generic implementation for a SIMD-optimised one without changing call sites.

Clean Code

SOLID is a set of five object-oriented design principles that, taken together, produce code that is easy to understand, extend, and test. Applying them consistently throughout OCUDU is a hard requirement, not a suggestion. Code review will flag violations.

Principle	Rule
Single Responsibility	Each class does one thing.
Open/Closed	Extend behaviour through interfaces, not by modifying existing code.
Liskov Substitution	Implementations are substitutable for their interface.
Interface Segregation	Prefer small, cohesive interfaces over large, general ones.
Dependency Inversion	Depend on abstractions, not concrete types.

Single Responsibility Principle (SRP)

> *A class should have only one reason to change.*

Each class does one thing and does it well. If a class changes because the scheduling algorithm changed *and also* because the logging format changed, it has two responsibilities and should be split.

In OCUDU: A MAC scheduler class is responsible for scheduling decisions. It does not format log messages, manage threads, or serialise ASN.1 messages - it delegates those concerns to collaborators injected through its constructor.

Common violation to watch for: "God classes" that accumulate helper methods over time. If a class has more than one section in its header that addresses a completely different concern, split it.

Open / Closed Principle (OCP)

> *Software entities should be open for extension, but closed for modification.*

Adding new behaviour should be possible by implementing a new class that conforms to an existing interface, not by editing existing classes. Editing an existing, tested class risks introducing regressions and spreads the cost of a change to every caller.

In OCUDU: Adding a new scheduling policy means implementing `mac_scheduler` and wiring it in - not adding another `if/else` branch inside the existing scheduler. Adding a new radio device means

implementing the radio plugin interface - not modifying the radio abstraction layer.

Common violation to watch for: Long `switch` or `if/else` chains that grow every time a new variant is added. Each new case is a signal that an interface and a set of implementations are missing.

Liskov Substitution Principle (LSP)

> *Objects of a subclass must be substitutable for objects of the base class without altering the correctness of the program.*

Any concrete implementation of an interface must honour the full contract of that interface - not just its method signatures, but its pre-conditions, post-conditions, and invariants. A mock used in tests must behave like a real implementation would, or the tests prove nothing.

In OCUDU: If `mac_cell_slot_handler::handle_slot_indication()` guarantees it will never emit a grant that exceeds the configured bandwidth, every implementation - including mocks - must uphold that guarantee. Callers are allowed to rely on it.

Common violation to watch for: Implementations that override a method to throw `not_implemented` or silently do nothing. If the full interface is not implementable, the interface is too large (see ISP below).

Interface Segregation Principle (ISP)

> *Clients should not be forced to depend on methods they do not use.*

Interfaces must be narrow and cohesive. A component that only needs to read metrics should not be given an interface that also exposes write and reset methods. Large interfaces couple unrelated callers and make mocking harder.

In OCUDU: Layer boundaries are crossed via small, purpose-built interfaces. A component that produces downlink grants exposes a `pdcch_resource_allocator` interface; a component that consumes them depends on that interface alone. It does not receive a handle to the entire scheduler.

Common violation to watch for: Interface files that grow long as new callers request new methods. When a method is added to an interface "because caller X needs it," check first whether a separate, narrower interface for X's use case is the right solution.

Dependency Inversion Principle (DIP)

> *High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details.*

This is the principle that directly enforces Clean Architecture's dependency rule in C++. Inner-layer classes hold references to interfaces. Outer-layer classes implement those interfaces and are injected at construction time. The inner layer never `#include`s a concrete outer-layer type.

In OCUDU: The MAC layer defines `lower_phy_downlink_handler`. The PHY layer implements it. The MAC entity is constructed with a reference to `lower_phy_downlink_handler` - it never instantiates a PHY object directly. This is what makes it possible to test the MAC without a PHY.

Common violation to watch for: Constructors that `new` their own collaborators instead of receiving them as arguments. Any use of `new ConcreteType()` inside a class body (rather than a factory) is a dependency inversion violation unless the class itself owns the concrete type as an implementation detail.

Quick reference

Principle	Rule	Violation signal
SRP	One reason to change	Class with unrelated methods; "and also" in descriptions
OCP	Extend via new types, not edits	Growing `if/switch` chains per new variant
LSP	Implementations honour full contract	`not_implemented` overrides; mocks that lie
ISP	Small, cohesive interfaces	Callers ignoring most of an interface's methods
DIP	Depend on abstractions	`new ConcreteType()` inside a class body

Object Oriented Programming

OCUDU is written in C++ and uses Object Oriented Programming (OOP) as its main programming paradigm. OOP is not just a language feature here - it is the mechanism that makes the layered architecture, interface-based design, and testability strategy work in practice. There are, however, more specialised use cases where other paradigms are more suitable, and these are applied selectively alongside OOP where they improve clarity or performance.

Core concepts and how they apply

Encapsulation

A class owns its data and controls how it is accessed. Implementation details - internal state, helper methods, data structures - are hidden behind a public interface. Callers depend only on what a class promises to do, not on how it does it.

In OCUDU this means:

- Internal state of a protocol entity (e.g. an RLC bearer's window, a MAC scheduler's queue) is never accessed directly from another layer.
- Only the methods declared in the public interface of a class are stable API. Everything else is free to change without breaking callers.

Abstraction

Classes model real domain concepts at the right level of detail for their layer. A MAC entity is not a bundle of raw buffers and timers - it is an object with meaningful operations (`handle_pdu`, `schedule_ul_grant`, …). Naming and interface design should reflect the 3GPP domain, not the internal implementation.

Inheritance - use sparingly

Inheritance is used to express genuine "is-a" relationships. It is **not** used as a code-reuse mechanism. In practice, most inheritance in OCUDU is one level deep: a concrete class inherits from a pure-virtual interface class and nothing else.

Prefer **composition over inheritance** when sharing behaviour. Injecting a collaborator as a constructor parameter is almost always cleaner and more testable than inheriting from a base class that bundles that behaviour.

Polymorphism

Polymorphism through virtual functions is the backbone of OCUDU's interface-based design. A pointer or reference to an interface class (pure virtual) can hold any conforming implementation at runtime. This is what makes it possible to:

- swap in a mock implementation during unit tests,
- replace the built-in PHY with a third-party one via a plugin,
- select a CPU-optimised algorithm at startup without changing call sites.

The dependency rule of Clean Architecture is enforced directly through this mechanism: inner layers hold pointers to interface classes defined in the same layer; outer layers provide the concrete implementations.

Composition over inheritance

When a class needs a capability, inject it as a collaborator rather than inheriting it:

```
// Prefer this:
class mac_entity {
public:
explicit mac_entity(std::unique_ptr<mac_scheduler> sched) :
scheduler(std::move(sched)) {}
private:
std::unique_ptr<mac_scheduler> scheduler;
};

// Over this:
class mac_entity : public mac_scheduler_base { ... };
```

Composition keeps classes focused, makes dependencies explicit, and means each collaborator can be mocked independently in tests.

Object ownership and lifetime

OCUDU follows clear ownership rules to avoid memory errors and undefined lifetime:

Ownership model	When to use
`std::unique_ptr`	One clear owner; ownership can be transferred
`std::shared_ptr`	Shared ownership with reference counting (use sparingly - prefer unique ownership)
Raw reference or pointer	Non-owning access to an object whose lifetime is guaranteed to outlive the reference

Raw pointers are never used for ownership. Returning a raw pointer signals "I am not giving you ownership; the object is managed elsewhere and will outlive this pointer." If that guarantee cannot be made, use a smart pointer.

Realization

This section explains how the design principles described earlier materialize into concrete implementation decisions in OCUDU. Each page covers one aspect of how abstract concepts - clean layering, interface-based design, asynchronous procedures - are actually expressed in the codebase.

Threading model

The threading model is fully configurable and completely decoupled from protocol logic. Business entities never create or manage threads directly. Instead, they schedule work onto executor abstractions - task executors, strands, thread pools, and lock-free queues - that can be wired up differently for each deployment target.

See Threading Model.

Asynchronous programming

5G NR procedures are inherently asynchronous: they wait for packet arrivals, timer expirations, peer responses, and cross-layer transactions. OCUDU expresses these multi-step procedures using a coroutine library that makes complex asynchronous logic read as straightforward sequential code.

See Asynchronous Programming.

Interfaces

All communication between software components crosses a well-defined interface boundary. Stack layers talk to each other through pure-virtual C++ interfaces. The PHY-MAC boundary follows the FAPI specification. Multiple functional splits (CU/DU, split 6, split 7.2x, split 8) are supported, each realised through the same interface discipline.

See Interfaces.

Metrics reporting

Every protocol layer reports its own performance and KPI metrics independently. This gives operators and developers a per-layer view of system behaviour without coupling the reporting mechanism to any specific observability backend.

See Metrics Reporting.

Real-time safety

The L1 and time-critical paths of the L2 must meet strict per-slot timing budgets. OCUDU enforces real-time safety through a combination of coding discipline (no dynamic allocation, no system calls, no explicit synchronisation on the critical path) and continuous instrumentation that catches violations automatically in CI.

See Real-Time Safety.

Use of C++

The entire codebase is C++17 compliant with selective use of C++20 and C++23 features. OCUDU makes heavy use of the STL, provides its own specialised containers for real-time use, relies on coroutines for asynchronous programming, abstracts the execution model through executors, and wraps SIMD intrinsics behind a portability layer.

See Use of C++.

Software Design Patterns

OCUDU uses a curated set of well-known patterns consistently throughout the codebase. Recognising them when reading code - and reaching for the right one when writing new code - is an important part of contributing effectively. The most common are Adapter, Strategy, Observer, Factory, Decorator, and Null Object.

See Software Design Patterns for the full catalogue with OCUDU-specific context.

Threading Model

Overview

Asynchronous Programming

The problem

Interfaces

Principle

Metrics Reporting

Overview

Real-Time Safety

The timing constraint

Use of C++

Language standard

Testing

OCUDU operates a multi-level testing strategy. Minimising internal dependencies inside components directly enables each level:

Plugins

5 items

Software Design Patterns

Design patterns are reusable solutions to recurring structural problems. OCUDU uses a curated set of well-known patterns consistently throughout the codebase. Recognising them when reading code - and reaching for them when writing new code - is an important part of contributing effectively.

Threading Model

Overview

The threading model in OCUDU is fully configurable and completely agnostic from the core protocol components. Protocol entities never create, own, or directly manage threads. Instead, all work is expressed as tasks submitted to **executor** abstractions. The concrete threading arrangement - how many threads exist, how they are prioritised, and which components share them - is a deployment-time configuration decision, not a protocol-logic decision.

This separation is what allows the same protocol code to run correctly on a single-core embedded board, a four-core edge server, or a large multi-socket machine, simply by changing the executor configuration.

Execution primitives

OCUDU provides a layered set of execution primitives:

Task executor

The fundamental unit. A task executor accepts a callable and arranges for it to run. The caller does not know - and must not assume - on which thread the callable will execute or when. Protocol code submits tasks and moves on; it does not block waiting for completion.

Thread pools

A thread pool backs one or more task executors with a set of worker threads. OCUDU supports:

Pool type	Description
Single worker	One dedicated thread; tasks run strictly in submission order
Multi-worker pool	N worker threads sharing a task queue; tasks may run concurrently
Multi-priority pool	Multiple priority lanes within one pool; high-priority tasks are dequeued first

Strands

A strand serialises task execution on top of an existing thread pool. Tasks submitted to a strand are guaranteed to run one at a time and in order, even though the underlying pool may have many workers. This gives a component the sequential execution guarantee of a single thread while sharing pool resources with other components.

Strands are the primary tool for protecting component-internal state without explicit locking. If all access to a component's state flows through a single strand, no mutexes are needed.

Lock-free queues

Inter-component communication on the critical path uses lock-free queues to avoid priority inversion and unbounded blocking. Lock-free queues are the standard tool at any boundary where a non-real-time producer must hand work to a real-time consumer.

A typical example: an RRC reconfiguration procedure runs on a non-real-time executor and decides to update the UE's bearer configuration. The MAC slot handler runs on a hard-deadline thread that fires every 500 µs. The reconfiguration cannot take a mutex that the slot handler might also hold — that would risk priority inversion and a missed slot deadline. Instead, the reconfiguration procedure pushes a configuration update command into a lock-free queue. The slot handler drains the queue at the start of each slot and applies any pending updates to the UE object before scheduling begins. Neither side ever blocks waiting for the other.

OCUDU provides two lock-free queue variants, both expressed as specialisations of the `concurrent_queue<T, Policy, BlockingPolicy>` template:

- **SPSC** (`concurrent_queue_policy::lockfree_spsc`) - single-producer, single-consumer. Backed by rigtorp's `SPSCQueue`. Use this when ownership of each end of the queue is fixed to one thread; it has the lowest overhead of any queue type. Header: `ocudu/adt/spsc_queue.h`.
- **MPMC** (`concurrent_queue_policy::lockfree_mpmc`) - multi-producer, multi-consumer. Backed by rigtorp's `MPMCQueue` (bounded) or moodycamel's `ConcurrentQueue` (unbounded). Use this when multiple threads may push or pop concurrently. Headers: `ocudu/adt/mpmc_queue.h`, `ocudu/adt/moodycamel_mpmc_queue.h`.

Both variants support a `non_blocking` and a `sleep` wait policy, selected as the third template parameter.

Fork limiters

A fork limiter caps the degree of parallelism for a set of tasks. This prevents a burst of parallel work from monopolising pool threads and starving other components with tighter timing requirements.

Dependency on the threading model

Protocol components express their threading requirements through their constructor: they receive executor references for the work they need to schedule. They never call threading APIs directly. This means:

- The threading model can be reconfigured without touching any protocol code.
- Unit tests can inject a synchronous (inline) executor, making async code testable without threads.
- The architecture team can reason about thread budgets and priorities centrally, in the wiring layer, rather than hunting for thread creation scattered across the codebase.

Asynchronous Programming

The problem

5G NR procedures are deeply asynchronous. A single procedure defined in a 3GPP TS may need to:

- send a message and wait for a response from the peer,
- arm a timer and react differently depending on whether the response or the timeout arrives first,
- suspend until a resource becomes available,
- retry after a back-off period,
- and branch on the outcome of a transaction that completes on a different thread.

Expressing this logic with callbacks or explicit state machines produces code that is hard to follow, error-prone, and difficult to test. The control flow is fragmented across many functions and state variables; understanding what a procedure does requires mentally stitching the pieces back together.

The solution: custom stackless coroutines

OCUDU uses a **custom stackless coroutine framework** to express asynchronous procedures as sequential code. The framework is implemented entirely in C++17 without relying on C++20 language coroutines (a migration to C++20 coroutines is planned for the future).

A procedure is a callable struct with an `operator()` that receives a `coro_context` reference. A set of macros marks suspension points inside that function body. When the procedure suspends - waiting for a response, a timer, or an event - control returns to the caller. When the awaited condition is met, the framework resumes the procedure at exactly the point where it suspended.

```
void rrc_setup_procedure::operator()(coro_context<async_task<void>>& ctx)
{
CORO_BEGIN(ctx);

create_srb1();
transaction = event_mng.transactions.create_transaction(procedure_timeout);
send_rrc_setup();

CORO_AWAIT(transaction);   // suspend until RRCSetupComplete or timeout

if (!transaction.has_response()) {
logger.log_warning("\"{}\" timed out", name());
rrc_ue.on_ue_release_required(cause_protocol_t::unspecified);
CORO_EARLY_RETURN();
}

process_setup_complete(transaction.response());
```

```
CORO_RETURN();
}
```

The procedure above spans a network round-trip yet reads as a linear sequence. The framework handles all suspension, resumption, and cancellation bookkeeping.

Coroutine macros

Macro	Purpose
`CORO_BEGIN(ctx)`	Opens the coroutine body; jumps to the last suspension point on resume
`CORO_AWAIT(awaitable)`	Suspends until `awaitable` is ready; discards the result
`CORO_AWAIT_VALUE(result, awaitable)`	Like `CORO_AWAIT` but stores the result
`CORO_RETURN(...)`	Returns a value (or `void`) and reaches the final suspension point
`CORO_EARLY_RETURN(...)`	Returns early from any point in the procedure

These macros expand to a `switch`-based state machine. Each `CORO_AWAIT` stores the current line number as the resume index; on the next call to `operator()` the switch jumps directly back to that line.

Task types

`async_task<R>` is the standard return type for a coroutine. It is a **lazy** task: it does not start executing until it is awaited by another coroutine or explicitly launched.

```
async_task<void> launch_rrc_setup_procedure(ue_context& ue);
```

`eager_async_task<R>` starts executing immediately upon creation. It is used for fire-and-forget or top-level procedures that must begin without being awaited.

Message passing and thread isolation

Coroutines handle async sequencing *within* an execution context. For communication *across* execution contexts — between a UE strand and a cell strand, or between a protocol layer and a worker pool - OCUDU uses message passing rather than shared mutable state.

The rule is: **post work to the owner's execution context instead of reaching into it from another thread.** This eliminates mutex contention, prevents priority inversion, and keeps latency bounded.

task_executor

`task_executor` is the interface used to post work across strand and thread boundaries:

```
class task_executor
{
public:
// Dispatches a task; may run inline if safe. Returns false if queue is full.
[[nodiscard]] virtual bool execute(unique_task task) = 0;

// Always enqueues for later execution. Returns false if queue is full.
[[nodiscard]] virtual bool defer(unique_task task) = 0;
};
```

`unique_task` is a move-only callable with a small-buffer optimisation that avoids heap allocation for typical closures.

Use `defer()` when the caller must not run the task inline (for example when strict ordering matters or the caller is on a different thread). Use `execute()` when inline execution is acceptable to reduce dispatch latency.

Strands

A *strand* wraps a `task_executor` and adds one guarantee: tasks posted to the same strand always run one at a time, regardless of how many producer threads are posting. This gives a component a safe, single-owner execution context without any locking - multiple external threads can safely drive a single-owner component by posting tasks to its strand.

The cross-strand pattern

Instead of protecting shared state with a mutex:

```
// Avoid: lock on shared state
{
std::lock_guard lock(ue_mutex);
ue.apply_config(new_config);
}
```

Post the mutation to the component's own strand:

```
// Prefer: deliver the mutation as a task
ue_exec.defer([this, new_config]() {
ue.apply_config(new_config);
});
```

All access to `ue` flows through its strand, so there is nothing to lock.

Switching execution context inside a coroutine

When a coroutine needs to continue on a different executor, `try_execute_on` provides an awaitable that suspends and resumes in the target context:

```
// Suspend and resume in other_exec's context.
// Returns false if the target queue was full.
bool ok;
CORO_AWAIT_VALUE(ok, try_execute_on(other_exec));
```

For compute-heavy work that should not block the control strand, `try_offload_to_executor` dispatches a callable to a worker executor and resumes the coroutine back on the original strand once the work completes:

```
result_t result;
CORO_AWAIT_VALUE(result, try_offload_to_executor(worker_exec, ctrl_exec, []{ return
compute(); }));
```

Transactions: the standard request/response pattern

Most 3GPP procedures follow a request/response pattern with a timeout. OCUDU models this with `protocol_transaction<ResponseType>`:

```
// Create a transaction with a timeout (e.g. T300).
transaction = event_mng.transactions.create_transaction(procedure_timeout);

send_rrc_setup();

CORO_AWAIT(transaction);   // suspend until response arrives or timeout fires

if (!transaction.has_response()) {
// transaction.failure_cause() is protocol_transaction_failure::timeout or ::cancel
handle_failure();
CORO_EARLY_RETURN();
}

process(transaction.response());
```

`protocol_transaction_failure` has three values: `timeout`, `cancel`, and `abnormal`. The `has_response()` / `failure_cause()` pattern avoids exceptions and keeps the happy path and error path adjacent.

Cancellation

A running coroutine can be cancelled by its owner. The framework propagates cancellation by negating the stored state index and re-entering `operator()`. The macro expansion detects the negative index,

cleans up the in-flight awaiter, and unwinds the procedure cleanly without resource leaks. Procedures do not need to add cancellation-specific code; the macro infrastructure handles it automatically.

Design principles for async code

Suspension points are the only suspension points

A coroutine only suspends at an explicit `CORO_AWAIT` or `CORO_AWAIT_VALUE`. Between awaits it runs synchronously on the executor that resumed it. Treat every block between two awaits as an atomic section.

No blocking on the executor thread

A coroutine must never block its executor thread - for example by calling `std::this_thread::sleep_for` or blocking on a mutex. Blocking prevents other tasks queued on the same executor from running and can cause missed slot deadlines. Any wait must be expressed as an awaitable that suspends the coroutine and returns the thread to the executor.

Errors propagate in-band

Async functions return `async_task<expected<T, E>>`. The caller awaits the result and handles the error inline, keeping the happy path and the error path adjacent and readable.

Testing async code

Because the threading model is injected, tests can drive coroutines on a manual executor that steps tasks forward one at a time. This makes it possible to test multi-step async procedures deterministically, without real timers or threads, by simply advancing the executor between assertions.

Interfaces

Principle

All communication between OCUDU software components crosses a well-defined interface boundary expressed as a pure-virtual C++ class. No component calls into another component's internals directly. This is the practical realisation of the Dependency Inversion Principle: the consuming layer defines the interface it needs; the producing layer implements it.

The benefits are consistent:

- Either side of an interface can be replaced - by a different implementation, a mock, a plugin, or a third-party component - without touching the other side.
- The interface is the documentation of the contract between two components. Reading it is sufficient to understand what one layer expects from the next.
- Unit and component tests stub out dependencies by providing lightweight implementations of the relevant interfaces.

Stack layer interfaces

Any two components that communicate across a significant boundary do so through a pure-virtual interface. This includes 3GPP protocol layers (PHY, MAC, RLC, PDCP, RRC), functional splits (DU-high to DU-low, CU to DU), infrastructure services (timers, executors, metrics reporters), and any plugin or third-party integration point.

The critical consequence is that **no layer object holds a direct pointer to another layer object**. RLC does not know that MAC exists as a concrete class; MAC does not know that RLC exists as a concrete class. What they hold are pointers to interfaces. The wiring - creating both objects and connecting them through their interfaces - happens in the application or test harness, not inside the protocol components themselves.

Adapter objects are what make this work in practice. When two layers need to communicate but their interfaces do not match exactly, an adapter sits between them, implementing the interface one side expects while delegating to the other side. The adapter is instantiated in the wiring layer; neither protocol component is aware of its existence. This is described further in the Adapter design pattern.

Typical 3GPP layer boundaries:

Boundary	Interface direction
MAC ↔ PHY	MAC holds `lower_phy_downlink_handler` and `lower_phy_uplink_handler`; PHY calls into `mac_cell_slot_handler`
RLC ↔ MAC	RLC holds `mac_rlc_ul_scheduler`; MAC calls into `rlc_rx_lower_layer_interface`
PDCP ↔ RLC	PDCP holds `rlc_tx_lower_layer_interface`; RLC calls into `pdcp_rx_lower_notifier`
RRC ↔ PDCP	RRC holds `pdcp_entity_control`; PDCP calls into `rrc_ul_dcch_pdu_handler`

Each interface is defined in the header of the layer that depends on it, not in the header of the layer that implements it - this is what keeps the dependency pointing inward.

FAPI - the PHY-MAC boundary

The **FAPI (Functional Application Platform Interface)** is the standardised interface between the L2 (MAC) and L1 (PHY). OCUDU implements FAPI as the primary mechanism for PHY-MAC communication. Using a standardised interface at this boundary means:

- The built-in OCUDU PHY can be swapped for a third-party PHY without any changes to the MAC. This is the `du_high` plugin.
- A third-party DU-high (L2 and above) can drive OCUDU's L1 via the `du_low` plugin interface, which exposes the FAPI boundary upward so external schedulers and MAC implementations can plug in above it.
- Integration with commercial small-cell PHYs, SDR frameworks, and FPGA-based modems is possible without modifying the core stack.

Functional splits

OCUDU supports multiple functional split points, each realised through the same interface discipline:

Split	Description
CU/DU split	RRC and PDCP run in the Central Unit (CU); RLC, MAC, and PHY run in the Distributed Unit (DU). Communication over the F1 interface.
Split 6	MAC and above in the DU High; PHY in the DU Low. The FAPI interface is the split point.
Split 7.2x	Upper PHY in the DU; lower PHY and radio in the RU. The Open Fronthaul (OFH) interface is the split point.
Split 8	Full PHY in the RU; the DU handles MAC and above. The radio plugin interface is the split point.

Each split is supported by defining the appropriate interface at the split boundary and providing implementations on both sides. Adding a new split means adding a new interface - existing code above and below the split is unaffected.

Metrics Reporting

Overview

Every protocol layer in OCUDU reports its own performance and KPI metrics independently. Metrics are not collected by a central observer that reaches into layer internals - each layer pushes metrics outward through a reporting interface, keeping the measurement concern separated from the protocol logic.

Metric categories

Category	Examples
RAN KPI metrics	Throughput (DL/UL), BLER, CQI distribution, HARQ retransmission rate, active UE count
Scheduling metrics	Grant utilisation, PRB allocation per UE, scheduler queue depth
Performance metrics	Task execution latency, inter-slot jitter, CPU utilisation per component
Execution metrics	Executor queue depth, strand backlog, thread pool saturation

Reporting interface

Each layer that produces metrics requires a `mac_metrics_notifier` interface injected at construction time. The layer calls `on_new_metrics_report()` with a typed metrics struct; it does not know - and must not assume - where the data goes. The wiring layer connects the notifier interface to the active backend (log file, Prometheus endpoint, in-memory ring buffer for testing, or a null-object no-op).

This design means:

- Adding a new metrics backend requires only a new implementation of `mac_metrics_notifier` - no protocol code changes.
- In unit tests, a `null_mac_metrics_notifier` or a capturing test double is injected; no real backend is needed.
- Metrics collection can be disabled at zero runtime cost by injecting the null object.

Per-layer ownership

Each layer owns the definition of its own metrics struct. The MAC defines `mac_metric_report`; the RLC defines `rlc_metrics`; and so on. This avoids a monolithic metrics schema that every layer must update whenever any other layer changes. Consumers that want a combined view aggregate the per-layer structs at the wiring layer.

Relationship to instrumentation

Metrics reporting covers aggregated, periodic measurements (e.g. throughput averaged over one second). It is complementary to - and separate from - the other instrumentation strategies:

- **Logging** captures discrete events and errors at human-readable granularity.
- **Traces** capture fine-grained, timestamped events for timing analysis of individual slot processing cycles.
- **Metrics** capture aggregated KPIs suitable for dashboards and alerting.

All three are injected through interfaces and can be enabled, disabled, or replaced independently.

Real-Time Safety

The timing constraint

A 5G NR base station processes one slot every **500 µs** (numerology µ=1) or **1 ms** (µ=0). The L1 modem and the time-critical portions of the L2 scheduler must complete all their work within this budget - every slot, without exception. A single missed deadline causes a dropped slot, which translates directly to degraded throughput and user experience for every UE in the cell.

This hard real-time requirement shapes how code on the critical path must be written.

What the critical path must never do

The following operations are **forbidden** on any code path that executes within the slot processing budget:

Forbidden operation	Why
Dynamic memory allocation (`new`, `malloc`, `std::vector::push_back` that triggers a resize, …)	Allocation involves a system call and a lock on the allocator; latency is unbounded
System calls (file I/O, socket operations, `clock_gettime` with VDSO disabled, ..)	Kernel transitions have unpredictable latency under load
Explicit synchronisation (`std::mutex::lock`, condition variables, semaphores)	Blocking on a contended lock causes priority inversion with no bounded wait time
Exceptions	Exception handling involves heap allocation and dynamic dispatch; timing is not guaranteed
Logging to a blocking sink	Writing to a file or socket on the critical path introduces I/O latency

Pre-allocated memory pools, lock-free queues, and non-blocking ring buffers are the standard alternatives for each of these.

Coding discipline

Real-time safety is enforced through code review and tooling primarily. We primarily use **RTSan (real-time sanitizer)**. RTSan runs daily in CI and reports any allocation, system call, or blocking primitive reached from a real-time-annotated call stack. OCUDU uses the `OCUDU_RTSAN_NONBLOCKING` macro to annotate any code that must not violate RT requirements.

When contributing code to a real-time path, annotate it with the appropriate marker so the sanitizer can enforce the constraint automatically.

Performance profiling

Meeting the timing budget requires knowing where time is actually spent. OCUDU uses:

- **Execution traces** - fine-grained timestamps at slot entry/exit and at key processing stages, written to a lock-free ring buffer and flushed off the critical path.
- **Benchmarks** - micro-benchmarks for inner-loop functions (channel estimation, LDPC encode/decode, FFT) run in CI to catch regressions before they land.
- **Profilers** - periodic profiling runs (perf, VTune) on representative workloads to identify hotspots and guide optimisation.

Real-time safe alternatives

Need	RT-safe approach
Variable-length buffer	Pre-allocated memory pool; fixed-capacity ring buffer
Passing data between threads	Lock-free queue (single-producer / single-consumer or MPSC)
Signalling an event	Atomic flag or lock-free notification primitive
Logging from the critical path	Write to a lock-free ring buffer; a background thread drains it
Timing measurement	`clock_gettime(CLOCK_MONOTONIC_RAW)` or a hardware TSC read

Use of C++

Language standard

The entire codebase is **C++17** compliant. Several features from C++20 and later are adopted ahead of the standard, either through in-house implementations or vendored single-header libraries.

The table below lists every in-house implementation and every vendored library, and what each provides:

Feature / library	In-house or external
Coroutines (`async_task`, `CORO_*` macros)	In-house - custom C++17 stackless coroutine framework
`span<T>`	In-house - contiguous view type
`flat_map<K, V>`	In-house - sorted-vector associative container
Executor and strand framework	In-house - `task_executor`, `strand_executor`, priority strands
SIMD abstraction	In-house - architecture-portable wrapper over SSE/AVX/NEON intrinsics
Memory pools and object pools	In-house - fixed-size, lock-free allocators for real-time paths
Custom containers	In-house - ring buffers, static vectors, concurrent queues built on the primitives below
ASN.1 code generator	In-house - generates strongly-typed C++ from 3GPP ASN.1 schemas
`expected<T, E>`	External: TartanLlama/expected - C++23 `std::expected` backport; re-exported as `ocudu::expected`
String formatting	External: fmt - C++20 `std::format` backport; used directly as `fmt::format`
Lock-free MPMC queue	External: cameron314/concurrentqueue
Lock-free MPMC / SPSC queues	External: rigtorp/MPMCQueue + SPSCQueue
JSON serialisation	External: nlohmann/json
CLI argument parsing	External: CLI11
WebSocket server	External: uWebSockets
Stack traces (debug builds)	External: Backward-cpp

STL and specialised containers

Heavy use of the STL is encouraged. Where the STL does not meet real-time or performance requirements, OCUDU provides its own specialised containers - or wraps the vendored lock-free queue libraries listed above:

- Lock-free and priority queues (backed by cameron314 and rigtorp primitives)

- Memory pools and fixed-size allocators
- Unique and shared object pools
- Custom vectors (`static_vector`) and ring buffers

ASN.1 generator

3GPP protocols define message formats in ASN.1. OCUDU uses its own **ASN.1 generator** that translates ASN.1 syntax directly into strongly-typed C++ types, eliminating manual serialisation code and reducing the risk of protocol encoding errors. The generator is not open-sourced yet.

Asynchronous programming with coroutines

Protocol procedures often involve waiting for responses from peer entities or lower layers. OCUDU uses a **custom C++17 coroutine framework** (not C++20 language coroutines) to express these asynchronous sequences as straightforward, sequential-looking code rather than callback chains or explicit state machines. See Asynchronous Programming for a full description.

Executors

OCUDU abstracts the execution model through an executor framework so that business logic does not depend on how threads are managed. The same protocol code can run on a single-core embedded system, a multi-socket server, or a cloud VM by changing only configuration, not source code.

Executor type	Description
Single worker	One dedicated thread, single-priority queue
Thread pool	Multiple workers sharing a task queue
Multi-priority thread pool	Workers with high/low priority lanes
Strand	Serialises tasks on top of a worker or pool; single and multi-priority variants
Fork limiter	Caps the degree of parallelism for bounded-concurrency tasks

SIMD abstraction

An abstraction layer wraps architecture-specific SIMD intrinsics (SSE, AVX2/512, NEON). Inner-layer algorithm code calls the abstraction; the correct instruction set is selected at compile time or runtime without changes to the algorithm implementation.

Testing

OCUDU operates a multi-level testing strategy. Minimising internal dependencies inside components directly enables each level:

Level	Scope
Unit tests	A single class or function in isolation
Component tests	A subsystem (e.g. MAC scheduler) with mocked neighbours
Integration tests	Multiple layers interacting through real interfaces
E2E tests	Full stack with real radio or simulated RF

See Testing policy for more details.

Plugins

The plugin system allows extending or replacing parts of OCUDU with third-party implementations without modifying the core codebase. Plugins live in the outermost layer of the Clean Architecture: they depend on OCUDU interfaces, but OCUDU never depends on them. The core stack does not know - and does not care - whether it is talking to a built-in implementation or a plugin.

Available plugin types

Plugin	Purpose
Radio	Add support for a new split-8 radio device
DU HI	Interface a 3rd-party PHY; the built-in PHY and higher layers are not instantiated
DU LO	Interface a 3rd-party MAC; the built-in MAC and higher layers are not instantiated
PHY TAP	Tap uplink IQ symbols at the PHY layer for custom processing, monitoring, or telemetry

How the plugin system works

The application controls the entire lifecycle. The pattern is the same for all plugin types:

1. At startup, OCUDU calls a **factory entry point** exported by the plugin, passing configuration parameters.
2. The plugin's factory creates one or more objects that implement a well-defined **C++ interface**.
3. OCUDU stores a pointer to that interface and calls it at runtime - it never touches the plugin's internals.

This directly realises the Dependency Inversion Principle. OCUDU defines the interface; the plugin provides the implementation. The dependency arrow points inward, toward OCUDU, even though the runtime call goes outward.

Plugin lifecycle

```
Startup
OCUDU calls create_phy_tap_factory(nof_rb, nof_ports, args)
└─ Plugin builds factory chain from args
└─ Returns phy_tap_factory to OCUDU

OCUDU calls factory->create() for each cell
```

```
└─ Factory creates external_ul_processor chain
└─ phy_tap_impl wraps it and returns phy_tap to OCUDU

Runtime - per symbol (allocated slot)
OCUDU upper PHY calls phy_tap::handle_ul_symbol()
└─ phy_tap_impl calls external_ul_processor::process()
└─ Decorator chain executes in order
└─ Custom DSP reads/writes resource grid
└─ (Optional) ZMQ decorator streams measurements async

Runtime - per slot (quiet slot, if enabled)
OCUDU calls phy_tap::handle_quiet_grid()
└─ external_ul_processor::process_quiet()

Runtime - per PRACH window
OCUDU calls phy_tap::handle_prach_window()
└─ external_ul_processor::process_prach()
```

Dependency rule

A well-written plugin:

- includes only the interface headers provided by OCUDU (`external_ul_processor.h`, `external_processor_factories.h`),
- never includes internal OCUDU headers outside of those interfaces,
- pre-allocates all memory in constructors - never inside processing callbacks (if it's executed in the critical path),
- defers any I/O (network, disk) to a background thread so the critical path is not blocked.

The PHY TAP Plugin Interface

The PHY TAP plugin is the primary extension point for custom uplink signal processing. It intercepts received IQ samples at the upper PHY layer - after the radio front-end and FFT, before OCUDU's own channel estimation and decoding - giving a plugin full access to the resource grid on every slot.

Implementing a PHY TAP Plugin

A PHY TAP plugin requires three pieces: the processor class, the factory class, and the exported entry point. Follow these steps in order.

Chaining Processors with the Decorator Pattern

More complex use cases need to compose multiple processing stages - for example, running a custom DSP algorithm and simultaneously streaming telemetry to an external system - without either stage knowing about the other. The Decorator pattern achieves this: each stage wraps the previous one and implements the same externalulprocessor interface.

Plugin Configuration

Enabling the PHY TAP plugin

Build System

Build model

The PHY TAP Plugin Interface

The PHY TAP plugin is the primary extension point for custom uplink signal processing. It intercepts received IQ samples at the upper PHY layer - after the radio front-end and FFT, before OCUDU's own channel estimation and decoding - giving a plugin full access to the resource grid on every slot.

The processing interface

Every PHY TAP plugin must implement `external_ul_processor`. This is the interface OCUDU calls:

```
class external_ul_processor
{
public:
virtual ~external_ul_processor() = default;

/// Called once per received uplink symbol on every allocated slot.
/// grid_reader gives read access to the raw IQ samples.
/// grid_writer allows the plugin to write modified samples back.
virtual void process(
resource_grid_writer&                                        grid_writer,
const resource_grid_reader&                                  grid_reader,
slot_point slot,
unsigned                                                     symbol,
span<const uplink_pdu_slot_repository::pusch_pdu>            pusch_pdus,
span<const uplink_pdu_slot_repository::pucch_pdu>            pucch_pdus,
span<const pucch_processor::format1_common_configuration> pucch_f1_pdus,
span<const uplink_pdu_slot_repository::srs_pdu>              srs_pdus) = 0;

/// Called once per slot for quiet (unallocated) slots.
/// Requires enable_quiet_processing=true in the plugin arguments.
virtual void process_quiet(const resource_grid_reader& grid_reader,
slot_point slot) = 0;

/// Called for each received PRACH window.
virtual void process_prach(prach_buffer&                buffer,
const prach_buffer_context& context) = 0;
};
```

There are three distinct callbacks:

Callback	Trigger	Receives
`process()`	Once per uplink symbol on an allocated slot	Resource grid reader/writer, slot/symbol index, all active PDU descriptors (PUSCH, PUCCH, SRS)
`process_quiet()`	Once per quiet (unallocated) slot	Resource grid reader, slot index
`process_prach()`	Once per PRACH reception window	PRACH buffer and context (format, ports, occasions)

The PDU descriptors passed to `process()` tell the plugin exactly which RNTIs are active and where their allocations are in the resource grid - without the plugin needing to decode any scheduling messages itself.

The factory interface

OCUDU does not instantiate processors directly. It asks a **factory** to create them, one per active cell. The factory interface is:

```
class external_ul_processor_factory
{
public:
virtual ~external_ul_processor_factory() = default;

/// Creates one processor instance. Called once per cell at startup.
virtual std::unique_ptr<external_ul_processor> create() = 0;
};
```

Factories receive all configuration they need at construction time - number of resource blocks, number of antenna ports, free-form argument string. This separates "how to configure the processor" from "how to process a symbol", keeping each class focused on a single responsibility.

The adapter layer

OCUDU's internal PHY TAP interface (`phy_tap`) is distinct from `external_ul_processor`. An adapter class bridges the two, translating OCUDU's internal callbacks into the external interface. This insulation means the external interface can remain stable even as OCUDU's internal interfaces evolve:

```
class phy_tap_impl : public phy_tap
{
public:
explicit phy_tap_impl(std::unique_ptr<external_ul_processor> processor) :
processor(std::move(processor))
{}

void handle_ul_symbol(resource_grid_writer&       grid_writer,
```

```
const resource_grid_reader& grid_reader,
slot_point slot, unsigned symbol,
/* PDU spans... */) override
{
processor->process(grid_writer, grid_reader, slot, symbol, /* ... */);
}

void handle_quiet_grid(const resource_grid_reader& grid_reader,
slot_point slot) override
{
processor->process_quiet(grid_reader, slot);
}

void handle_prach_window(prach_buffer&               buffer,
const prach_buffer_context& context) override
{
processor->process_prach(buffer, context);
}

private:
std::unique_ptr<external_ul_processor> processor;
};
```

The entry point

Every plugin must export exactly one free function. This is the only symbol OCUDU resolves from the plugin at startup:

```
// Declared in OCUDU's plugin header; defined in the plugin's factory source file.
std::shared_ptr<phy_tap_factory>
create_phy_tap_factory(unsigned          nof_rb,
unsigned          nof_ports,
const std::string& processor_arguments);
```

The implementation creates the plugin's factory chain and returns it to OCUDU:

```
std::shared_ptr<phy_tap_factory>
create_phy_tap_factory(unsigned nof_rb, unsigned nof_ports,
const std::string& processor_arguments)
{
return std::make_shared<phy_tap_factory_impl>(nof_rb, nof_ports, processor_arguments);
}
```

OCUDU calls this once at startup, holds the returned `phy_tap_factory`, and calls `factory->create()` once per cell to get the per-cell `phy_tap` instance it drives at runtime.

You can find a toy example implementation, and fully detailed instructions, in the PHY TAP Plugin Example repository. To test it, just clone it inside the `plugins` folder (create one if it's not already present) and compile normally.

Implementing a PHY TAP Plugin

A PHY TAP plugin requires three pieces: the processor class, the factory class, and the exported entry point. Follow these steps in order.

Step 1 - implement the processor

Create a class that inherits from `external_ul_processor` and implements all three callbacks.

```
class iq_tap_processor : public external_ul_processor
{
public:
iq_tap_processor(unsigned nof_rb, unsigned nof_ports) :
nof_ports(nof_ports),
// Pre-allocate the working buffer in the constructor.
// Dynamic allocation inside process() is forbidden.
temp(nof_rb * NOF_SUBCARRIERS_PER_RB)
{}

void process(resource_grid_writer&       grid_writer,
const resource_grid_reader& grid_reader,
slot_point slot, unsigned symbol,
span<const uplink_pdu_slot_repository::pusch_pdu> pusch_pdus,
/* ... */) override
{
for (unsigned port = 0; port != nof_ports; ++port) {
// Read the raw IQ samples for this port and symbol.
grid_reader.get(temp, port, symbol, 0);

// ---- Insert custom DSP here ----
// Example: scale by 0.5 (attenuate by 6 dB).
srsvec::sc_prod(temp, temp, 0.5f);
// --------------------------------

// Write the modified samples back.
grid_writer.put(port, symbol, 0, temp);
}
}

void process_quiet(const resource_grid_reader&, slot_point) override
{
// Nothing to do for quiet slots in this example.
// Quiet callbacks only fire when enable_quiet_processing=true.
}

void process_prach(prach_buffer&              buffer,
const prach_buffer_context& context) override
{
// Iterate ports, time/frequency occasions, and PRACH symbols.
// Use buffer.get_symbol(port, td_occasion, fd_occasion, symbol)
// to access the IQ samples.
```

```
}

private:
unsigned          nof_ports;
std::vector<cf_t> temp; // pre-allocated; never resize inside process()
};
```

> **REAL-TIME CONSTRAINT**
>
> `process()` runs on the L1 timing-critical thread and must complete within the slot budget (500 µs at numerology µ=1). Never allocate memory, call blocking APIs, perform I/O, or take locks inside `process()`, `process_quiet()`, or `process_prach()`. Pre-allocate all buffers in the constructor. See Real-Time Safety for the full list of forbidden operations.

Step 2 - implement the factory

The factory is responsible for constructing the processor with its configuration. OCUDU calls `create()` once per active cell at startup.

```
class iq_tap_processor_factory : public external_ul_processor_factory
{
public:
iq_tap_processor_factory(unsigned nof_rb, unsigned nof_ports) :
nof_rb(nof_rb), nof_ports(nof_ports)
{}

std::unique_ptr<external_ul_processor> create() override
{
return std::make_unique<iq_tap_processor>(nof_rb, nof_ports);
}

private:
unsigned nof_rb;
unsigned nof_ports;
};
```

Step 3 - export the entry point

Define `create_phy_tap_factory()`. This is the single symbol OCUDU looks up from the plugin. It receives the cell parameters and the free-form argument string from the OCUDU configuration, builds the factory, and returns it.

```
std::shared_ptr<phy_tap_factory>
create_phy_tap_factory(unsigned           nof_rb,
unsigned           nof_ports,
const std::string& processor_arguments)
{
// Parse processor_arguments here if your plugin needs configuration.
```

```
auto ext_factory = std::make_shared<iq_tap_processor_factory>(nof_rb, nof_ports);
return std::make_shared<phy_tap_factory_impl>(std::move(ext_factory));
}
```

Once these three pieces are in place and linked into the OCUDU binary (see Build System), the plugin is compiled in but **not active by default**. It must be enabled at runtime through the configuration.

Step 4 - enable the plugin in configuration

Even with the plugin present in the binary, OCUDU will not invoke it unless `enable_phy_tap` is set to `true` under the `expert_phy` section. This lets the same binary be deployed with or without the plugin active without recompilation.

```
expert_phy:
enable_phy_tap: true
phy_tap_arguments: ""   # optional: passed verbatim to create_phy_tap_factory()
```

Or equivalently on the command line:

```
--expert_phy.enable_phy_tap=true
--expert_phy.phy_tap_arguments="key=value,..."
```

When `enable_phy_tap: false` (the default), OCUDU skips factory creation entirely and the plugin code is never executed. This means the plugin can be left linked in release builds and gated by configuration.

Chaining Processors with the Decorator Pattern

More complex use cases need to compose multiple processing stages - for example, running a custom DSP algorithm and simultaneously streaming telemetry to an external system - without either stage knowing about the other. The Decorator pattern achieves this: each stage wraps the previous one and implements the same `external_ul_processor` interface.

The pattern

```
create_phy_tap_factory()
└─ ul_epre_zmq_tap_factory::create()
└─ ul_epre_zmq_tap
└─ iq_tap_processor_factory::create()
└─ iq_tap_processor ← innermost, runs first
```

Each decorator calls the inner processor before (or after) its own work, then adds its own behaviour. The chain is assembled in the entry point from the argument string; no stage modifies the others.

Example: a ZMQ telemetry decorator

This decorator wraps any existing processor and streams per-subcarrier energy measurements (EPRE) to an external receiver over ZMQ after the last symbol of each slot. The ZMQ send is dispatched to a background worker so the critical path is not blocked by network I/O.

```
class ul_epre_zmq_tap : public external_ul_processor
{
public:
ul_epre_zmq_tap(std::unique_ptr<external_ul_processor> inner,
const std::string& zmq_address) :
inner(std::move(inner)),
zmq_backend(zmq_address),
worker("ULPhyTap", default_queue_size),
executor(worker)
{}

void process(resource_grid_writer&       grid_writer,
const resource_grid_reader& grid_reader,
slot_point slot, unsigned symbol,
/* PDU spans... */) override
{
// 1. Delegate to the wrapped stage first.
if (inner) {
inner->process(grid_writer, grid_reader, slot, symbol, /* ... */);
}
```

```
// 2. On the last symbol of the slot, compute and stream EPRE.
//    This is done after the inner stage so it sees any modifications
//    written back by the inner processor.
if (symbol == grid_reader.get_nof_symbols() - 1) {
send_epre_async(grid_reader);
}
}

void process_quiet(const resource_grid_reader& grid_reader,
slot_point slot) override
{
if (inner) inner->process_quiet(grid_reader, slot);
}

void process_prach(prach_buffer& buffer,
const prach_buffer_context& context) override
{
if (inner) inner->process_prach(buffer, context);
}

private:
void send_epre_async(const resource_grid_reader& grid_reader)
{
auto buffer = compute_epre(grid_reader); // reads from the grid, no allocation

// Hand off to the background thread. The critical path returns immediately.
executor.defer([this, buf = std::move(buffer)]() {
zmq_backend.send_buffer(*buf);
});
}

std::unique_ptr<external_ul_processor> inner;
zmq_server_backend zmq_backend;
task_worker worker;
task_worker_executor executor;
};
```

The decorator's factory

The decorator's factory wraps any existing `external_ul_processor_factory`, which makes it composable with any other stage without knowing its type:

```
class ul_epre_zmq_tap_factory : public external_ul_processor_factory
{
public:
ul_epre_zmq_tap_factory(std::shared_ptr<external_ul_processor_factory> inner_factory,
std::string zmq_address) :
inner_factory(std::move(inner_factory)),
zmq_address(std::move(zmq_address))
{}

std::unique_ptr<external_ul_processor> create() override
{
return std::make_unique<ul_epre_zmq_tap>(
inner_factory->create(), zmq_address);
}
```

```
private:
std::shared_ptr<external_ul_processor_factory> inner_factory;
std::string zmq_address;
};
```

Wiring the chain in the entry point

The entry point builds the chain from the argument string. Adding the ZMQ decorator requires no changes to `iq_tap_processor` or its factory - it is purely additive:

```
std::shared_ptr<phy_tap_factory>
create_phy_tap_factory(unsigned nof_rb, unsigned nof_ports,
const std::string& args)
{
// Start with the base processing stage.
std::shared_ptr<external_ul_processor_factory> factory =
std::make_shared<iq_tap_processor_factory>(nof_rb, nof_ports);

// If tap_ul_epre=<address> appears in args, wrap with the ZMQ decorator.
std::smatch m;
if (std::regex_search(args, m, std::regex(R"(tap_ul_epre=([^,]*))"))) {
factory = std::make_shared<ul_epre_zmq_tap_factory>(std::move(factory), m[1].str());
}

return std::make_shared<phy_tap_factory_impl>(std::move(factory));
}
```

At runtime this produces the chain **ZMQ decorator → custom processor**, assembled entirely from the configuration string, with no `if` branches inside any processing stage. Adding a further stage is another `if` block in the entry point and a new decorator class - nothing else changes.

Plugin Configuration

Enabling the PHY TAP plugin

Plugins are enabled and configured through the `expert_phy` section of the OCUDU configuration file:

```
expert_phy:
enable_phy_tap: true
phy_tap_arguments: "enable_quiet_processing=true,tap_ul_epre=tcp://*:5555"
```

Or equivalently via command-line flags:

```
--expert_phy.enable_phy_tap=true \
--expert_phy.phy_tap_arguments="enable_quiet_processing=true,tap_ul_epre=tcp://*:5555"
```

How arguments are passed to the plugin

The entire `phy_tap_arguments` string is passed verbatim to `create_phy_tap_factory()` as the `processor_arguments` parameter. Parsing is the plugin's responsibility. The convention used in the example plugin is comma-separated `key=value` pairs, parsed with standard regex:

```
std::smatch m;
if (std::regex_search(args, m, std::regex(R"(enable_quiet_processing=(true|false))"))) {
enable_quiet = (m[1].str() == "true");
}
```

Supported arguments

Argument	Type	Default	Description
`enable_quiet_processing`	`true|false`	`false`	Enables `process_quiet()` callbacks for slots with no uplink allocations. Disabled by default to avoid unnecessary CPU load on idle slots.
`log_level`	`none|error|warning|info|debug`	`warning`	Controls the verbosity of the plugin's logger.
`tap_ul_epre`	ZMQ address string	*(disabled)*	Activates the ZMQ EPRE telemetry decorator and binds to the given address, e.g. `tcp://*:5555`. See Chaining Processors.

Adding custom arguments

Any `key=value` pair can be added to the argument string. Custom arguments are ignored by the built-in parsing; only the plugin's own `create_phy_tap_factory()` implementation reads them. This means a plugin can expose its own configuration surface without any changes to OCUDU's configuration schema.

Build System

Build model

The PHY TAP plugin is compiled as a CMake **OBJECT library** that is linked directly into the OCUDU binary at build time. This avoids shared-library symbol resolution overhead at runtime while still keeping the plugin's source tree in a separate repository that can be developed and version-controlled independently.

A compile definition (`OCUDU_HAS_PHY_TAP`) is set on the OCUDU upper PHY target so that the rest of the build knows the tap is present and can conditionally activate the tap call sites.

Repository layout

```
phy_tap_plugin/
├── CMakeLists.txt ← top-level: sets source root, adds subdirectory
├── include/
│ ├── external_ul_processor.h ← core processing interface (implement this)
│ └── external_processor_factories.h ← factory interface and factory creation declarations
└── lib/
├── CMakeLists.txt ← wires the plugin into ocudu_upper_phy
├── phy_tap_factories.cpp ← exports create_phy_tap_factory()
├── phy_tap_impl.h ← adapter: phy_tap → external_ul_processor
└── external_processors/
├── CMakeLists.txt
├── external_processor_factories.cpp ← factory chain construction
├── iq_tap_processor_impl.cpp ← your processor implementation
└── ...                              ← additional stages / decorators
```

Top-level CMakeLists.txt

```
cmake_minimum_required(VERSION 3.14)
project(phy_tap_plugin)

# Expose the include directory to the lib subtree.
set(OCUDU_PLUGIN_INCLUDE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/include)
add_subdirectory(lib)
```

lib/CMakeLists.txt

```
# Object library: compiled into OCUDU, not a standalone .so.
add_library(ocudu_phy_tap OBJECT
phy_tap_factories.cpp)
```

```
target_include_directories(ocudu_phy_tap PRIVATE
${OCUDU_PLUGIN_INCLUDE_DIR})

target_link_libraries(ocudu_phy_tap
external_processors)

# Attach the plugin to the OCUDU upper PHY target.
target_link_libraries(ocudu_upper_phy ocudu_phy_tap)

# Tell the rest of the OCUDU build that the tap is active.
target_compile_definitions(ocudu_upper_phy PRIVATE -DOCUDU_HAS_PHY_TAP)

add_subdirectory(external_processors)
```

lib/external_processors/CMakeLists.txt

```
# All processor implementations go in this library.
add_library(external_processors
external_processor_factories.cpp
iq_tap_processor_impl.cpp)

target_include_directories(external_processors PRIVATE
${OCUDU_PLUGIN_INCLUDE_DIR})

target_link_libraries(external_processors
dsp_lib # any third-party DSP dependency
zmq # only if using the ZMQ telemetry decorator
ocudu_support)      # OCUDU utility types (span, task_worker, …)
```

Integrating the plugin into the OCUDU build

The plugin repository is added to the OCUDU source tree via `add_subdirectory()` in the top-level OCUDU `CMakeLists.txt`, typically guarded by a CMake option:

```
option(OCUDU_ENABLE_PHY_TAP "Build the PHY TAP plugin" OFF)

if(OCUDU_ENABLE_PHY_TAP)
add_subdirectory(path/to/phy_tap_plugin)
endif()
```

Build with the plugin enabled:

```
cmake -DOCUDU_ENABLE_PHY_TAP=ON ..
make -j$(nproc)
```

Key points

- **Object library, not shared library.** The plugin is compiled into the binary, not loaded at runtime via `dlopen`. This means it must be present at build time.
- **No ABI boundary.** Because the plugin is an object library, there is no need to manage C symbol visibility or worry about C++ ABI compatibility between the plugin and the main binary.
- **Third-party dependencies** belong in `external_processors`, not in the object library. This keeps the link graph clean and makes it easy to see what the plugin brings in.
- **The `OCUDU_HAS_PHY_TAP` definition** is what gates the call sites inside OCUDU. Without it, the tap code paths are compiled out entirely and have zero runtime cost.

Software Design Patterns

Design patterns are reusable solutions to recurring structural problems. OCUDU uses a curated set of well-known patterns consistently throughout the codebase. Recognising them when reading code - and reaching for them when writing new code - is an important part of contributing effectively.

The patterns below are grouped by category and annotated with where and why they appear in OCUDU.

Creational patterns

Factory / Abstract Factory

Problem: A component needs to create objects without knowing their concrete type.

In OCUDU: Protocol entities are constructed through factory functions or factory classes that read configuration and return the right concrete implementation behind an interface pointer. This keeps construction logic out of business logic and makes it straightforward to return a different implementation (e.g., a stub or a test double) by swapping the factory.

```
std::unique_ptr<mac_scheduler> make_mac_scheduler(const scheduler_expert_config& cfg);
```

Builder

Problem: Constructing a complex object requires many optional parameters and a specific assembly order.

In OCUDU: Configuration objects and test harness setups use a builder pattern to avoid multi-argument constructors that are easy to call incorrectly. The builder validates the configuration before constructing the final object. Have a look at the FAPI message builders for an example of this pattern, and check the unit tests here as well.

Structural patterns

Adapter

Problem: Two components have incompatible interfaces; one must be wrapped to look like the other.

In OCUDU: Layer boundaries regularly require adapters. The F1AP layer is a clear example. The DU-high holds a pointer to an `f1ap_message_handler` interface and calls it to send F1AP messages upward. Two concrete adapters implement that same interface:

- `f1ap_sctp_adapter` — wraps a real SCTP association. It serialises the ASN.1 message and hands the byte stream to the transport layer. This is the adapter used in a deployed system where the DU and CU run as separate network nodes.
- `f1ap_local_adapter` (fast path) — skips serialisation and SCTP entirely. It passes the already-decoded message object directly to the CU-CP handler in the same process. This is used when DU and CU are co-located, eliminating unnecessary encode/decode round-trips and loopback overhead.

The DU-high is never aware of which adapter is active. Swapping between split and co-located deployments is purely a wiring decision made at startup. This is one of the most heavily used patterns in the codebase.

Decorator

Problem: Behaviour needs to be added to an object without modifying its class.

In OCUDU: Instrumentation (metrics collection, logging, tracing) is layered on top of a core implementation using decorators that implement the same interface and delegate to the inner object. This keeps instrumentation code out of protocol logic.

```
// Wraps a real scheduler and records scheduling metrics around every call.
class scheduler_metrics_decorator : public mac_scheduler { ... };
```

Behavioural patterns

Strategy

Problem: An algorithm needs to be selectable at runtime or configuration time.

In OCUDU: The strategy pattern is the default way components interact. Every handler interface (`mac_scheduler`, `f1ap_message_handler`, `lower_phy_downlink_handler`, etc.) is a strategy contract: the holder knows only the interface, not the concrete type behind it. This means any component can be replaced - with a real implementation, a stub, a decorator, or an adapter - without touching its caller. Scheduling policies, HARQ retransmission strategies, and link adaptation algorithms are specific examples of this, but the pattern is pervasive across all layer boundaries.

Observer (Notifier / Event dispatcher)

Problem: One component needs to notify others when something happens, without knowing who is listening.

In OCUDU: Cross-layer events (slot indications, UE state changes, measurement reports) are dispatched through an observer/notifier pattern. Producers fire events into an event dispatcher; consumers register as listeners. This keeps producers and consumers decoupled - neither includes the other's header.

Command

Problem: An operation needs to be encapsulated as an object so it can be queued, deferred, or cancelled.

In OCUDU: Protocol procedures that span multiple slot boundaries are modelled as command-like objects or coroutine tasks. This allows the executor framework to schedule, prioritise, and cancel them independently of the protocol logic that issues them.

Template Method

Problem: The skeleton of a procedure is fixed, but certain steps vary between implementations.

In OCUDU: Common procedure scaffolding (timer management, state machine transitions, PDU assembly) is implemented in a base class. Subclasses override only the steps that differ. This avoids duplicating the procedural skeleton across multiple variants.

Null Object

Problem: A component requires a collaborator, but in some configurations that collaborator does nothing.

In OCUDU: Rather than littering code with `if (reporter != nullptr) reporter->report(...)`, OCUDU uses null-object implementations that conform to the full interface but do nothing. The `null_mac_metrics_notifier` is a typical example. This keeps calling code clean and makes the "no-op" case explicit.

References

Architecture

Clean Architecture - Robert C. Martin (Prentice Hall) Part 3 "Design Principles" dedicates one chapter to each of the five SOLID principles, with worked examples and the motivation behind each. Part 5 "Architecture" covers the Clean Architecture philosophy in depth.

Clean Code - Robert C. Martin (Prentice Hall) Practical guidance on writing readable, maintainable code. Covers naming, functions, comments, formatting, and error handling with concrete before/after examples.

Online references:

- The Clean Architecture - Robert C. Martin, 2012
- SOLID Relevance - Robert C. Martin, 2020

C++ Software Design

C++ Software Design - Klaus Iglberger (O'Reilly) A modern treatment of software design specifically for C++. Covers classic design patterns reimagined for C++17/20, including value semantics, type erasure, and the trade-offs between object-oriented and value-based approaches. Highly recommended for contributors working on the OCUDU core libraries.

OOP Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley) The original "Gang of Four" reference catalogue for software design patterns.

Head First Design Patterns - Eric Freeman & Elisabeth Robson (O'Reilly) An accessible introduction to design patterns with many examples and a step-by-step teaching style. A good starting point before tackling the GoF book.

Online references:

- Refactoring Guru - Design Patterns
- Game Programming Patterns

OCUDU Software Architecture Documentation

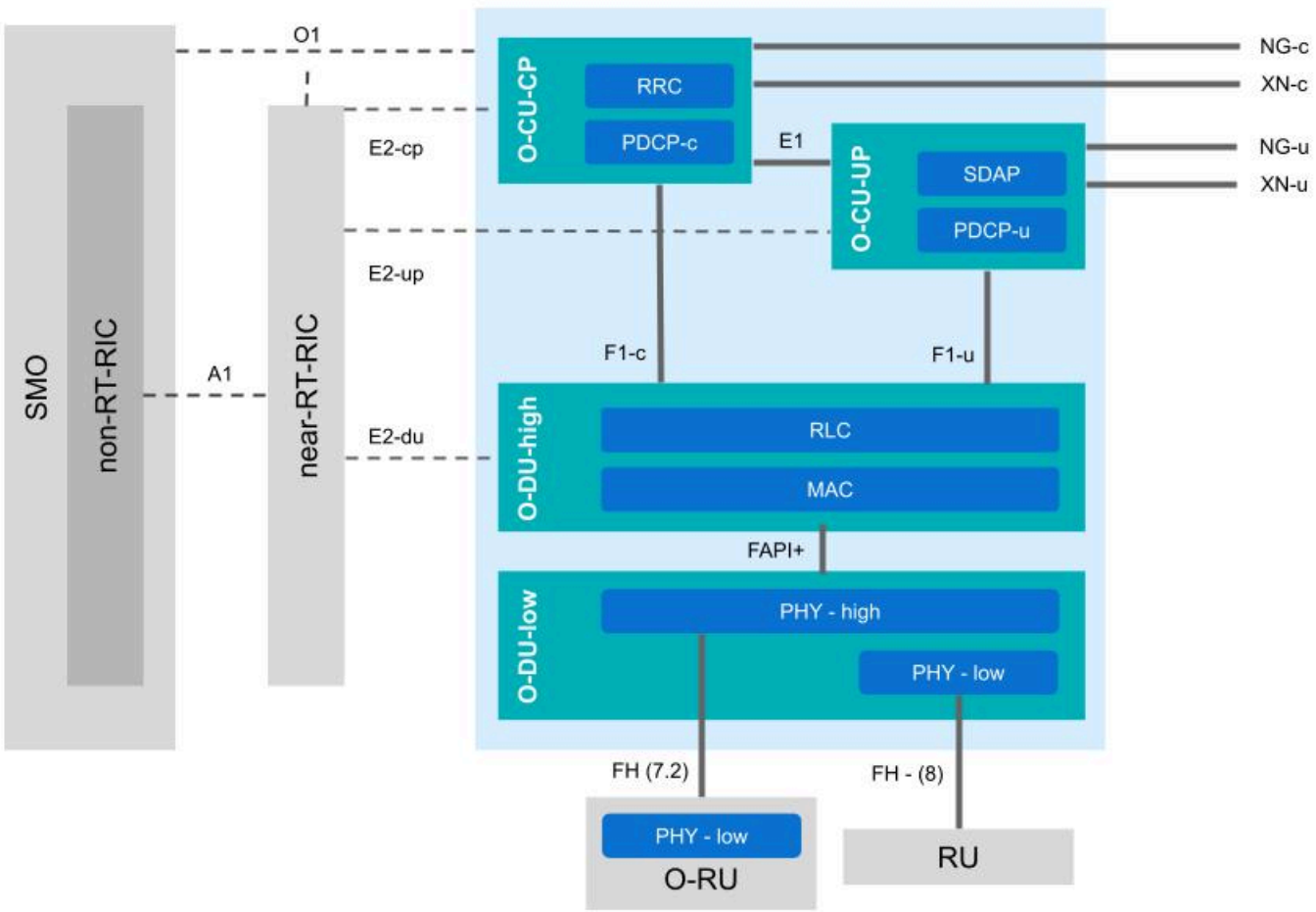
A primer on the O-RAN gNB architecture has already been outlined in the Knowledge Base, this can be found here.

This documentation aims to outline how this architecture is implemented in the OCUDU codebase. The function and implementation of each component will be discussed in subsequent sections.

Overview and Components

The code implements all of the components and interfaces in the below diagram. All of these elements have been implemented in software and are fully performant, customizable and compliant with the O-RAN standard. Users can also integrate 3rd-party RICs, RUs, and gNB components with OCUDU components.

- CU-CP
- CU-UP
- DU



Centralized Unit

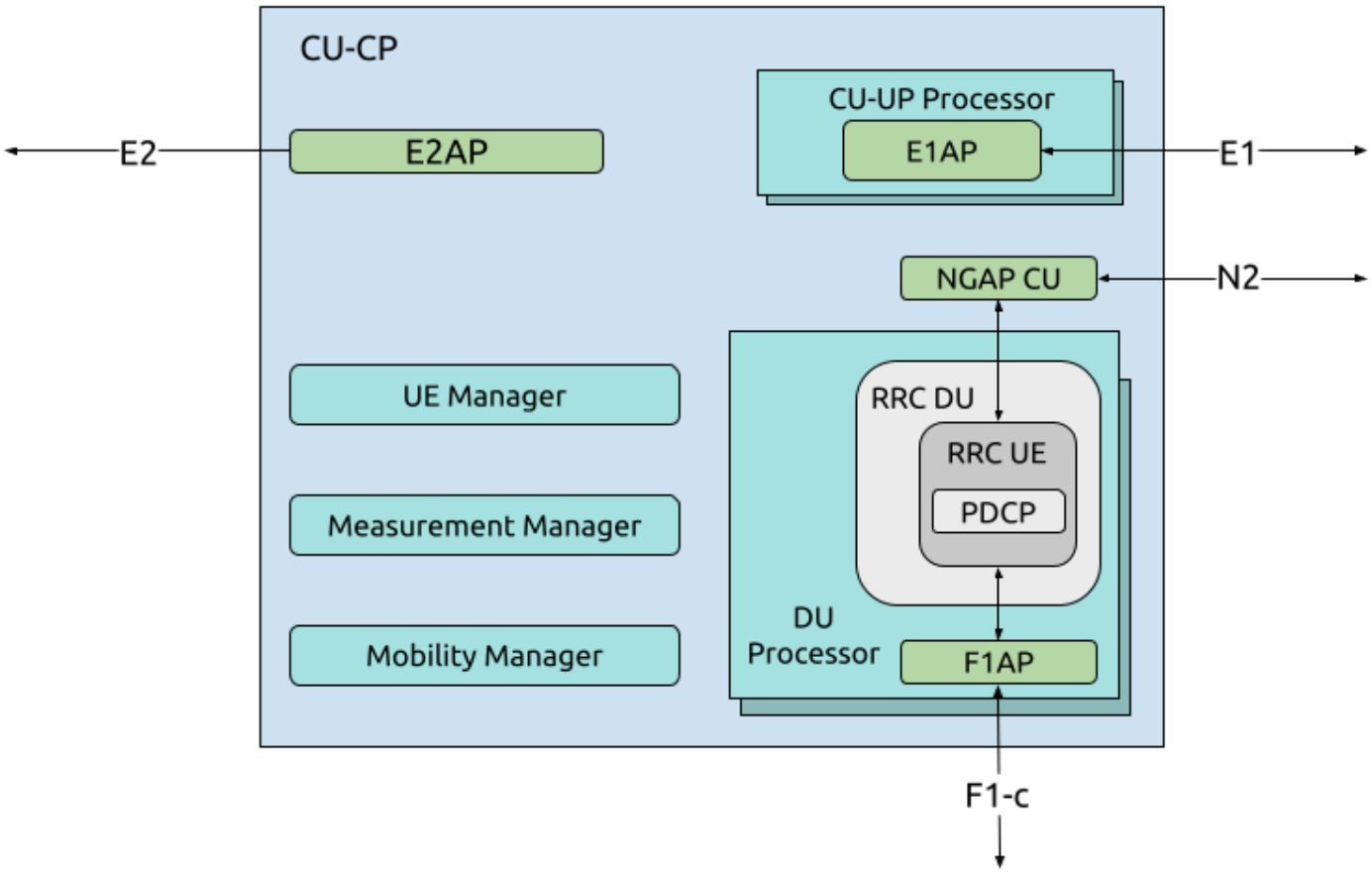
CU-CP

The CU-CP, or Central Unit - Control Plane, is responsible for the handling of control plane messaging, specifically, the control plane part of the PDCP protocol. In the CU-CP,...

CU-UP

The CU-UP, or Central Unit - User Plane, is responsible for the handling of user plane messaging. Specifically the user plane aspect of the PDCP and SDAP protocols. In the...

CU-CP



The CU-CP, or Central Unit - Control Plane, is responsible for the handling of control plane messaging, specifically, the control plane part of the PDCP protocol. In the CU-CP, there are five main components and four main interfaces. The CU-CP communicates directly with the 5G Core (via the N2 interface), the CU-UP (via the E1 interface), the DU-high (via the F1-c interface) and can also be connected to the near-RT RIC (via the E2 interface). This implementation takes a UE-centric approach.

Components

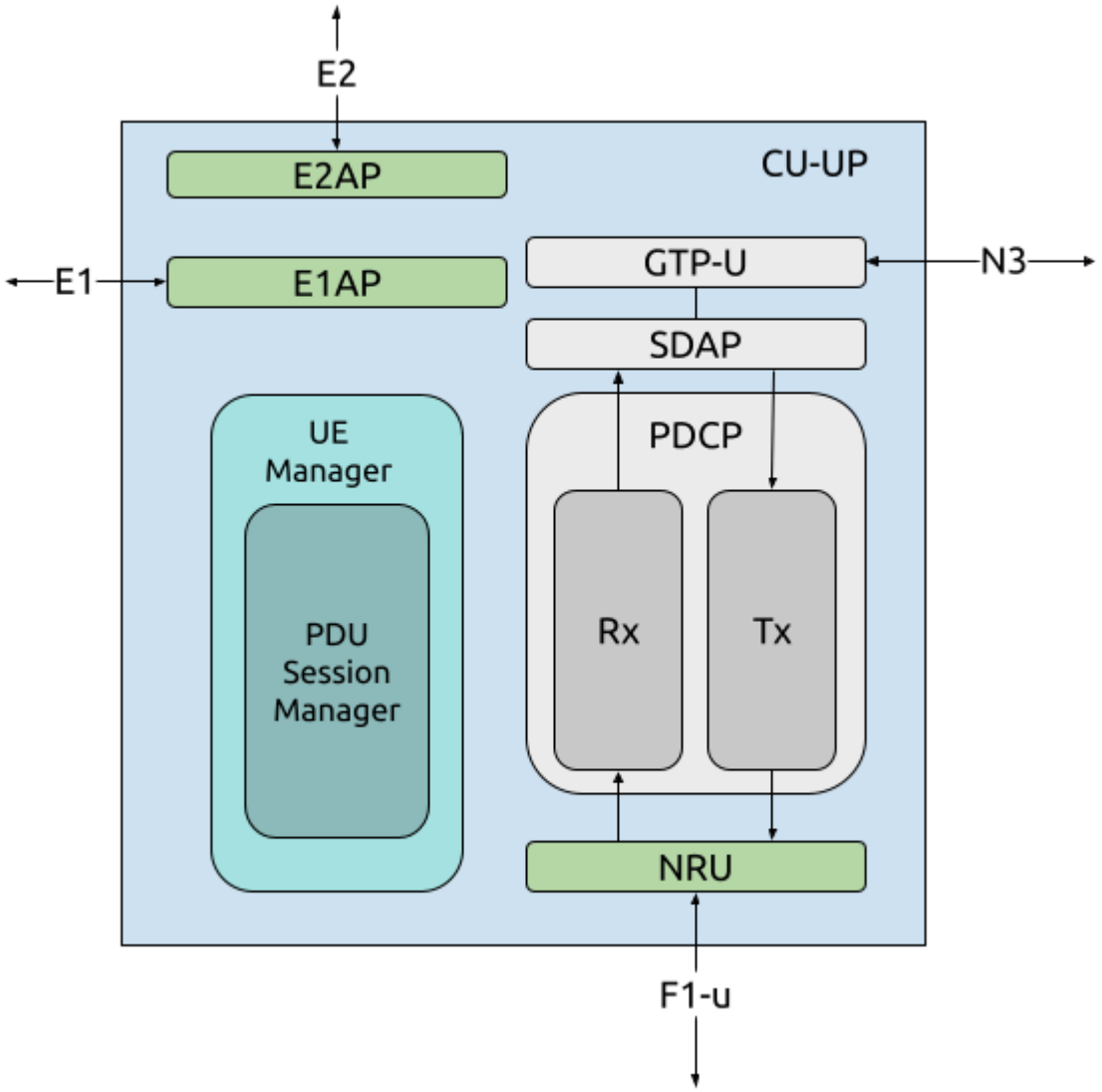
- CU-UP Processor: Custom component for handling each CU-UP connected to the CU-CP. Multiple CU-UPs can be connected to one CU-CP, as a result the E1AP is created inside the CU-UP processor for each connected CU-UP.
- DU Processor: Custom component for handling DUs for connected to the CU-CP. Each connected DU has its own DU processor with bundled F1AP, PDCP, and RRC procedures. Each UE connected to the DU also has its own RRC UE containing the PDCP processes.
- UE Manager: Custom component for managing connected UEs in the CU-CP. Responsible for adding and removing UEs and providing relevant UE information to other processes. Communicates information to/from the CU-UP, DU and Core.
- Measurement Manager: Custom component for managing cell measurement within the gNB.

- Mobility Manager: Custom component for managing UE mobility in the CU-CP.

Interfaces

- E2: Interface with the near-RT RIC.
- E1: Interface with the CU-UP.
- F1-c: Control plane interface with the DU.
- N2: Control plane interface with the 5G Core (AMF).

CU-UP



The CU-UP, or Central Unit - User Plane, is responsible for the handling of user plane messaging. Specifically the user plane aspect of the PDCP and SDAP protocols. In the implementation of the CU-UP, there are four main components and four main interfaces. The CU-UP communicates directly with the UPF in the 5G Core (via the N3 interface), the CU-UP (via the E1 interface), the DU-high (via the F1-u interface) and can also be connected to the near-RT RIC (via the E2 interface).

Components

- UE Manager: Custom component for managing connected UEs in the CU-UP. Responsible for adding and removing UEs and providing relevant UE information to other processes. Communicates information to/from the CU-CP, DU and Core.
- GTP-U: The GPRS Tunneling Protocol - User Plane (GTP-U) is responsible for transporting User Plane traffic to and from the UPF of the 5G Core via the N3 interface.

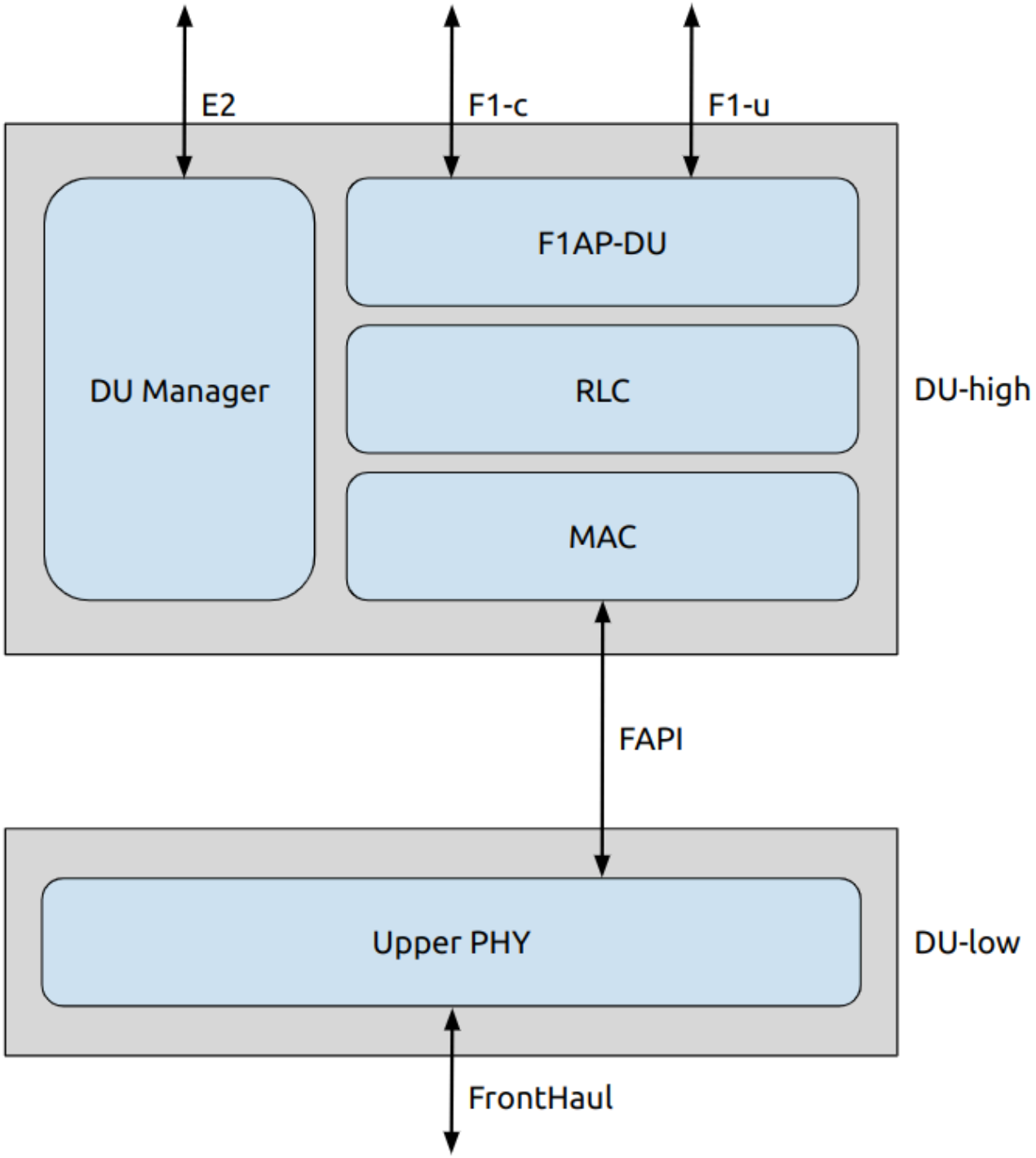
- SDAP: The Service Data Adaptation Protocol (SDAP) is responsible for adaption and mapping of Quality of Service (QoS) requirements on User Plan traffic.
- PDCP: The Packet Data Convergence Protocol (PDCP) is responsible for handling the User Plane data before/ after it enters the DU-high.

Interfaces

- E2: Interface with the near-RT RIC.
- E1: Interface with the CU-CP.
- F1-u: User plane interface with the DU.
- N3: User plane interface with the 5G Core (UPF).

Distributed Unit

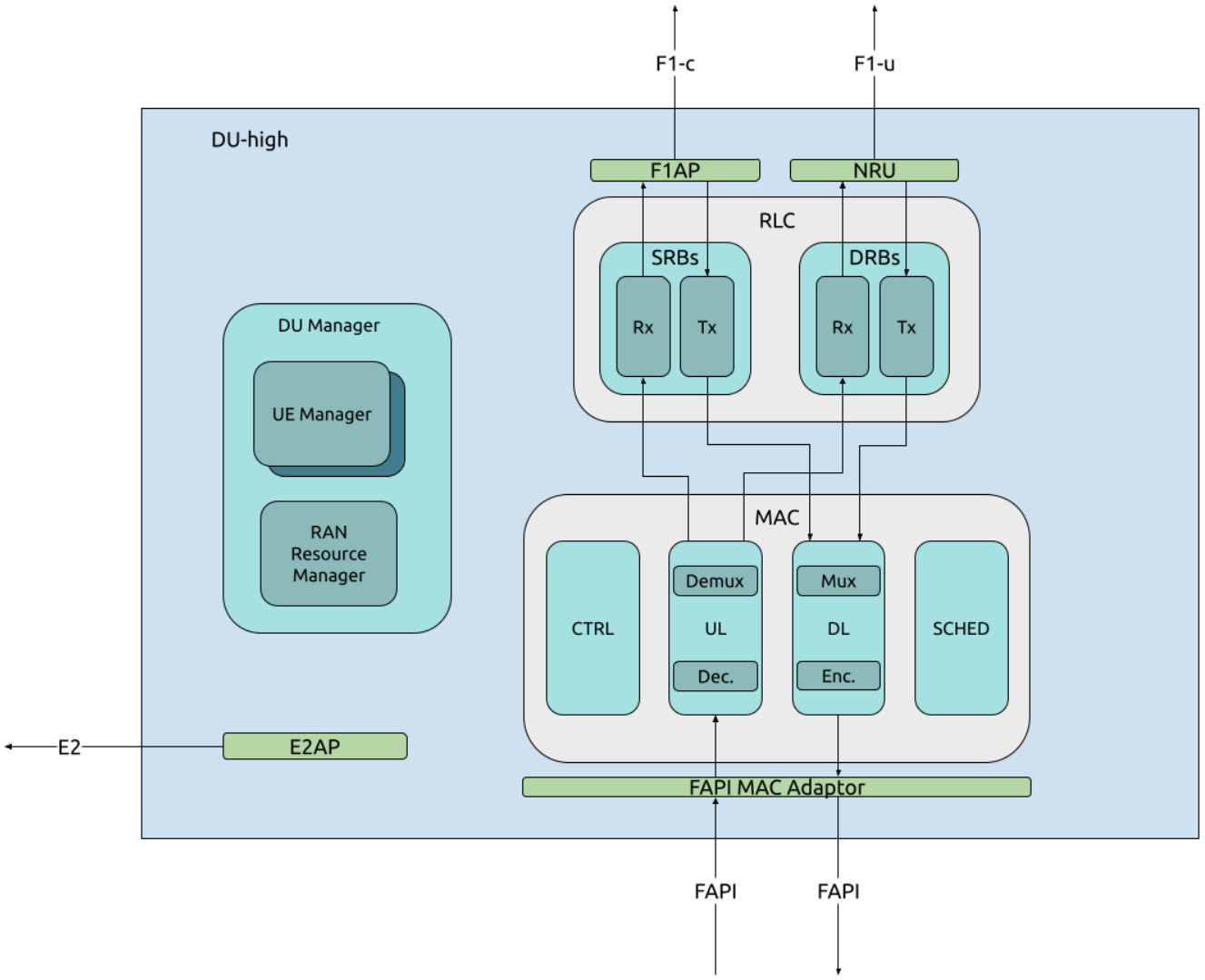
Components



Components

- DU-high
- DU-low

DU-high



The DU-high, or Distributed Unit - High, is responsible for the handling of both uplink and downlink traffic. Specifically the MAC and RLC processing of these signals. The DU-high, has three main components and three main interfaces. The DU-high communicates directly with the CU-CP and CU-UP via the F1-c and F1-u interfaces, the DU-low via the FAPI interface, and can also be connected to the near-RT RIC via the E2 interface.

Components

- DU Manager: The DU Manager is responsible for managing the DU, specifically the connected UEs and RAN Resources.
- RLC: The Radio Link Control (RLC) layer sits between the MAC in the DU-high and the F1AP/PDCP in the CU, and provides the transport services for SRBs and PRBs between these layers. It operates in three modes: transparent, unacknowledged and acknowledged.

- MAC: The Medium Access Control (MAC) is responsible for the encoding and decoding of MAC PDUs, scheduling uplink and downlink grants and other control services.

Interfaces

- E2: Interfaces with the nRT RIC.
- F1: Interfaces with the CU control and user plane.
- FAPI: Interface between the DU-high and DU-low.

DU Manager

The DU manager acts as a mediator in the process of configuring different layers of the DU. It is particularly important in the pre-operational stage, when it provides system information to the FAPI, MAC and F1AP that is essential to their operation. The DU manager is also essential during the operational stage, in the creation/reconfiguration/deletion of UEs and in the dynamic connection between bearers across different layers.

The main responsibilities of the DU Manager include:

- Configuring the MAC, FAPI and F1AP during the pre-operational stage with relevant DU system information, such as supported DU cells and their configuration
- Orchestration of the creation/reconfiguration/removal of UEs in the DU, in particular, the setting up of UE contexts in the MAC, FAPI and F1AP.
- Orchestration of the creation/reconfiguration/removal of UE bearers in MAC, RLC and F1AP, and connection of each bearer with its respective upper and lower layers.
- Management of commands received through E2.

The DU Manager contains the following components:

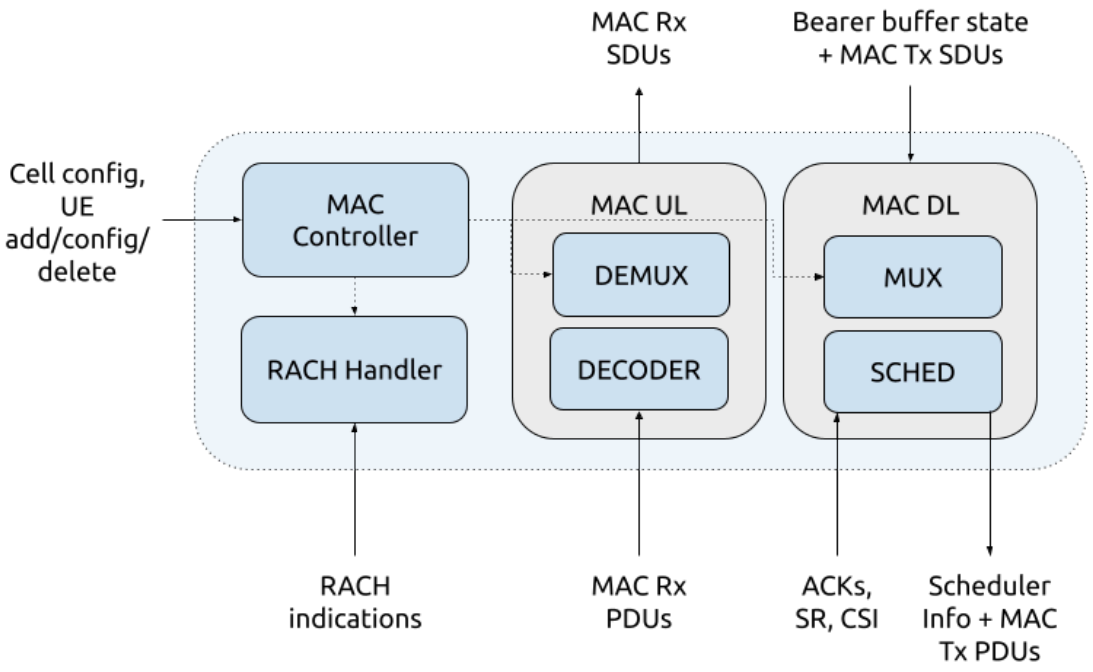
- UE Manager: Manages the UEs connected to the DU
- RAN Resource Manager: Manages the physical RAN resources associated with the DU

MAC

The main responsibilities of the MAC are:

- Encoding and decoding of MAC PDUs sent and received over the FAPI interface.
- Scheduling of DL/UL grants for System Information, Paging, UE data (RLC PDUs + MAC CEs) and Random Access, via the embedded gNB scheduler.
- Demultiplexing and forwarding of decoded MAC Rx SDUs to their respective logical channels.
- Handling of received MAC CEs.
- PRACH handling and RNTI allocation.

Architecture



The MAC interacts with FAPI, the RLC and the DU control plane. It is built from the sub-components shown above and exposes its functionality through abstract C++ interfaces (`include/ocudu/mac/`); the top-level `mac` interface (`mac.h`) hands out the per-cell and global handlers via accessors such as `get_slot_handler()`, `get_pdu_handler()`, `get_cell_manager()` and `get_ue_configurator()`. The three

peers map onto the figure's edges: the DU control plane configures the MAC, **SDUs are exchanged with the RLC**, and **PDUs are exchanged over FAPI**.

Sub-components

- **MAC Controller** (`mac_controller`) — translates DU configuration requests (add cell, add/reconfigure/remove UE) into commands for the other sub-components, applying them with minimal service disruption and without races (see Configuration Lifecycle).
- **RACH Handler** (`rach_handler`) — allocates RNTIs for received PRACH preambles and associates each RNTI with a DU UE index (see RNTI Management).
- **MAC UL Processor** (`mac_ul_processor`) — decodes received MAC PDUs and demultiplexes the SDUs and MAC CEs (see UL Data Path).
- **MAC DL Processor** (`mac_dl_processor`) — drives the scheduler and assembles the DL MAC PDUs (see DL Data Path and Scheduler).

DU control plane interface

Configuration and UE lifecycle events, driven by the DU manager:

- **Configuration** — `mac_cell_manager` (`add_cell`/`remove_cell`, plus per-cell `start`/`stop`/`reconfigure`) and `mac_ue_configurator` (`handle_ue_create_request`/`..._reconfiguration_request`/`..._delete_request`, returned as `async_task`s).
- **UL-CCCH / contention resolution** — `mac_ul_ccch_notifier` (`on_ul_ccch_msg_received`, `on_crnti_ce_received`) reports a received Msg3 to the upper layers, triggering UE creation in the DU.

RLC interface

User-plane SDUs, one bearer per logical channel, bound through `mac_logical_channel_config`:

- **Input (from RLC):** a `mac_sdu_tx_builder` supplies DL SDUs (`on_new_tx_sdu`) for DL PDU assembly and RLC bearer's buffer occupancy reports (`on_buffer_state_update`), which the MAC forwards to the scheduler, and
- **Output (to RLC):** a `mac_sdu_rx_notifier` (`on_new_sdu`) which the MAC uses to forward decoded UL SDUs to the RLC.

FAPI interface

The MAC's lower-edge handlers are translated to and from real FAPI messages by the FAPI adaptor, which lives outside `lib/mac`, so the MAC is written purely against these abstract handlers.

Inputs (from FAPI):

- `mac_cell_slot_handler` — delivers the per-slot timing trigger (`handle_slot_indication`) that drives scheduling and the generation of the slot's DL/UL results; also surfaces lower-layer processing errors

(`handle_error_indication`) and the completion of a cell stop (`handle_stop_indication`).

- `mac_cell_rach_handler` — delivers the PRACH preambles detected by L1 (`handle_rach_indication`), starting the Random Access procedure.
- `mac_cell_control_information_handler` — delivers the UL control feedback decoded by L1 and forwards it to the scheduler: PUSCH decode results (`handle_crc`), PUCCH/PUSCH UCI carrying SR, HARQ-ACK and CSI (`handle_uci`), and SRS channel reports (`handle_srs`).
- `mac_pdu_handler` — delivers the decoded UL MAC PDUs (`handle_rx_data_indication`) for decoding and demultiplexing in the UL data path.

Outputs (to FAPI): the MAC pushes each slot's results back through `mac_cell_result_notifier` (obtained from `mac_result_notifier::get_cell`):

- `on_new_downlink_scheduler_results` — the DL scheduling decisions (encoded PDCCH DCIs, SSB and PDSCH allocations), i.e. the `DL_TTI.request`.
- `on_new_downlink_data` — the assembled DL MAC PDU payloads (SI, RAR, paging, UE DL-SCH), i.e. the `Tx_Data.request`.
- `on_new_uplink_scheduler_results` — the UL grants (PUSCH/PUCCH), i.e. the `UL_TTI.request`.
- `on_cell_results_completion` — the end-of-slot sentinel signalling that all results for the slot have been delivered.

Concurrency and Parallelism

The MAC does not own any threads of its own. Instead, every MAC task runs on a `task_executor` provided at construction time, and the mapping of tasks to executors is what determines the MAC's concurrency model. This lets the same MAC code run single-threaded in tests and simulators, or spread across a worker pool in a real-time deployment, just by swapping the executor mapper.

Executors are supplied to the MAC through `mac_config` as three handles:

- `ctrl_exec` — a single, DU-wide control executor.
- `cell_exec_mapper` (`mac_cell_executor_mapper`) — per-cell executors.
- `ue_exec_mapper` (`mac_ue_executor_mapper`) — per-UE executors.

External events and thread-safety

The MAC's public entry points are invoked from threads it does not control: slot indications, RACH indications, CRC/UCI/SRS indications and received PDUs all arrive on FAPI threads, while configuration requests arrive on the DU manager's thread. These callers touch no MAC or scheduler state directly. Each entry point either:

- **dispatches the event onto a MAC executor** — e.g. the slot indication is enqueued onto the cell's `slot_ind_executor` and received UL PDUs onto the UE's `mac_ul_pdu_executor` — so that the actual handling runs serialized on the owning strand; or

- **forwards the event to the scheduler**, which manages its own thread-safety internally. CRC, UCI, SRS and error indications are passed straight into the scheduler from the calling thread; the scheduler buffers them in its own thread-safe queues and applies them later, in the cell's `slot_indication()` context.

This is the central invariant of the MAC's concurrency model: any state mutation must reach the owning strand (or the scheduler's internal queues) before it touches shared state. A new entry point that reads or writes MAC/scheduler state inline on the caller's thread, instead of hopping onto the appropriate executor or deferring to the scheduler, introduces a data race.

Serialization vs. parallelism

The MAC relies on **strands** (see `support/executors/strand_executor.h`) rather than locks to keep shared state consistent. A strand guarantees that tasks dispatched to it run one-at-a-time, in a serialized fashion, even when the strand is backed by a multi-threaded worker pool. Parallelism is therefore achieved *between* strands, not within one: tasks on different strands may run on different workers concurrently, while tasks on the same strand never overlap.

This yields three independent axes of parallelism:

- **Across cells** — each cell has its own strand, so different cells schedule and assemble their DL/UL grants in parallel.
- **Across UEs** — UEs are distributed over a bounded set of strands, so UL PDU processing for different UEs can proceed concurrently.
- **Control vs. data** — control-plane reconfiguration is funnelled onto its own serialized executor, decoupled from the per-cell real-time path.

Conversely, work that must not race is forced onto the *same* strand and thus serialized.

Per-cell execution

Each cell exposes two executors via the `mac_cell_executor_mapper`:

- `slot_ind_executor(cell)` — high-priority path for the periodic slot indication coming from FAPI.
- `mac_cell_executor(cell)` — default path for other, non-slot cell tasks.

The slot indication is the heartbeat of the cell: `mac_cell_processor::handle_slot_indication()` merely enqueues onto `slot_ind_executor`, and the actual work — invoking the **scheduler** for that cell, assembling the DL/UL PDUs, and forwarding the result to FAPI — runs in that executor's context. The scheduler therefore has no executor of its own; it is driven synchronously, once per cell per slot, on the cell strand. Because each cell uses a distinct strand, scheduling for different cells runs in parallel, while everything within a single cell is serialized.

The cell executors can be configured two ways (`du_high_executor_config::cell_executor_config`):

- **Dedicated worker per cell** — each cell gets its own thread, fully isolating cells.

- **Strand-based worker pool** — one strand per cell, all sharing a common worker pool. Slot indications use a higher strand priority than other cell tasks so they are never starved by lower-priority work.

Per-UE execution

UE-specific work runs on the `mac_ue_executor_mapper`:

- `ctrl_executor(ue)` — UE state changes (creation, reconfiguration, removal); infrequent.
- `mac_ul_pdu_executor(ue)` — decoding of received MAC PDUs and demultiplexing of UL SDUs.

UEs are mapped onto a bounded pool of strands (policy `per_cell` or `round_robin`, capped by `max_nof_strands`), trading off parallelism against memory. Within a UE's strand, control tasks take priority over UL PDU handling, which in turn takes priority over DL PDU handling. Two UEs on different strands process their UL PDUs concurrently; two UEs sharing a strand are serialized.

Control-plane serialization

Configuration requests from the DU manager (add cell, add/reconfigure/remove UE) are handled by the **MAC Controller** on the single DU-wide `ctrl_exec`, which is a serialized strand. Routing all reconfiguration through one strand is what lets the MAC Controller apply changes without locks and without racing against the per-cell real-time path — it hands off to and from the cell executors at well-defined points (e.g. cell activation/deactivation) rather than mutating cell state directly.

Summary

Task	Executor	Scope	Serialized within	Runs in parallel across
Slot indication, scheduling, DL PDU assembly	`slot_ind_executor`	per cell	a cell	cells
Other cell tasks (lower priority)	`mac_cell_executor`	per cell	a cell	cells
UL PDU decode / demux	`mac_ul_pdu_executor`	per UE	a UE's strand	UEs on different strands
UE control (add/reconfig/remove)	`ctrl_executor`	per UE	a UE's strand	UEs on different strands
MAC configuration (cells, UEs)	`ctrl_exec`	DU-wide	the whole MAC	nothing (fully serialized)

DL Data Path

The DL path is driven by the per-cell slot indication.
`mac_cell_processor::handle_slot_indication_impl()` runs once per cell per slot and:

1. Invokes the scheduler for that cell, obtaining a `sched_result`.
2. `assemble_dl_sched_request()` builds the `mac_dl_sched_result` — SSB (`ssb_helper`) and the encoded PDCCH DCIs (`encode_dci`) — and forwards it to FAPI via `on_new_downlink_scheduler_results`.
3. `assemble_dl_data_request()` builds the `mac_dl_data_result`, i.e. the actual MAC PDU payloads, and forwards it via `on_new_downlink_data`:
   - SIB / broadcast via `sib_pdu_assembler`,
   - RAR via `rar_pdu_assembler`,
   - UE DL-SCH via `dl_sch_pdu_assembler`,
   - paging via `paging_pdu_assembler`.
4. UL grants are forwarded via `on_new_uplink_scheduler_results`, and `on_cell_results_completion` closes the slot.

DL-SCH assembly (`dl_sch_pdu_assembler::assemble_newtx_pdu`) allocates a HARQ buffer and, for each scheduled logical channel, pulls SDUs from RLC (`mac_sdu_tx_builder::on_new_tx_sdu`) and encodes the subheader and payload; MAC CEs such as the UE Contention Resolution Identity and Timing Advance Command are multiplexed in, and the transport block is padded out. Retransmissions (`assemble_retx_pdu`) reuse the cached HARQ buffer rather than re-fetching SDUs. Once results are handed to FAPI, `update_logical_channel_dl_buffer_states()` reads the new RLC buffer state (`on_buffer_state_update`) and feeds it back to the scheduler (`handle_dl_buffer_state_update`).

UL Data Path

Received PDUs enter through `mac_pdu_handler::handle_rx_data_indication()` (`mac_ul_processor`). For each PDU the UE is resolved through the RNTI table and the work is dispatched onto that UE's `mac_ul_pdu_executor`, where `pdu_rx_handler::handle_rx_pdu()` runs:

- The UL-SCH PDU is unpacked into subPDUs (`mac_ul_sch_pdu::unpack`).
- SDUs are forwarded to the logical channel's RLC bearer (`mac_sdu_rx_notifier::on_new_sdu`).
- MAC CEs are parsed and forwarded: BSR via `handle_ul_bsr_indication`, PHR via `handle_ul_phr_indication`, both consumed by the scheduler.
- **CCCH (Msg3 / SRB0)** — when no UE exists yet for the TC-RNTI, the UL-CCCH message is delivered to upper layers via `mac_ul_ccch_notifier::on_ul_ccch_msg_received`; the SDU itself is pushed once the DU has created the UE (`push_ul_ccch_msg`).
- **C-RNTI CE** — `handle_crnti_ce()` resolves the existing UE for the carried C-RNTI, re-dispatches the remaining subPDUs onto that UE's executor, and notifies both the scheduler (`handle_crnti_ce_indication`) and upper layers (`on_crnti_ce_received`) for contention resolution.

Scheduler

The MAC contains the gNB scheduler but treats it as a self-contained subsystem behind `ocudu_scheduler_adapter` (implementing `mac_scheduler_adapter`). The MAC *pushes* events to the scheduler — `rach_indication`, CRC/UCI/SRS, BSR/PHR, DL buffer state, and UE add/reconfigure/remove — and *queries* it once per cell per slot through `slot_indication()`, whose returned `sched_result` drives the DL data path above. As described under Concurrency, the scheduler manages its own thread-safety for indication ingest. Its internal design is documented separately in `lib/scheduler/README.md`.

Configuration Lifecycle

Configuration is handled by the **MAC Controller** (`mac_controller`) running on `ctrl_exec`. Adding a cell registers it with the timing source, metrics, scheduler and DL handler. Cell removal unwinds these steps in reverse order. The MAC cell start/stop controls the scheduler cell activation/deactivation.

UE add, reconfigure and remove are asynchronous procedures (`ue_creation_procedure`, `ue_reconfiguration_procedure`, `mac_ue_removal_procedure`) implemented as coroutines that hop between the control executor and the per-cell/per-UE executors, setting up or tearing down the DL and UL contexts. These operations dispatched to the per-cell/per-UE executors should be latency bounded, so that other latency-sensitive tasks running in these executors do not get affected.

UE creation in the MAC and in the DU as a whole is deferred until Msg3 (see the RA Procedure). Removal deletes the scheduler context first — so no further grants are produced — then the UL/DL contexts, and finally the controller's UE entry.

RNTI Management

`rnti_manager` allocates a TC-RNTI for each detected contention-based PRACH preamble and C-RNTIs for UEs created from upper layer procedures (e.g. F1AP UE Context Setup Request). The `rnti_value_table` maps RNTI ↔ `du_ue_index` in a lock-free manner.

Radio Link Failure Detection

The `rlf_detector` (owned by the scheduler adapter) tracks three per-UE atomic counters: consecutive DL KOs (HARQ-ACK NACK/DTX), UL KOs (PUSCH CRC failures) and undecoded CSI (DTX). The thresholds come from `mac_expert_cell_config` (`max_consecutive_dl_kos`, `max_consecutive_ul_kos`, `max_consecutive_csi_dtx`, default 100). `handle_ack`/`handle_crc`/`handle_csi` — driven from the UCI decoder and CRC handler — reset the relevant counter on success and increment it on failure; the first time a counter reaches its threshold, an RLF is reported to the DU manager via `mac_ue_radio_link_notifier::on_rlf_detected`, deferred onto `ctrl_exec`. A received C-RNTI MAC CE resets all counters and invokes `on_crnti_ce_received`, cancelling a spurious RLF once the UE re-synchronises.

Procedures

Step-by-step descriptions of the MAC's runtime procedures (e.g. Random Access) are documented separately in `procedures.md`.

MAC Procedures

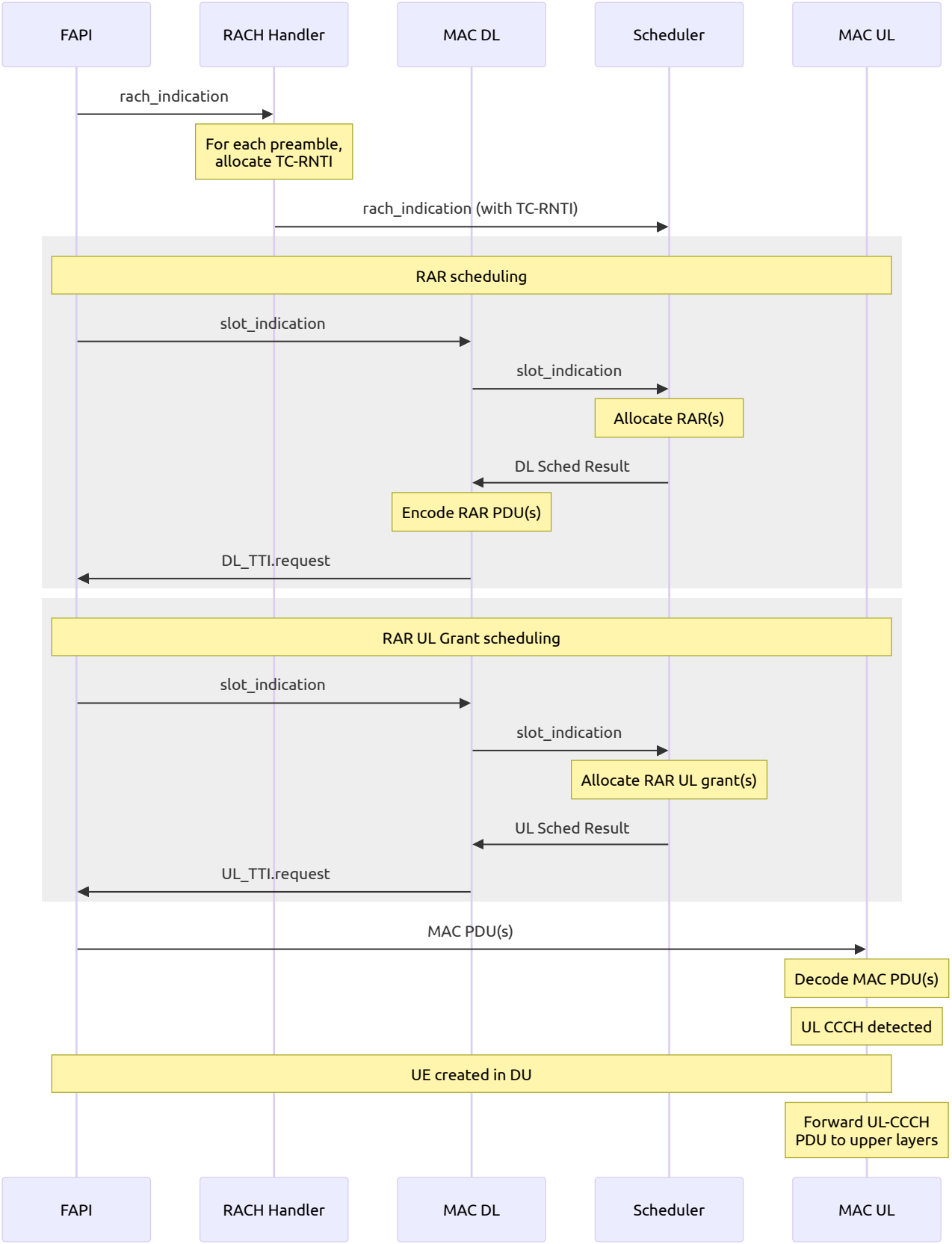
This document describes the runtime procedures driven by the MAC. For the MAC's architecture, interfaces and data paths, see README.md.

RA Procedure

The Random Access procedure is handled primarily by the DU MAC in the following steps:

1. The RACH handler manages the allocation of temporary RNTIs (TC-RNTIs) for each of the detected PRACH preambles in a given slot, and forwarding of these preambles and TC-RNTIs to the scheduler.
2. The scheduler is then responsible for allocating the RAR and Msg3 grant for each detected PRACH preamble.
3. The MAC DL processor has the role of encoding the MAC DL PDUs sent to FAPI.

This leads to the following messaging flow graph:



It is worth noting that the creation of a new UE in the DU is deferred until Msg3 is received. This design has the following advantages:

- UE Resources are not allocated unnecessarily for the cases when a phantom PRACH is detected, the UE fails to detect the RAR, or the Msg3 is not correctly decoded by the gNB.
- UE Resources are allocated after the RAR window has passed. With this design, any latency in the UE resource allocation thus won't affect the ability of the scheduler to timely schedule RARs. Notice that the RAR window in NR can be relatively short, depending on the type of service provided.

MAC Scheduler

The MAC scheduler allocates radio resources — time, frequency, and control channels — across all active cells in a DU. It runs once per slot per cell and produces a `sched_result` that the MAC layer uses to generate DL assignments and UL grants.

Component Hierarchy

```
scheduler_impl
├── sched_config_manager # Validates and distributes cell/UE config
├── cell_scheduler [per cell]    # All resources specific to one cell
│  ├── cell_metrics_handler # Per-slot metric aggregation for this cell
│  ├── cell_resource_allocator # Circular-buffer resource grid (16 slots)
│  ├── ssb_scheduler # SSB bursts
│  ├── pdcch_resource_allocator # PDCCH/DCI allocation
│  ├── si_scheduler # SIB1 and SI messages
│  ├── csi_rs_scheduler # CSI-RS
│  ├── ra_scheduler # Random access response (RAR)
│  ├── prach_scheduler # PRACH occasions
│  ├── pucch_allocator # PUCCH resources (HARQ-ACK, SR, CSI)
│  ├── uci_allocator # UCI scheduling (PUCCH or PUSCH mux)
│  ├── srs_allocator # Sounding Reference Signal scheduling
│  ├── paging_scheduler # Paging messages
│  ├── uci_scheduler # Periodic SR and CSI scheduling
│  ├── srs_scheduler # Periodic SRS management
│  ├── uci_indication_selector # Matches UCI indications to their grants
│  ├── ue_cell_repository # UE contexts of this cell (ue_cell objects)
│  ├── cell_event_manager # Queues and dispatches every event of the cell
│  └── ue_cell_scheduler # UE grants for this cell (via ue_scheduler)
└── ue_scheduler [per cell group]  # Shared UE state across a CA group
├── ue_repository # UE contexts of the cell group (ue objects)
├── cell_group_event_manager # Owns one event handler per cell of the group
├── ue_fallback_scheduler # SRB0 / contention-resolution grants
├── inter_slice_scheduler # Prioritizes slices for each slot
└── intra_slice_scheduler # PDSCH/PUSCH allocation within a slice
├── grant_params_selector # MCS, PRBs, HARQ selection
└── ue_cell_grid_allocator # Writes grants into the resource grid
```

A `ue_scheduler` is shared across all cells in a cell group (Carrier Aggregation), and holds the state that spans a UE's carriers. Each `cell_scheduler` holds a `ue_cell_scheduler` handle, which is an RAII view into the shared `ue_scheduler`.

The `uci_scheduler` and the `srs_scheduler` run after the schedulers of the common channels and before the UE scheduling step, as they have to reserve their periodic opportunities before any UE grant of the slot.

Slot Processing Flow

`scheduler_impl::slot_indication(sl_tx, cell_index)` is the entry point. It calls `cell_scheduler::run_slot()`, which executes in this order:

1. **Reset resource grid** — clear stale allocations for this slot.
2. **Events** — process everything that arrived since the last slot (`cell_event_manager`).
3. **SSB** — schedule Synchronization Signal Blocks.
4. **CSI-RS** — schedule CSI-RS if due.
5. **SI** — schedule SIB1 and SI-message windows.
6. **PRACH** — schedule PRACH occasions.
7. **RA** — schedule random access (RA) grants (e.g. RAR and Msg3) for detected RACH preambles.
8. **Paging** — schedule paging PDCCH and PDSCH.
9. **Periodic UCI** — schedule the SR and CSI PUCCH opportunities (`uci_scheduler`).
10. **Periodic SRS** — schedule the periodic and aperiodic SRS (`srs_scheduler`).
11. **UE scheduling** (`ue_cell_scheduler::run_slot`):
12. Advance UE state machines (DRX, timing advance, HARQ timers).
13. Schedule configured grant PUSCH opportunities, if configured.
14. Schedule SRB0 / fallback grants (`ue_fallback_scheduler`).
15. Prioritize slices for this slot (`inter_slice_scheduler::slot_indication`).
16. For each slice in priority order:
    - Schedule PDSCH retransmissions, then new transmissions.
    - Schedule PUSCH retransmissions, then new transmissions.
17. Post-process allocations (finalize PUCCH/UCI state).
18. Inject synthetic BSRs for the UEs that need a triggered UL grant.
19. **UCI indications** — match the UCI grants of the finished slot to their indications (`uci_indication_selector`).
20. **Logging and metrics** — flush the event log, the result log and the slot metrics.

Event Handling

Everything that reaches the scheduler outside a slot indication — RACH, CRC, UCI and SRS indications, buffer status and power headroom reports, MAC CEs, paging and SI requests, positioning requests, UE creation, reconfiguration and deletion — is an event. Events arrive from other executors, so none of them are applied where they arrive:

1. **`cell_event_manager`** (`cell_event_manager.h`) queues the event in a lock-free MPMC queue, with the payloads that do not fit in the callback taken from a pool of their own. The queue holds every payload the pools can hand out, so an event type is only ever limited by its own pool. One queue per cell means the events of a cell keep their arrival order, whatever their type.
2. The queue is drained at the start of the cell slot indication, in the cell executor. An event that only touches the state of the cell is applied there.

3. An event that needs the state shared by a UE's carriers is handed to the `cell_group_event_handler` of the cell and applied synchronously. It reports back whether it applied the event, so that the cell logs it and accounts for it in its metrics only when it did.

`cell_group_event_handler` (`cell_group_event_handler.h`) groups the two interfaces that the cell calls: `cell_group_ue_config_handler` for the UE configuration requests, and `cell_group_ue_indication_handler` for the UE indications and for the outcomes that a cell derives for one of its UEs while processing its own events. `cell_group_event_manager` owns one implementation of it per cell of the group. The cell scheduler creates its `ue_cell_scheduler` handle before its `cell_event_manager`, so the handler already exists when the manager is given the reference.

The cell also aggregates the DL buffer occupancy updates that it receives for a bearer since the last slot, and relays only the resulting update.

Resource Grid

`cell_resource_allocator` is a circular ring buffer over `cell_slot_resource_allocator` entries — one per slot. Each entry contains:

- `sched_result` — the scheduling decisions (PDCCHs, PDSCHs, PUSCHs, etc.)
- A DL and UL `carrier_subslot_resource_grid` per SCS — a Symbol × CRB bitmap used for collision detection and availability checks.

Allocators receive a `cell_slot_resource_allocator&` and write into it directly. The ring buffer provides a lookahead window so allocators can inspect and reserve resources in future slots (e.g., PUSCH scheduled K2 slots ahead of the PDCCH slot).

RAN Slicing

The scheduler supports multiple RAN slices, each configured with a `slice_rrm_policy_config` (min/max RB limits, S-NSSAI). Two scheduling layers handle slicing:

Inter-slice scheduler (`slicing/inter_slice_scheduler.h`) — runs once per slot and produces a priority-ordered queue of `ran_slice_candidate` objects for DL and UL. Priority is computed from slice SLA parameters, recent utilization, and RB limits.

Intra-slice scheduler (`ue_scheduling/intra_slice_scheduler.h`) — consumes one slice candidate at a time and allocates PDSCH/PUSCH grants for UEs in that slice. Within a slice, UE ordering is determined by the slice's `scheduler_policy`.

Scheduling Policies

`scheduler_policy` (`policy/scheduler_policy.h`) is the pluggable per-slice algorithm interface:

- `compute_ue_dl_priorities()` / `compute_ue_ul_priorities()` — assign a `ue_sched_priority` to each UE candidate.
- `save_dl_newtx_grants()` / `save_ul_newtx_grants()` — called after allocation to update internal state.
- `add_ue()` / `rem_ue()` — track UE membership.

Available implementations (in `policy/`) include Round-Robin with QoS weighting and time-domain fair scheduling.

UE Context

UE state is split into two levels:

`ue` (in `ue_context/`) — per-UE, cell-group-wide state:

- Logical channel repository and DL/UL buffer occupancy.
- DRX controller and timing advance manager (shared across cells).
- Link to all serving `ue_cell` objects.

`ue_cell` (in `ue_context/`) — per-UE, per-cell state:

- Active BWP configuration and HARQ entities.
- Channel state manager (CQI, RI, PMI from UCI indications).
- Link adaptation controller (translates CQI/SINR to MCS).
- Fallback mode flag (set on repeated HARQ failures; cleared by MAC).

The `ue` objects live in the `ue_repository` of the cell group; the `ue_cell` objects live in the `ue_cell_repository` of the cell that owns them, which the cell scheduler owns.

Configuration Management

`sched_config_manager` (`config/sched_config_manager.h`) is the single point of entry for all configuration:

- **Cell config** — validates `sched_cell_configuration_request_message` and builds a `cell_configuration` object used throughout the scheduler.
- **UE config** — validates `sched_ue_creation_request_message` / `sched_ue_reconfiguration_message`, builds a thread-safe config snapshot, and emits a `ue_config_update_event` that the UE PCell queues like any other event.

Configuration changes are never applied mid-slot; they are queued as events and applied at the start of the next `run_slot()` call, in their arrival order relative to the indications of the same UE.

Logging

The scheduler writes to the `SCHED` logger via two loggers in `logging/`: a result logger (`Slot decisions` lines) and an event logger (`Processed slot events` lines). See log_reference.md for the full format of every log line and field.

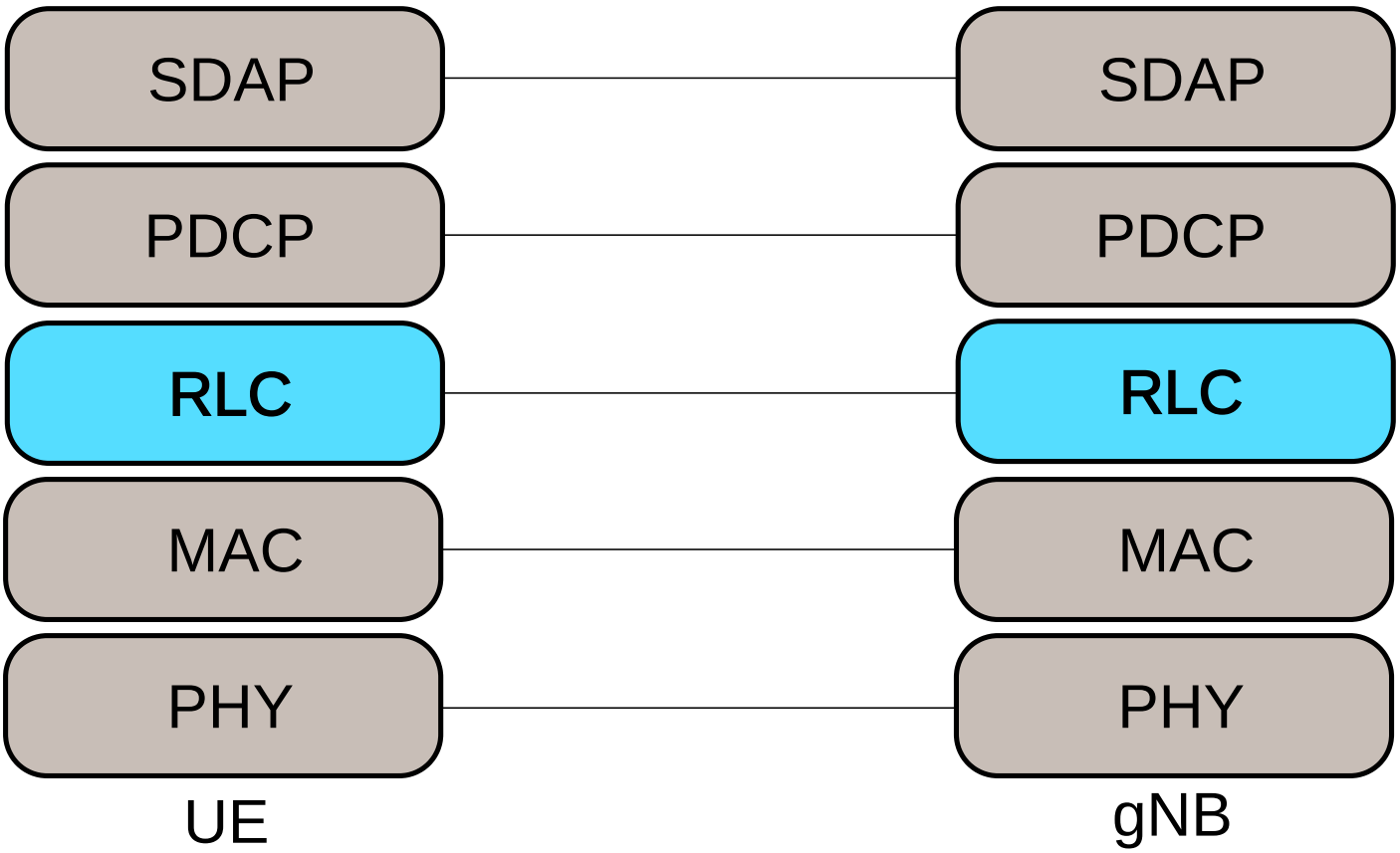
Directory Layout

```
lib/scheduler/
├── scheduler_impl.{h,cpp}      # mac_scheduler implementation
├── scheduler_factory.cpp # create_scheduler() factory
├── cell_scheduler.{h,cpp}      # Per-cell orchestrator
├── cell/ # Resource grid, HARQ entities, PRB tracking
├── common_scheduling/ # SSB, SI, CSI-RS, RA, PRACH, paging schedulers
├── config/ # Cell/UE configuration management and validation
├── logging/ # Event logger, result logger, metrics handler ([log_reference.md]
(log_reference.md))
├── pdcch_scheduling/ # PDCCH/DCI allocation
├── policy/ # Scheduling policy interface and implementations
├── pucch_scheduling/ # PUCCH resource allocation
├── slicing/ # RAN slice instances and inter-slice scheduler
├── srs/ # SRS resource allocation
├── support/ # MCS tables, BWP helpers, DMRS, power control
├── uci_scheduling/ # UCI allocation (PUCCH and PUSCH mux)
├── ue_context/ # UE and UE-cell state objects
└── ue_scheduling/ # UE grant scheduler, fallback, intra-slice allocator
```

The event handling spans both levels: `cell_event_manager.{h,cpp}` and the interfaces in `cell_group_event_handler.h` sit at the top level, next to `cell_scheduler`, and `ue_scheduling/cell_group_event_manager.{h,cpp}` holds the cell-group side.

Radio Link Control (RLC)

RLC layer connects the MAC and the F1AP/PDCP, as shown in the figure below, and provides that transfer services to these layers.



It does this, by providing three different modes:

Transparent Mode (TM)

TM is used only in control signaling (SRB0 only) and provides data transfer services without modifying the SDUs/PDUs at all.

Unacknowledged Mode (UM)

UM can only be used in data traffic (DRBs only) and provides data transfer services with segmentation/reassembly. It is usually used in delay-sensitive and loss-tolerant traffic, as it does not provide re-transmissions.

Acknowledged Mode (AM)

AM can be used in data and control traffic (mandatory for SRBs, optional for DRBs) and provides data transfer services with segmentation and ARQ procedures. This mode is usually used for traffic that is more loss-sensitive, but more delay-tolerant.

More details about our implementation can be found here.

RLC Acknowledged Mode (AM)

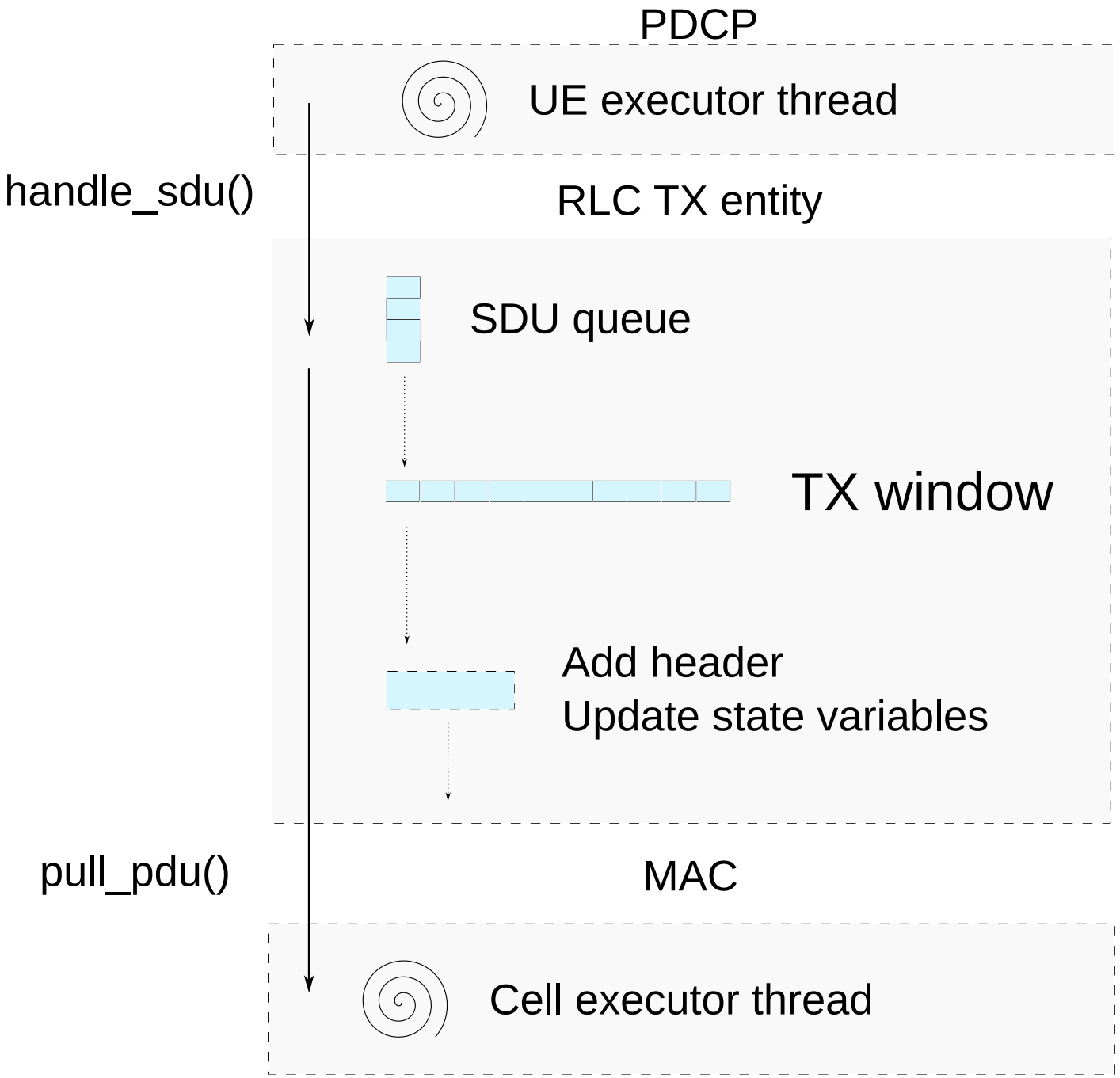
The RLC AM provides data transfer services with segmentation/concatenation and ARQ. It does this by keeping two windows, one for TX and another for RX, which store information regarding the transmitted/received SDUs. Unlike LTE, each window entry will contain information regarding a – single – SDU and its associated Sequence Number (SN).

Depending on the provided space in each transmission opportunity that is indicated by lower layer (MAC), the RLC may split SDUs into smaller SDU segments. The RX entity is responsible to store the received SDU segments in the RX window, and to pass the assembled SDU to the upper layer when it is fully received.

To provide ARQ services, the RX entity will generate *Status Reports* with the information of which SDUs/SDU segments have been received. This will be transmitted to the peer RLC entity, which will use the NACK information to trigger re-transmissions. The Tx entity will keep information regarding the transmitted SDUs on its TX window, so that it can retransmit the SDUs requested by its peer. This information is freed when the SDUs have been sequentially ACKed.

Data transmission

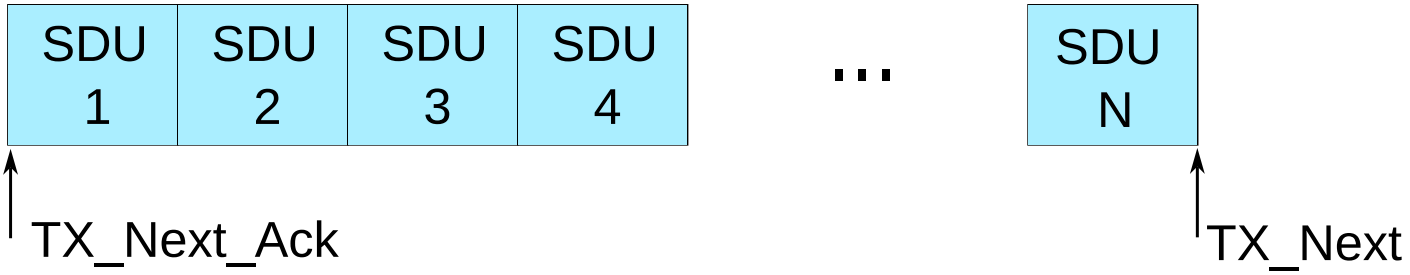
In the figure below, we can see a simplified illustration of the RLC AM Tx entity. There, we can see two threads do the work of pushing SDUs into the RLC and pulling PDUs from the RLC.



The thread from the *UE executor* will push SDUs from the F1/PDCP into a thread-safe queue. The thread from the *Cell Executor* will call the *pull_pdu()* method to pull PDUs from the RLC. The MAC will let know the RLC the size of the grant that it can fill, and the TX entity of the RLC will fill the payload with a new PDU, a retransmission or a status report. The *pull_pdu()* can be called multiple times in a single slot.

When a PDU is pulled, the RLC entity will generate either a status report, an RETX, an SDU segment, or a new PDU, in that order of priority. When generating a new PDU, the RLC will keep the information about the RLC SDU in the TX window, most notably copy of the SDU bytes, and the next Segment Offset (SO), i.e the progress in case of segmentation.

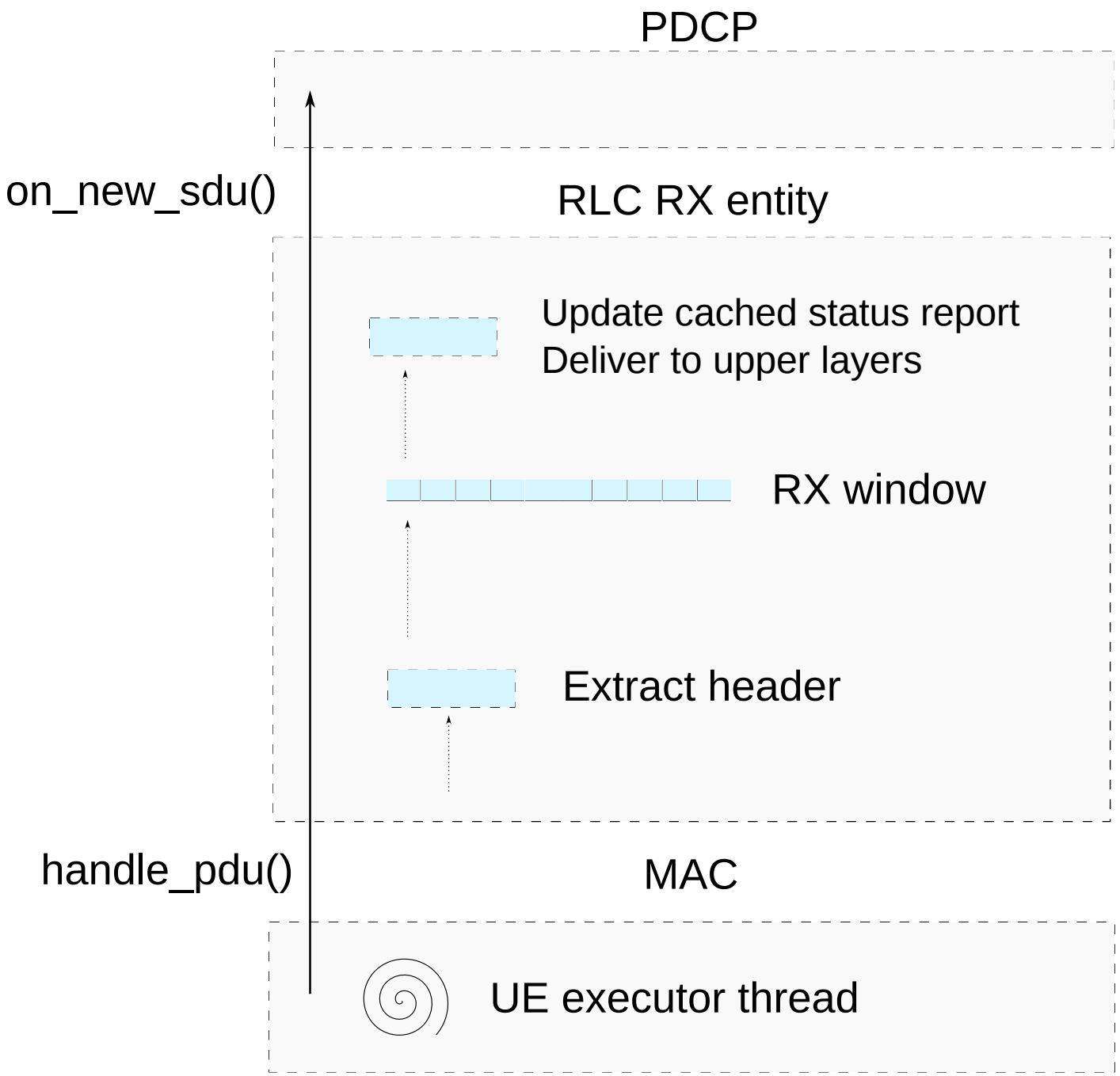
This will be kept until the transmitted PDUs are ACKed in sequence. This is illustrated in the figure below. There, we can see that the TX window will maintain two state variables: *Tx_Next_Ack* and *Tx_Next*. *Tx_Next_Ack* refers to the bottom of the TX window, i.e., the SN next to the last PDU that has been sequentially ACKed. *Tx_Next* is the SN that will be assigned to the next PDU.



We will detail the RETX and Status Report generation later on, but first it is important to detail the reception of RLC data PDUs.

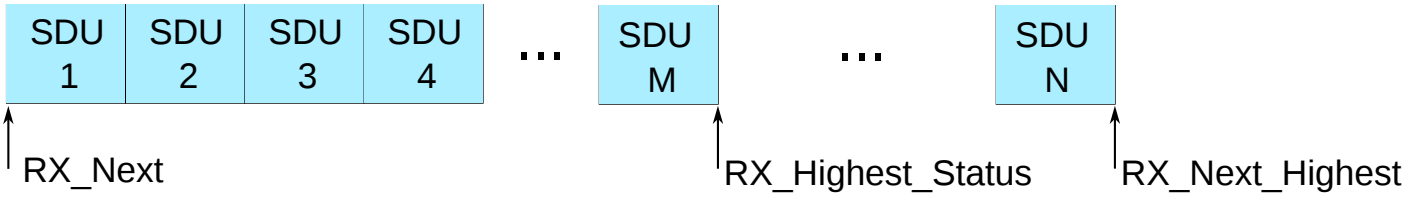
Data reception

We can see in this figure a simplified illustration of the RLC AM Rx entity. There we can see that only a single thread will push PDUs into the RLC Rx entity, the *UE executor* thread. When receiving a PDU, first the thing the entity will do is to check whether this is a Data PDU or a Control PDU (i.e. a status report.)



If it is a Data PDU, the SDU or SDU segment will be added/appended to the RX window, using the received SN. If the SDU has been fully received, SDU will be passed to the upper layers. The RX window will release the SDU information, when all SDUs have been received in order.

This is illustrated in this figure, where we can see four state variables: *RX_Next*, *RX_Highest_Status*, *RX_Next_Status_Trigger* and *RX_Next_Highest*. There *RX_Next*, the lower edge of the RX window, will contain the first PDU that has not been fully received. *RX_Next_Higest*, the higher edge of the Rx window, is the highest PDU received. The RX window will also keep *RX_Highest_Status*, which is the SN of the first SDU that is considered lost, as determined by the *t-Reassembly* timer. Finally, the *RX_Next_Status_trigger*, will keep the SN that triggered the *t-Reassembly*. This is for updating *RX_Highest_Status* to the first known lost PDU, when *t-Reassembly* expires and new losses are detected.

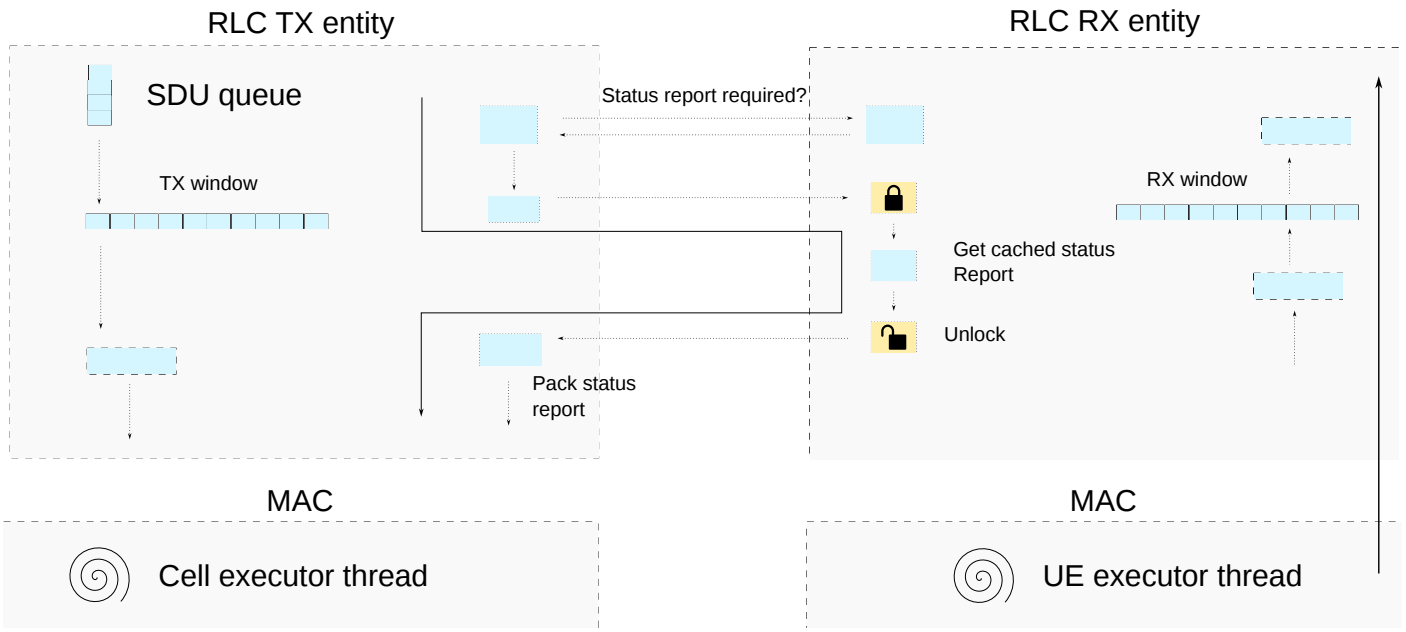


ARQ and status reporting

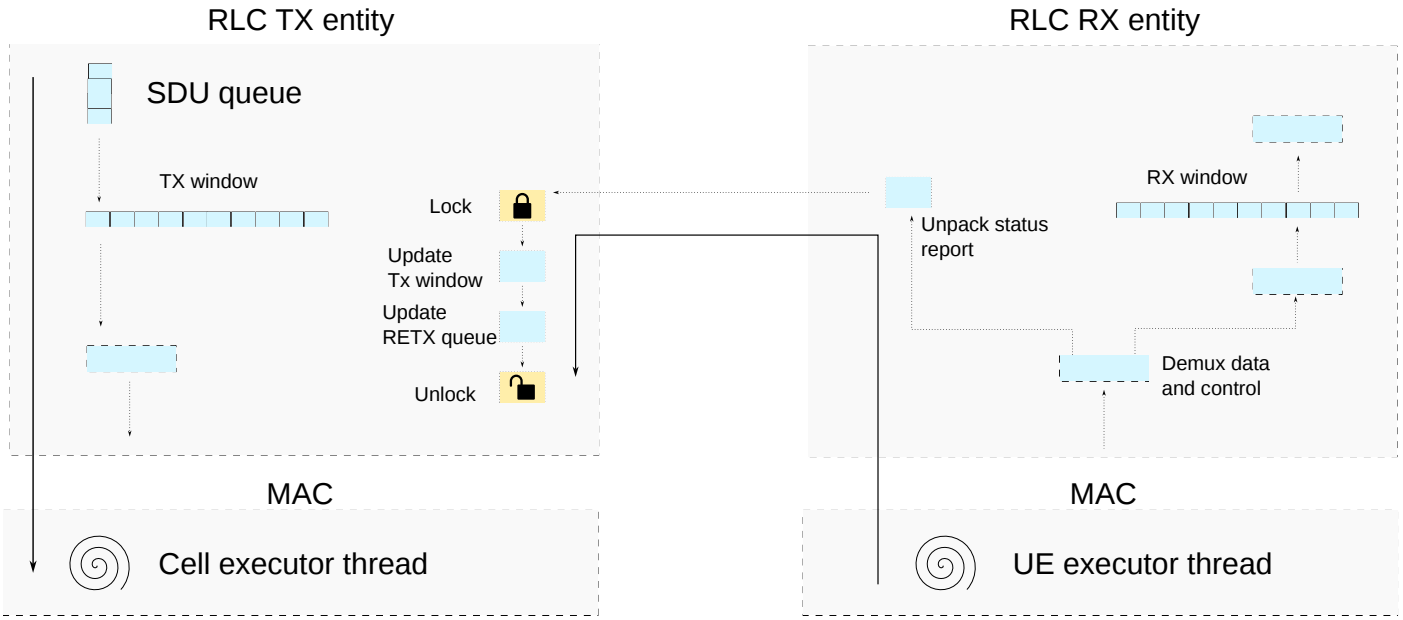
When the RLC receives a PDU with the header's *Polling Bit* set, and *t-Status-Prohibit* is not running, the RLC entity must transmit a status report to its peer in the next transmit opportunity. The TX entity will check whether the status report is required, by checking a boolean in the RX entity, that is set upon receiving the polling bit.

If the status report is required, the TX entity will retrieve a cached status report from the RX entity. This cached status report is updated at the reception of every PDU, to avoid blocking blocking the MAC generating a status report during the *pull_pdu()*.

An illustration of the process of generating the Status Report can be found in this figure.



When the RLC receives a Status Report, it must be passed to the TX entity for processing. The TX entity will use the received status report to update the TX window and the RETX queue. Because both the *UE executor* thread and *Cell Executor* thread can update both the TX window and RETX queue, both of these variables need to be protected with a lock. An illustration of the process of handling the Status Report can be found in this figure.

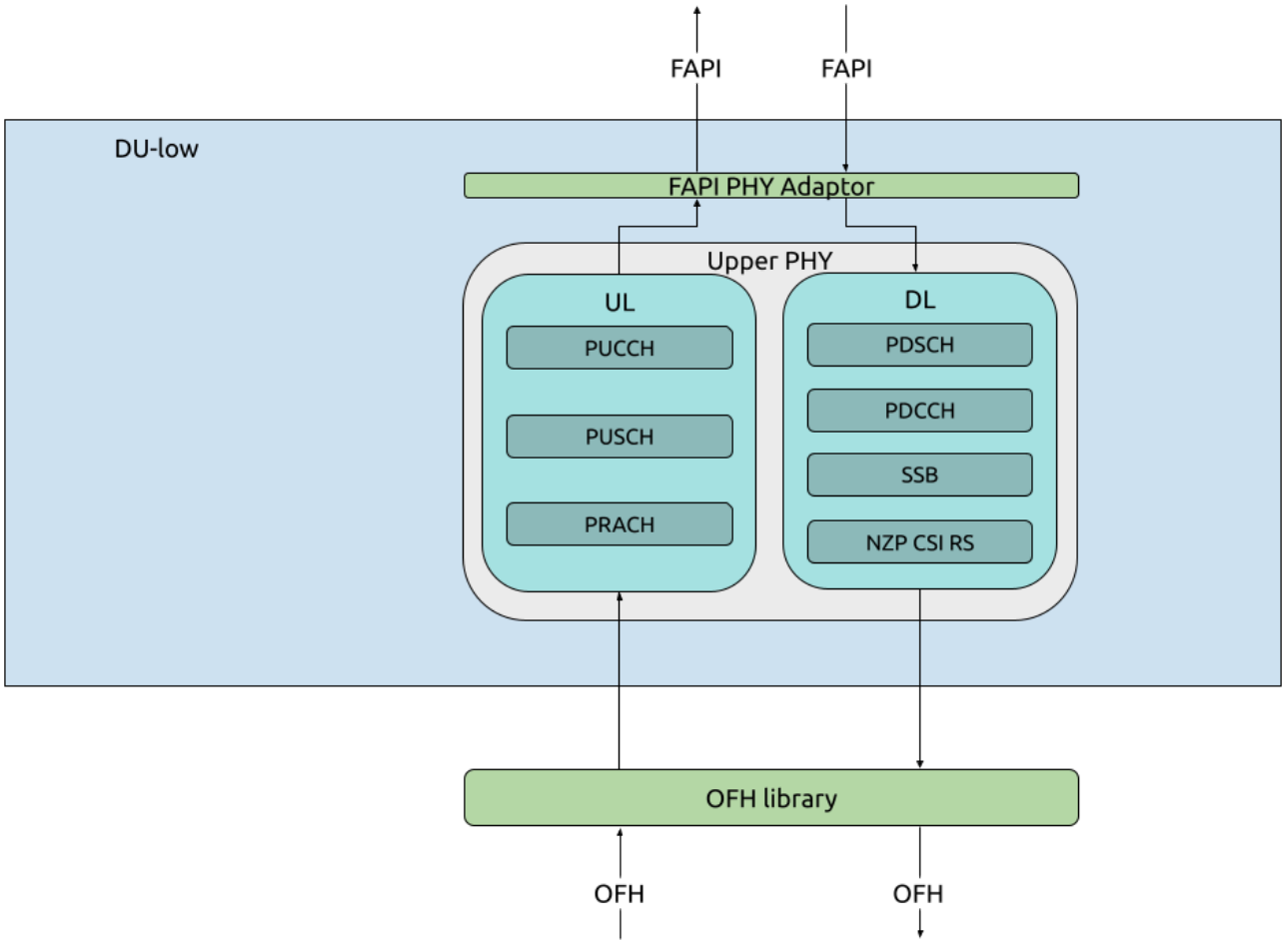


MAC buffer status reporting

The MAC needs to know the size of the buffer in the RLC TX entity to give appropriate grants. As polling by the MAC is inefficient when there are many inactive UEs and bearers, the RLC will push the new state of the buffer whenever this value is updated. This can be done when a PDU is pulled, when a PDU is received, or when a timer expires that, for example, requires an update to the status report.

As this can be done from any thread, cached sizes of all queues and any pending status report are kept, so we can update the buffer state with minimal blocking to the MAC.

DU-low



The DU-low, or Distributed Unit - Low, is responsible for the handling of both uplink and downlink traffic. Specifically, it handles the Upper PHY processing related to these signals. The DU-low, contains only the Upper PHY and has two main interfaces. The DU-low communicates directly with the DU-high via the FAPI interface, and with the RU via the Open FrontHaul (OFH) interface. The lower PHY processing is carried out in the RU. This architecture is specific to ORAN Split 7.2.

Components

- Upper PHY: The Upper PHY handles the processing of UL and DL signals coming from and to the RU.

Interfaces

- FAPI: Interfaces with the MAC in the DU-high.
- OFH: Interfaces with the lower PHY in the Radio Unit (RU).

Code Style Guide

This document establishes the basic guidelines and recommendations for C and C++ programming styles within OCUDU code base. Many of the guidelines are C++ specific, however some can still be applied to C. The main goal of this document is to help establish a consistent programming style throughout OCUDU code base, and improve the readability and maintainability of code committed by users.

This document is heavily inspired in LLVM's coding standards, please refer to this if you would like to take a further look into coding standards and best practices.

Contents:

Language and libraries

C & C++ version

Mechanical source aspects

Source Code Formatting

Style Aspects: High Level Issues

Self-contained Headers

Style Aspects: Low Level Issues

Naming Conventions

Recommendations

Function and Class Length

Self Generating Documentation

The OCUDU repository uses Doxygen to generate API documentation directly from the annotated C++ source files. In order to contribute to the API

Commit Formatting

When committing code to OCUDU codebase it is important that commit messages are clear and succinct. This means having clear guidelines for

Language and libraries

C & C++ version

The OCUDU project is written in standard C++17, thus it is important to avoid the use of compiler specific extensions to ease code portability and adding support for additional toolchains in the future.

If you need to use custom features such as builtin functions, guard them behind a macro or a function and always provide a fallback mechanism for toolchains where this feature is not available.

OCUDU is compiled and tested using the following toolchains with the following minimum requirements:

- GCC v11.4.0 (or later)
- Clang v14.0.0 (or later)

Use of the C++ standard library

It is really encouraged to use the C++ standard library to avoid reinventing the wheel for many common programming tasks and utilities.

When writing C++ code, it is preferable to use the C++ library functions instead of falling back to the C library if possible. For example: `std::this_thread::sleep_for()` vs `::usleep()`.

Additionally, OCUDU code base provides many additional support utilities and custom abstract data types. So before implementing a new utility or container, please check if it has been already implemented or if an existing one could be extended.

When both C++ and OCUDU support libraries provide similar functionality, and there is a reason to not favor the C++ implementation, it is generally preferable to use OCUDU support library, especially for custom containers found in the ADT folder.

Mechanical source aspects

Source Code Formatting

The OCUDU project uses a defined set of formatting rules, and heavily relies on clang-format for automatic source code formatting. You must use the custom clang-format file in the root of the repository, see here for more information about the options found in this file.

If you are interested in specific rules regarding spaces, maximum width, etc., then please check the contents of the above file.

When adding new commits to the repository, please make sure you have formatted your new changes with clang-format **before** submitting them.

Comments

Comments have to be regarded as an essential part of the code, as they greatly facilitate reviewers' and maintainers' tasks. From this perspective, comments should describe what the code is trying to do and why, without getting into the implementation details at a micro level and, most importantly, targeting an audience that may not be as familiar as you are with the topic at hand.

The following brief sections provide a very rough introduction to how developers are expected to comment code. A more thorough guide can be found here.

Comment Guidelines

- Write comments as English prose, using proper capitalization, punctuation, etc., ending them with a period sign. Words are spelled in American English, as given by the main entry of The Merriam-Webster Dictionary.

- Comment lines should start with a double slash for *normal* comments and with a triple slash for *documentation* comments. The double/triple-slash format should be used always, also for multiline comments.

```
/// \brief Does something.
///
/// More details about the function.
void some_func(){
// Now we print a simple text.
fmt::print("Some comment.");
}
```

- Comments should always start on a new line above the code that is being commented. Specifically, documentation comments should precede the class or function declarations and not go inside their body. Normal comments cannot be placed next to the code or outside the body of a function.

```
// Some general comment outside the body of a function - WRONG!

/// Does something. - OK
void some_func(){
// Now we print a simple text. - OK
fmt::print("Some comment.");

fmt::print("Something else."); // Side comment. - WRONG!
}
```

- In cases where indentation is helpful to support the content of the comment, the suggested form is to start the line with one or more "greater than" (>) signs, depending on the indentation level. For instance, the comments in the example below suggest that the PUCCH configuration is a step within the UL configuration.

```
uplink_config srsran::config_helpers::make_default_ue_uplink_config(const cell_config_builder_p
{
// > UL configuration.
uplink_config ul_config{};
ul_config.init_ul_bwp.pucch_cfg.emplace();

// >> PUCCH configuration.
auto& pucch_cfg = ul_config.init_ul_bwp.pucch_cfg.value();

// ... more code ...
}
```

- C-style comments `/* */` are, in general, not allowed. The only exceptions are file headers (see below) and the documentation of parameters in a function call (very useful when, e.g., passing a `bool` or a `nullptr` as arguments).

```
// Hard to see what "true" and "nullptr" actually refer to.
obj.method(a, b, true, nullptr);
// Easy to see what the input values actually do.
obj.method(a, b, /*enable_X=*/true, /*options=*/nullptr);
```

- Merge requests should **not** contain lines of code that have been commented out. If you really need to do it for documentation purposes or maybe for debug printing, use `#if 0` and `#endif`. These nest properly and are better behaved in general than C-style comments.

File Headers

Every source file of the project should have a copyright header and a short description of the basic purpose of the file. The standard header looks like the example below.

```
/*
*
* Copyright 2021-2023 Software Radio Systems Limited
*
* This file is part of srsRAN.
*
* srsRAN is free software: you can redistribute it and/or modify
* it under the terms of the GNU Affero General Public License as
* published by the Free Software Foundation, either version 3 of
* the License, or (at your option) any later version.
*
* srsRAN is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU Affero General Public License for more details.
*
* A copy of the GNU Affero General Public License can be found in
* the LICENSE file in the top-level directory of this distribution
* and at http://www.gnu.org/licenses/.
*
*/
```

Header Guards

All header files should have an include guard to prevent double inclusion. The OCUDU codebase uses the `#pragma once` directive, which is widely supported by common compilers. Unlike conventional include guards (via `#ifndef` and `#define`), neither a unique identifier nor a closing expression (`#endif`) is required.

The following example shows this:

```cpp
/*
* File header...
*/

#pragma once

#include "foo.h"
#include <file.h>

namespace ocudu {
// ...
} // ocudu
```

#include Style

Try to include a minimal list of `#include` and keep it clean of redundant header files as dependencies change. To that end, it is OK to exploit the fact that includes propagate transitively. So, if for instance *foo.h* already includes *bar.h*, only *foo.h* needs to be included when using functions or classes from both files.

The include list should be immediately after the header file comment, and after the include guards if working on a header file. Include files should be listed in the following order:

1. Main module header.
2. Local and private headers.
3. OCUDU or subproject headers (ocudu/…, etc).
4. System library includes.

Keep each category sorted lexicographically by the full path and avoid adding newlines between categories or include directives. The main module header should be always the first in the list. Sub-project headers should be included before ocudu headers (from most specific to least specific eg: more specific ocudu subfolders first).

```
#include "my_class_header.h"            // category 1
#include "private_module_utils.h"       // category 2
#include "srsenb/hdr/public_header.h"   // category 3
#include "ocudu/adt/bounded_vector.h"   // category 3
#include <string>                       // category 4
#include <vector>                       // category 4
```

Use C++ library header files in C++ files instead of using the C library headers, eg: `cstring`, `cassert`, `cstdint`, etc.

As a final note, `clang-format` will lexicographically sort all includes files automatically for you.

Language and compiler aspects

Treat compiler warnings as errors

Avoid submitting code that generates compiler warnings. Sometimes you may get a false positive warning, in this case find a way to suppress it.

Code Portability

Try to write portable code under all circumstances. If you are under the situation where you need to do something that is not portable, then put it behind a documented interface so it is centralized in a single place and not scattered in different places.

Avoid RTTI and Exceptions

In C++ code, **do not** use exceptions or RTTI (runtime type information eg: `dynamic_cast<>`).

Use of auto

In general, use auto when it makes the code more readable and easier to maintain. Typical cases would be for iterators or complex template types. Some examples:

```
auto *p = std::unique_ptr<int>(new int);  // good: easy to infer that p is an int*
auto i = my_map.begin();                  // good, easy to see it is an iterator and avoids
writing std::unordered_map<std::string, int>::iterator
auto a = my_func();                       // bad: what type does my_func return?
```

Style Aspects: High Level Issues

Self-contained Headers

Header files should be self-contained, this means that they can compile on their own.

In general, a header may be implemented by one or more source files. Each of these source files will include the header file that defines the interface first in the include list (see include style).

Using `#include` Sparingly

Headers included in other header files propagate transitively. For instance, if header file `foo.h` includes header file `bar.h`, any file that includes `foo.h` will also include `bar.h` implicitly. To avoid long compilation times, includes of large header files should be taken with care.

When you are about to include a large header file `large_include.h` in another header file `my_module.h`, consider the following improvements:

1. Can `large_include.h` be included by the respective source file `my_module.cpp` instead? This will avoid the transitive propagation of `large_include.h` by any file that includes `my_module.h`.
2. Can `my_module.h` just include a subset of the `large_include.h` included headers? E.g. `my_module.h` may just need access to interface definitions, rather than derived class implementations.
3. If 1. is not possible without altering code, consider making forward declarations of `large_include.h` types and use reference/pointers in `my_module.h`. Then, place the `large_include.h` header in `my_module.cpp` instead.

Using “Internal” Headers

When writing a module, avoid adding implementation details, helper classes and utility functions in the public interface header that are only meant to be used by the module internally. Instead, create a private header file in the same directory as the source files and include it locally. This will avoid polluting the public interface with unnecessary dependencies reducing recompilations.

Use of `namespace`

Avoid opening namespace blocks inside a .cpp file that spans for the complete file.

When adding a free function implementation in a .cpp file, avoid opening namespace blocks. Instead, use the namespace qualifier to ensure that the function definition matches the new declaration.

```
// foo.h
namespace ocudu {
int foo(const char *s);
}

// foo.cpp
#include "foo.h"
using namespace ocudu;
int ocudu::foo(const char *s) {
// ...
}
```

Avoid this:

```
// foo.cpp

#include "foo.h"
namespace ocudu { // Namespace block.
int foo(char *s) { // Mismatch between "const char *" and "char *"
// ...
}
} // end namespace ocudu
```

The above code snippet will add a new overload of ocudu::foo instead of providing a definition of the existing function declared in the header. To make things a bit worse, this error will be detected by the linker towards the end of the build, instead of being detected instantly by the compiler.

A final word about namespace qualification: when you need to call a C library or system function like `read()`, `socket()`, etc, you should prefix the calls with `::` such as `::socket()`, this will avoid potential name clashes and unexpected behavior.

Using “early exits” and `continue`

When reading code, keep in mind how much state information and how many previous decisions have to be remembered by the reader to understand a block of code. Aim to reduce indentation where possible, unless doing so would make the code more readable. One great way to do this is by making use of early exits and the `continue` keyword in long loops.

This example does not use early exits:

```
int *my_function(my_class *p) {
if (!p->f1() && p->f2() &&
my_other_function(p)) {
// ... some long code ....
}

return nullptr;
}
```

This can be transformed to:

```
int *my_function(my_class *p) {
// f1 for this p is true because ...
if (p->f1()) {
return nullptr;
}

// f2 for this p is false because ...
if (!p->f2()) {
return nullptr;
}

// Something else ...
if (!my_other_function(p)) {
return nullptr;
}

// ... some long code ....
}
```

Similarly, in loops:

```
for (int i : my_vector) {
if (i % 7 == 0) {
if (check_something(i)) {
do_something(i);
} else {
// ... some long code ....
}
}
}
```

Can be transformed to:

```
for (int i : my_vector) {
if (i % 7 != 0) {
continue;
}
if (check_something(i)) {
do_something(i);
continue;
}
// ... some long code ....
}
```

Avoid `else` after a `return` statement

Following the same reasoning as the previous point, avoid using `else` or else if after a statement that interrupts control flow like return, break, continue, goto, etc.

For example:

```
case 2: {
if (something) {
v = get_buffer_type_1();
if (!v.is_valid()) {
error_string = "Invalid buffer type 1";
return -1;
} else {
break;
}
} else {
v = get_buffer_type_2();
if (!v.is_valid()) {
error_string = "Invalid buffer type 2";
return -1;
} else {
break;
}
}
}
```

Can be transformed to:

```
case 2: {
if (something) {
v = get_buffer_type_1();
if (!v.is_valid()) {
error_string = "Invalid buffer type 1";
return -1;
}
} else {
v = get_buffer_type_2();
if (!v.is_valid()) {
error_string = "Invalid buffer type 2";
return -1;
}
}
break;
}
```

Or optimally to:

```
case 2: {
if (something) {
v = get_buffer_type_1();
} else {
v = get_buffer_type_2();
}

if (!v.is_valid()) {
error_string = (something) ? "Invalid buffer type 1"
: "Invalid buffer type 2";
```

```
return -1;
}
break;
}
```

This way helps to understand the code better as you need to keep track of less context and reduces indentation.

Use of Static Helper Functions

It is very common to write small loops that just compute a boolean value, for example:

```
bool found_valid = false;
for (unsigned i = 0, e = vector.size(); i != e; **i)
if (vector[i]->is_valid()) {
found_valid = true;
break;
}

if (found_valid) {
...
}
```

Instead of writing this loop inside a bigger function, extract it to a predicate function (usually static) that also uses early exits:

```
/// Returns true if the input vector contains a valid buffer.
static bool has_valid_buffer(const std::vector<buffer *> &vector) {
for (unsigned i = 0, e = vector.size(); i != e; **i) {
if (vector[i]->is_valid()) {
return true;
}
}
return false;
}
...
if (has_valid_buffer(vector)) {
...
}
```

Writing it this way has many benefits:

1. Reduces indentation.
2. Factors code that can be potentially reused by other places.
3. Forces the programmer to give a name for the function, converting a piece of code into a concept which is easier to understand by the reader (a raw loop vs a function call with name `has_valid_buffer()`).
4. Includes a documentation block for the function.

Please extrapolate this example to more complex cases where the predicate is not as obvious. Instead of being faced with all the inline details of how the predicate is checked, we can trust the function and continue reading with better locality.

Style Aspects: Low Level Issues

Naming Conventions

Names should be coded following the snake_case pattern. For variables, functions and classes the name schema will always be lowercase and separate the words by an underscore. Exceptions are SI units and other common units with uppercase letters (dB, dBm) which should be kept in their original form. Names should be descriptive and easy to read and avoid abbreviations unless they are well known.

```
int n_pdu          // Not so good, it's difficult to imagine what represents.
int number_of_pdu  // Good, descriptive name.
int pdu_count      // Good, descriptive name.
int path_loss_dB   // Good, descriptive name with unit suffix.
```

Non-descriptive names should be avoided except for iterators in `for` loops or variables within a very small scope.

```
for (unsigned i = 0, e = size; i != e; ++i) { // i and e are iterators inside a for loop.
std::lock_guard<std::mutex> lck(mutex);     // Good as the scope is only 2 lines long.
detach(i);
}
```

Names should follow these rules (always use snake_case):

> - **Type names** (classes, structs, enums, typedefs, …) should be nouns.
> - **Variable names** should be nouns (they represent state).
> - **Function names** should be verb phrases (they represent actions) and command-like functions should be imperative.
> - **Enum declarations** are types, so they should follow the naming conventions for types.
> - **Enumerators** (eg: `enum { red, green }`) should follow the naming conventions for types. Enumerators that are convenience constants should be written in uppercase. For instance:

```
enum {
MAX_ELEMENTS = 32,
BYTES_PER_ELEMENT = 64
};
```

Assert and expect

OCUDU provides a custom assert macro called `ocudu_assert`. Use it as much as you can to check all your preconditions and assumptions. This will help to reduce debugging times as the assert may be triggered by your code or even by external faulty code.

Make sure to write a descriptive error message in the assert statement, which will be printed when the assertion is triggered. For example:

```
char *get(unsigned idx) {
ocudu_assert(idx < v.size() && "get() out of range!");
return v[idx];
}
```

Other additional examples:

```
ocudu_assert(buffer->is_valid() && "Buffer should always be valid!");
ocudu_assert((channel == DLSCH || channel == ULSCH) && "Channel type is invalid!");
ocudu_assert(idx < get_num_ues() && "UE index value is out of range!");
ocudu_assert(v1.size() == v2.size() && "vector sizes must be identical!");
```

> **INFO**
>
> If an error condition can be triggered by user input then do not use an assert, instead, use a recoverable error mechanism.

Another nice side effect of using assertions is that you can apply the "design by contract" approach for many interfaces, helping to reduce a lot of error checks and error handling clutter. As software is usually layered, you may validate user inputs in a certain layer, keeping it centralized in a single place and then let inner layer interfaces be designed by contract by using assertions. Using this approach will benefit the appearance of simpler interfaces, reduces corner cases and the implementation will have less states to validate which will reduce bugs in untested corner cases.

When using assertions you may get warnings for *"unused value"* if assertions are disabled (mainly in release builds). For example:

```
unsigned size = v.size();
ocudu_assert(size > 42 && "Vector smaller than it should be");

bool is_value_new = set.insert(x);
ocudu_assert(is_value_new && "The value shouldn't be in the set yet");
```

In the first case, the call to `v.size()` is only useful in the assert, and we don't want it executed if assertions are disabled. In this case the code should be moved inside the assertion. In the second case,

the side effects of the call must happen whether the assert is enabled or not. In this case, the value should be cast to void to disable the warning.

```
ocudu_assert(v.size() > 42 && "Vector smaller than it should be");

bool is_value_new = set.insert(x);
(void)is_new_value;
ocudu_assert(is_value_new && "The value shouldn't be in the set yet");
```

Do not use `using namespace std`

When you need to refer to identifiers in the standard library then prefer to explicitly use a `std::` prefix rather than relying in `using namespace std;`. In header files, adding a using namespace directive will pollute the namespace of any source file that includes this header, causing maintenance issues.

The exception to this rule (not for the `std` namespace) is for implementation files (`.cpp`). For example, all the code in OCUDU implements code that lives in the ocudu namespace. In this case, it is clearer for the .cpp files to have a using namespace ocudu directive at the top of the file, just after the include list. This will reduce indentation in the body of the file.

The general form of this rule is that any .cpp file that implement code in any namespace may use that namespace, including its parents, but should not use any others.

Using Range for Loops

Since the introduction of range-based for loops in C++11, it is rarely necessary to do any kind of explicit manipulation of iterators. Try to use range-based for loops wherever possible, for example:

```
for (const auto &ue : ue_db)
... use ue ...
```

Loop Structure

In cases where a range-based for loop cannot be used and it is necessary to write an explicit iterator-based loop, pay attention to the whether end() is re-evaluated on each loop iteration. The most common mistake is writing a loop this way:

```
for (auto i = x.begin(); i != x.end(); ++i)
... use i ...
```

The problem is that it evaluates `x.end()` on each iteration. Instead, use loops that evaluate it once before the loops starts. This can be done using this form:

```
for (auto i = x.begin(), e = x.end(); i != e; ++i)
... use i ...
```

These two loops have different semantics: if the container is being mutated inside the loop, `x.end()` may change its value every time through the loop, so the second form may not be correct. If you actually depend on this behavior, please write the loop in the first form and add a comment indicating you did it intentionally.

By consistently using the second form, the reader will implicitly see that the loop is not mutating the container without needing to analyse the loop body, making easier to read and understand what the code does.

In general, the C++ syntax for iterator comparison in loops is to use the `!=` operator instead of `<`. This should be used consistently, even if the iterator is a plain integer.

For example, the preferred form looks like this:

```
for (unsigned i = 0, e = x.size(); i != e; ++i)
... use i ...
```

Instead of the C-style way:

```
for (unsigned i = 0, e = x.size(); i < e; ++i)
... use i ...
```

Using Pre-increment

In C++ (does not apply to C code), pre-increment (`++x`) may be no slower than post-increment (`x++`) and could be a lot faster than it. As a result, it is preferential to use pre-increment whenever possible.

Use of Anonymous Namespaces

Use an anonymous namespace for making the contents surrounding it private to the file, just like `static` is used for C functions and global variables.

The problem with anonymous namespaces is that they encourage indentation of their body, and they reduce locality of reference: if you see a random function definition in a C++ file, it is easy to see if it is marked as static, but seeing if it is inside an anonymous namespace may require scanning a big chunk of the file.

For these reasons, follow this simple rule: make anonymous namespaces as small as possible and only use them for class declarations, **not** for functions.

```
namespace {
class foo { // foo class has internal linkage
...
public:
int bar(); // define this method outside the anonymous namespace.
...
};
} // end anonymous namespace

static void some_helper() { // static function marked as static outside anonymous namespace.
...
}

int foo::bar() { // method definition outside of anonymous namespace to keep namespace as
compact as possible.
return 42;
}
```

Using C++ Casts

Avoid using C-style casts in C++ code.

C++ casts are more explicit to read in code than the C counterpart. Another reason is that C++ has four different types of casts, allowing the programmer to choose the right tool depending on the circumstances. While the C cast may be used for any case and potentially disabling any warnings from the compiler.

Recommendations

Function and Class Length

Avoid writing classes or functions that are too large.

As a rule of thumb, functions should fit in a window and no scrolling should be needed to review them. Try to write functions that only do one thing or action. If you write a function that does more than one thing, try to split it into multiple functions, each of which do a single thing. This will dramatically improve code maintainability. By using concise function names, and by having each function well documented, code will be much easier to understand.

For classes, it is harder to set a maximum number of lines, so instead, give a class a single responsibility. Although the concept of responsibility may sound abstract, another way of thinking about it is that a class should only have "one reason to change". If a class does multiple things or responsibilities, it is better to split it into multiple classes. This way we may avoid falling into the *god class* anti-pattern which can be a maintenance nightmare.

Scope

Define variables where they are going to be used, making their scope as short as possible. This applies to both C++ and C code.

For example:

```
int i;
float f;
...
... other unrelated code ...
...
f = get_float();
a = f * get_multiplier();
```

Should be transformed to:

```
...
... other unrelated code ...
...
float f = get_float();
int i = f * get_multiplier();
```

Logical Operators

The codebase uses the standard form of logical operators:

- `&&`, `||` and `!`

As opposed to the alternative form:

- `and`, `or` and `not`

The following code block shows the correct and incorrect use of logical operators:

```
if (!a && (b == 8))     // Correct, only used the standard form.

if (not a && (b == 8))  // Incorrect, used 'not' instead of the standard form '!'.
```

Using References(&) Over Pointers(*)

In the general, it is better to use references instead of pointers. C++ references have a simpler syntax than pointers and always refer to a valid object (unless you do some undefined behavior).

Pointers are very useful when the pointed object may optionally be *nullptr*, however managing this case may require adding additional checks to manage this situation.

```
static bool check_objects_are_equal(foo *left, foo *right);              // Bad, left or
right may be nullptr, prefer references.
static bool check_objects_are_equal(foo left, foo right);                // Bad, copies
objects.
static bool check_objects_are_equal(foo &left, foo &right);              // Bad, missing
const, checking the objects should not modify them.
static bool check_objects_are_equal(const foo &left, const foo &right);  // Good.
static void fill_optional_object(foo *x);                                // Good, x is
optional, function checks for nullptr.
```

Const Correctness

Write code that is `const` correct.

```
const object& get_object() const; // Return value cannot be modified, method does not modify
the contents of the class.
```

Avoid Complex Expressions

Try to use local variables to store intermediate results or to cache the result of a function call when calling it multiple times and it is known not to change. This will help to avoid complex expressions.

For example:

```
for (unsigned i = 0; i != get_bar_count(); ++i) {
if (get_bar_count() < i) {
do_something();
}
do_other();
}
```

Can be transformed to this code, saving many calls to `get_bar_count()`

```
unsigned bar_count = get_bar_count();
for (unsigned i = 0; i != bar_count; ++i) {
if (bar_count < i) {
do_something();
}
do_other();
}
```

There may be cases where you *need* to work with complex and deep nested structures. In these cases try to store intermediate members as pointers or references (if writing C++ code) and remember to use `const` if only reading the values. This will avoid needless typing, having more compact code and potentially improving performance for unneeded indirection chasing pointers.

For example:

```
for (uint32_t j = 0, je = metrics_tmp.size(); j != je; ++j) {
metrics[j].dl.n_samples += metrics_tmp[j].dl.n_samples;
metrics[j].dl.mcs += metrics_tmp[j].dl.n_samples * metrics_tmp[j].dl.mcs;
metrics[j].ul.n_samples += metrics_tmp[j].ul.n_samples;
}
```

Can be transformed into:

```
for (uint32_t j = 0, je = metrics_tmp.size(); j != je; ++j) {
downlink_metrics &dl = metrics[j].dl;  // store non const reference to access the struct
const downlink_metrics &dl_tmp = metrics_tmp[j].dl;  // const ref to access the struct

dl.n_samples += dl_tmp.n_samples;
dl.mcs += dl_tmp.n_samples * dl_tmp.mcs;

const uplink_metrics &ul_tmp = metrics_tmp[j].dl;
```

```
ul.n_samples += ul_tmp.n_samples;
}
```

Magic numbers

Avoid the use of magic numbers. Instead, use a variable (`static`, `constexpr`, etc.): this way, you will be forced to give it a descriptive name, improving code readability.

```
if (b < 2.0)  // Bad
...

static constexpr float detection_threshold_dB = 2.0;
if (b < detection_threshold_dB)  // Good - we now know this is a detection threshold value
in dBs
...
```

Fixed Width Integer Types

Use the fixed width integer types (`uint32_t`, etc…) found in the `cstdint` and `stdint.h` header files when you need to ensure that a variable needs to be of fixed size, independent of the target architecture.

Usually you will need to use these types when defining message structures, interfacing hardware, network programming, etc… However, **avoid** using them when the type width is not important because the use of a fixed size type is a declaration done by the programmer that the affected code needs special treatment and careful handling.

Notice that all 32-bit architectures and above define `int` and `unsigned int` to be of 32-bits, so it is safe to use the generic types safely unless you know you will need a wider type.

```
for (uint32_t cc_idx = 0; cc_idx != num_cc; ++cc_idx)
// here we know we're never going to exceed 4 billion cc's - safe to use a generic unsigned
int (32 bits)

for (uint32_t i = 0, e = vector.size(); i != e; ++i)
// how many elements will the vector hold?
// a) if we know it is going to hold less than 4 billion, use unsigned int (32 bits)
// b) if in doubt, use size_t.
// but as you can see there is no need to use a fixed width type, generic types can be
safely used here.
```

Avoid using the `long` type since it can be either 32 or 64-bits depending on the architecture. See the standard definition for more information.

Function signatures

When declaring a function, the list of arguments shall be ordered as follows:

1. Arguments passed by reference or pointer, if any, that are only used to store the function output. These arguments are documented by the Doxygen command `\param[out]`.
2. Arguments passed by reference or pointer, if any, that contain inputs to the function and will also be modified by the function itself. These arguments are documented by the Doxygen command `\param[in,out]`.
3. Arguments passed by value, constant pointer or constant reference, if any, that represent inputs to the function. These arguments are documented by the Doxygen command `\param[in]`. In particular, structures gathering a number of configuration parameters, shall appear last in the argument list.

For example:

```
/// \brief Does something.
///
/// This function writes a span, reads and modifies a vector, only reads the rest of
arguments.
/// \param[out]    fnct_out       Main function output (recall that \c span behaves like a
pointer).
/// \param[in,out] my_vector      A vector of integers the function may read and modify.
/// \param[in]     another_vector A vector of integers the function may only read.
/// \param[in]     quantity       Another value the function needs to know about.
/// \param[in]     cfg_struct     A structure describing the context the function works in.
void my_function(span<int> fnct_out, std::vector<int>& my_vector, const std::vector<int>&
another_vector, int quantity, const config_t& cfg_struct);
```

Class Layout Example

This section describes the recommended layout of a class declaration, recommendations found inline.

```
/// Class description block.
class my_class : public interface_1 {
// Define private constant configurable parameters first - easy to find by class maintainers
to be always in the
// same place.
/// Constant 1 explanation.
static constexpr unsigned some_constant_1 = 2;
/// Constant 2 explanation.
static constexpr unsigned some_constant_2 = 7;

public:
// Declare/define special members first (constructors, destructor, etc...)
// No need to document special members unless something not trivial requires an explanation.
my_class() { ... }  // If the ctor body is big, move it down to the cpp file.
~my_class() { ... }  // Likewise, if the dtor body is big, move it down to the cpp file.

/// Method 1 explanation block.
void method1();
// Separate method declarations with a single newline.
```

```
/// Method 2 explanation block.
void method2();

// Try to group methods of the same category, accessors, modifiers, utilities...

/// Getter explanation block.
something& get_something();
const something& get_something() const; // Feel free to provide two identical explanation
blocks or declare
// both methods in a row, no need of newline here.

/// Getter explanation block.
another_thing& get_anotherthing();  // Here we separated the const and non const method
declarations.
// Please be consistent per class basis.

/// Getter explanation block.
const another_thing& get_anotherthing() const;

// No need to add documentation for overridden interface methods, as they are already
documented in the interface,
// we don't want to keep two copies of documentation blocks since they can easily diverge
creating maintenance issues.
void interface_method1() override;
// Remember to add a newline between methods.
void interface_method2() override;

/// Enum description block.
// Define the enum near the site where it is going to be used to improve locality when
reading the interface.
enum class my_enum {
enumerator1,
enumerator2
};

/// Method using above enum description block.
void method_uses_enum(my_enum e); // my_enum is declared above - easy to find.

protected:  // After the public method section is done, declare the rest if applicable
(protected, private).
// The audience here has changed to class developers, not public interface users anymore, we
avoid polluting the public
// interface with implementation details
/// Protected method explanation block.
void prot_method();

private:  // Private method section.
// This will be only read by users who need to understand implementation details. Keep away
of public interface.
/// Private method explanation block.
void private_method();

protected: // Declare member variables at the end. Keep them separated from methods.
/// Protected member variable.
unsigned protected_member_var;

private: // Last, declare private member variables.
// Variable names should be descriptive, so most of the times there is no need to document
them.
unsigned memb_var_1;
```

```
void* memb_var_2;
/// Protects access to memb_var_2.
// In the case of a mutex it helps to add a short explanation of what is protecting...
mutable std::mutex m;
};
```

Self Generating Documentation

The OCUDU repository uses **Doxygen** to generate API documentation directly from the annotated C++ source files. In order to contribute to the API documentation with an homogeneous style, the following sections provide some general guidelines focusing on the most common code elements. Please refer to the Doxygen Documentation for a complete overview of the Doxygen commands and features.

General Aspects

The documentation must be written in English, with American English spelling as given by the main entry of The Merriam-Webster Dictionary. For all editorial matters (e.g., acronyms, plurals, capitalization, equations), the suggested reference is the IEEE Editorial Style Manual for Authors , especially *Section E* thereof.

All code entities are documented with a comment block just before the declaration of the entity. Placing the documentation block on the same line as the code element is not allowed, even for short, one-line comments. As mentioned here, all lines of a documentation comment block start with a triple slash. Also, OCUDU documentation prefers the `\command` form of Doxygen commands (as opposed to `@command`).

Generally, a documentation block consists of a brief description (ideally, not more than one line) that starts with the command `\brief` and one or more paragraphs separated by empty lines. When no detailed description is provided, the `\brief` command in the brief description can be omitted. The brief descriptions should go directly to the point, without expressions like “The `myclass` class provides/ specifies…” (more on this below).

```
/// \brief Brief description of the code element.
///
/// Detailed description starts here. This can span several lines, randomly
/// filled in this example. Lorem ipsum dolor sit amet consectetur adipiscing
/// elit lacinia maecenas, hendrerit luctus libero vivamus elementum feugiat
/// torquent accumsan eleifend, diam orci aptent tincidunt a iaculis sed nisi.
///
/// More detailed description. Paragraphs are separated by empty lines. Curae
/// arcu tempor urna convallis pulvinar conubia rutrum auctor, rhoncus nam
/// faucibus montes velit non molestie, nostra proin metus senectus sem tempus
/// tincidunt.
class example1;

/// Brief and only description of the code element.
class example2;
```

All code entities should be documented when declared. It is not required (actually, it is discouraged) to repeat the documentation block when defining the code entity. In particular, virtual methods are only

documented when declaring the interface/base class and the documentation is not repeated when defining the implementation.

Other general recommendations are as follows

- When referencing technical specifications or similar documents, only reference specific sections and tables, since they do not change between releases, and avoid using page numbers. Consider including extracts of the document if they provide clarification.

```
/// The class implements TS38.211 Section 4.2.1.
```

- Look for examples inside the code about the spelling of acronyms and references. For instance, the recommended forms are TS38.211 (note the absence of space between TS and the number) or DM-RS.

- Whenever possible, use rigorous mathematical notation for formulas, equations and related matters. For instance, $(0, 1)$ denotes the open interval between 0 and 1 (that is, all real numbers x such that $0 < x < 1$), while $[0, 1]$ stands for the closed interval, with both endpoints included. Half-open intervals, e.g., $[0, 1)$ or $(0, 1]$, are also possible. Discrete sets are written in enumerated notation (e.g., ${1, 4, 23}$), with possible ellipsis when the meaning is obvious (e.g., ${1, ..., 10}$ for the first ten natural numbers, or ${1, 1.2, 1.4, ..., 2.4}$ for all numbers between 1 and 2.4 with increments of 0.2).

- Wrap code examples between `\code` and `\endcode` commands.

```
/// \code
class example_class {
int useless_field;
};
/// \endcode
```

- Use the `\c` command or the `<tt>...</tt>` tags for short inline bits of code, e.g. `\c variable_name` or `<tt> variable_name = 5 </tt`.

- Start a paragraph with `\note`, `\warning` or `\remark` if you want to put extra emphasis (with the obvious meaning) on it.

```
/// \warning An exception will be thrown if the input sequences are empty.
```

Files

A description of the contents of a file can be provided just after the the copyright header. This is particularly recommended for source files of tests and applications. For files, the `\brief` command is

always required.

```
/*
*
* Copyright header...
*
*/

/// \file
/// \brief Unit test for LDPC encoder and decoder.
///
/// For all possible base graphs and lifting sizes, the test extracts from a
/// file a small set of messages and corresponding codeblocks. The messages are
/// fed to the encoder, whose output is compared to the codeblocks. Similarly,
/// the codeblocks are fed to the decoder and the resulting messages are
/// compared to the example ones.
```

Classes and Structures

Class and structure documentation should provide enough information about what it represents and how an instantiation of the class should be used. In the brief description, do not write expressions like “The class/structure represents/defines” but provide directly more meaningful information. Use the `\brief` command when the brief description is followed by a detailed one (optional if the documentation is limited to the brief description).

It is a good practice to explicitly describe edge cases, side effects and less evident aspects of the class (see last example).

```
/// LDPC rate dematcher interface.
class ldpc_rate_dematcher;

/// Decoder configuration.
struct configuration;

/// \brief PHY&amp;ndash;FAPI bidirectional adaptor interface.
///
/// This adaptor is a collection of interfaces to translate FAPI messages into
/// their PHY layer counterpart and vice versa.
///
/// \note All implementations of this public interface must hold the ownership
/// of all its internal components.
class phy_fapi_adaptor;
```

Class Methods and Free Functions

Methods and functions correspond to actions. As such, the brief description typically starts with a verb (in the third singular person).

```
/// \brief Finds the smallest prime number greater than \c n.
unsigned prime_greater_than(unsigned n);
```

The free function in the example above is very simple, with an input and an output that are clear at first sight. For more complex cases, prefer providing more information by means of the `\param` and `\return` commands. Argument description should follow the same guidelines as general variables (see next section).

Also, similarly to what explained for class and structures, edge cases and unpredictable behaviors should be pointed out.

```
/// \brief Decodes a codeblock.
///
/// By passing a CRC calculator, the CRC is verified after each iteration allowing,
/// when successful, an early stop of the decoding process.
///
/// \param[out] output Reconstructed message of information bits.
/// \param[in] input Log-likelihood ratios of the codeblock to be decoded.
/// \param[in] crc Pointer to a CRC calculator for early stopping. Set
/// to \c nullptr for disabling early stopping.
/// \param[in] cfg Decoder configuration.
/// \return If the decoding is successful, returns the number of LDPC iterations
/// needed by the decoder. Otherwise, no value is returned.
/// \note A codeblock of all zero-valued log-likelihood ratios will automatically
/// return an empty value (i.e., failed CRC) and set all the output bits to one.
virtual optional<unsigned> decode(bit_buffer& output,
span<const log_likelihood_ratio> input,
crc_calculator* crc,
const configuration& cfg) = 0;
```

Class Data Members, Objects, Variables

Objects and class data members, both variable and constant, static or not, are treated as if they were the concept they represent. As such, their brief description should not show terms like “represents”, “indicates”, “denotes” or similar. Also, there is no need to repeat the type of the object, since Doxygen repeats the declaration of the object together with its description. Although not common, documentation of this type of entities may also require one or more paragraphs of detailed description. Some good documentation examples are reported below.

```
/// Maximum number of iterations.
unsigned max_iterations = 6;

/// New data flag (\c true if first HARQ transmission).
bool new_data = true;

/// \brief LDPC decoding statistics.
///
/// Provides access to LDPC decoding statistics such as the number of decoded
/// codeblocks (via <tt>ldpc_stats->get_nof_observations()</tt>) or the average
```

```
/// number of iterations for correctly decoded codeblocks (via
/// <tt>ldpc_stats->get_mean()</tt>).
sample_statistics<unsigned> ldpc_decoder_stats = {};
```

Commit Formatting

When committing code to OCUDU codebase it is important that commit messages are clear and succinct. This means having clear guidelines for how commits should be structured, both in the title and body of the commit messages.

The preferred commit message style is as follows:

```
component: brief change description
```

Here, component refers to the section of the codebase being modified by the commit. Some examples of this could be:

- `mac_test`, `equalizer` - changes are limited to a specific function/class
- `phy`, `mac`, `rlc`, etc - multiple changes across a specific layer
- `cmake` - changes to the cmake file
- `all` - a commit modifies something across the whole codebase
- `misc` - a miscellaneous change

Multiple components can be used if a commit touches two or more components. Simply separate the components with a comma. For example:

```
rlc, mac: some commit message
```

The brief change description should be just that, *brief*. This means no more than 50 characters, including the component(s). This ensures that the commit message will be displayed properly with no cuts. If you need to create a commit message with more information, the body of the commit message may be used. To do this, separate the heading from the body with a line break in your editor. For example:

```
component: brief commit outline less than 50 char

Body of commit message giving more detail about the commit. This should
only be used when further explanation is needed. This can span multiple
lines if necessary, but keep it as succinct as possible.
```

Note that all words in the subject line are lower case and there is no trailing period.

To summarize, here are a few brief points to follow when committing to the OCUDU codebase:

- Each commit should have a `component` and a `brief outline`

- Commit subject lines should be limited to 50 characters in total
- The body of a commit should be separated by a line break
- Keep all text as succinct as possible
- No upper case letters or trailing period in subject line

If you would like more information on writing “good” commits, this guide is a useful resource.

Logging Style Guide

To ensure consistency in the formatting of the log entries, some rules should be followed by the developers. This allows both users and developers to be able to easily read and parse the logs, accelerating problem discovery.

This document describes the logging rules and reasoning behind them in OCUDU.

Rule 1: Lists of fields only need to be separated by a whitespace " "

```
rnti=0x4602 h_id=0 prb=[3, 21) symb=[0, 14) mod=QPSK rv=0 tbs=437 crc=OK iter=2.0 snr=12.8dB
t=145.5us
```

Rule 2: Maintain consistent order between classes of fields

There are different classes of fields:

- fields that represent a context (e.g. UE ID, bearer ID, carrier ID)
- fields that represent a value (e.g. snr, tbs, etc.)
- fields that represent a one-time event (e.g. "Received InitialContextSetupRequest")

Each logging message should have the following structure:

```
<timestamp> [layer] [log_level] <contextual fields>: <one-time event>. <value fields>
```

This should result in a message that looks like:

```
2023-05-22T15:36:33.070813 [RLC     ] [I] DL ue=5 SRB1: TX PDU. dc=data p=1 si=full sn=0
so=0 pdu_len=11 grant_len=11
```

Rule 3: Order of contextual fields

Contextual fields should always follow the order of lower granularity to highest granularity:

For example:

```
<direction> <ue ids> <bearer ids> proc=\"<procedure>\".
```

This should result in a message that looks like:

```
DL ue=5 SRB1:…

ue=5 proc="UE Context Release":…
```

Rule 4: For one time events, provide a cause whenever it's unclear what it might be

For example:

```
2023-05-22T15:36:33.070813 [MAC     ] [I] ue=5: Unable to forward UL-CCCH message to upper
layers. Cause: task queue is full.
```

Rule 5: ASN.1 Messages should be represented using the respective ASN.1 type name.

For instance the RRC ASN.1 message RRC Release should be formatted using the name "RRCRelease", which corresponds to the ASN.1 type that is going to be stored in the DL-CCCH message. This name is distinct from the camelCase name "rrcRelease" and white space separated name "RRC Release" that appears in Wireshark.

Rule 6: Procedures should be identified by the name specified in the TS clauses.

For instance, the procedure "UE Context Setup" (notice the spaces) has associated ASN.1 messages "UEContextSetupRequest", "UEContextSetupResponse". According to Rule 1, the *ASN.1 messages* should be formatted **without** white spaces, but according to Rule 2, the *procedure* name should be formatted **with** spaces.

Rule 7: Messages with a context that is not formatted are strictly forbidden!

Messages with a context that have **not** been formatted correctly are strictly forbidden.

General Tips

Use prefix loggers or prefix structs with a specified formatter to maintain order/format consistency of contextual fields.

Testing Policy

This section of our knowledge base outlines our comprehensive testing strategy, which adheres to the Shift-left development approach. Unlike the traditional Waterfall model, where testing occurs after coding, our strategy emphasizes testing at the earliest possible stages of development. This proactive approach ensures that the development team considers testing from the project's inception.

Throughout the development and validation stages, we employ various types of tests to ensure the robustness and reliability of our software:

Unit Tests

Unit tests are the foundational layer of our testing strategy. They scrutinize software behavior at a granular level, examining individual components in isolation. We prioritize unit testing because it is faster, more straightforward, and cost-effective compared to other types of tests. Our development team writes unit tests during development. In most cases, new features in the code and their associated unit tests are in the same Pull Request, even in the same commit. In order to merge a Pull Request into the main development branch, all unit tests (old and new) must succeed.

Key aspects of our Unit Testing approach:

- **Focus on requirements and behavior:**
  > Our unit tests primarily assess whether the software meets its specified requirements and behaves correctly, instead of internal design validation. Most of our unit tests are designed as "sociable unit tests," which means they remain agnostic of implementation details and solely depend on interfaces. This approach allows us to refactor code without affecting the tests and minimizes test instability. However, we also maintain some traditional unit tests for legacy components.

- **Testing happy, bad and edge Paths:**
  > Our unit tests include test cases that explore various scenarios, including corner cases, in addition to testing the expected behavior. Besides that, We encourage testing not only the "happy path" but also adverse and error-prone paths (bad paths and death paths).

- **AAA Pattern (Arrange-Act-Assert):**
  > Whenever possible, our unit tests adhere to the AAA pattern for improved readability and maintainability.

- **Multiple Small Tests vs. Large Test Cases:**
  > We prefer multiple small unit tests over large test cases with multiple assertions. This approach enhances test debugging and comprehension.

- **Mandatory Test Additions:**
  > Whenever developers introduce new features or bug fixes, they are required to create corresponding unit tests. We enforce this through code reviews and code coverage.

- **Component Labeling:**
  > We label unit tests by component, making it easy to assess testing coverage for each software component.

We mainly use Google Test for unit and integration tests, although we have some tests written in pure C++ without the help of a testing framework

---

Integration and Component Tests

Integration tests encompass a few sub-components or complete O-RAN items, verifying input-output relationships as black-box tests. These tests play a crucial role in the initial development stages of (sub)components, serving as skeleton tests to validate general scenarios.

End-to-End (E2E) Tests

Given the complexity of our ecosystem, we still need E2E and system tests to evaluate the behavior and integration of our CU/DU solutions with real and simulated UEs and 5G cores. These tests are primarily managed by the testing team and follow two distinct approaches:

- **Tests without Hardware:**
  > Using ZMQ and simulators, we have developed a test suite encompassing basic scenarios such as attaches, reattaches, and various types of traffic. These tests are easily executable and do not require hardware devices, making them suitable for developers to assess the software's behavior after major changes. They run nightly and can be triggered inside Pull Requests.

- **Tests with Hardware:**
  > For real-world scenarios, we employ commercial (COTS) UE devices and/or RUs to trigger complex tests. These tests evaluate behavior, performance, and integration with external components. They run nightly and weekly and provide valuable metrics and Key Performance Indicators (KPIs).

To facilitate these tests, we've developed an in-house E2E testing framework, containerized for deployment flexibility, ensuring testing replicates production environments. Additionally, we utilize testing solutions from VIAVI to validate our product.

While our E2E test suite does not grow with every code change like unit tests, we continually add new tests when significant features are introduced or when exploratory testing identifies noteworthy scenarios to automate.

Exploratory Tests

Exploratory testing holds particular importance in our strategy due to the complexity of the real-world scenarios we aim to support. Whenever an issue or unexpected behavior is identified, we endeavor to create a test at the lowest possible level, ideally as a unit test.

Code Contribution Guide

Welcome! We are glad that you want to contribute to OCUDU. The project accepts contributions via GitLab merge requests. This document outlines the process to help get your contribution accepted.

Reporting a Security Issue

Most of the time, when you find a bug in OCUDU, it should be reported using GitLab issues. However, if you are reporting a *security vulnerability*, please email a report to security@ocudu.org. This will give us a chance to try to fix the issue before it is exploited in the wild.

We welcome vulnerability reports and fixes, but do not accept contributions derived from unauthorized access or data exfiltration.

Outline

- New Contributor Guide
  - Ways to Contribute
  - Issues
  - Proposing an Idea (Tier 1: GitLab issue · Tier 2: OIP)
  - Merge Requests
  - Licensing
  - Use of AI Coding Assistants
  - Labels

As you get started, you are in the best position to give us feedback on areas of our project that we need help with including:

- Problems found during setting up a new developer environment
- Gaps in our Quickstart Guide or documentation
- Bugs in our automation scripts

If anything doesn't make sense, or doesn't work when you run it, please open a bug report and let us know!

Ways to Contribute

We welcome many different types of contributions including:

- New features
- Builds, CI/CD
- Bug fixes
- Documentation
- Issue Triage
- Answering questions (e.g. in GitLab)

Not everything happens through a GitLab merge request. Please come to our meetings or contact us and let's discuss how we can work together.

Come to Meetings

Absolutely everyone is welcome to come to any of our meetings. You never need an invite to join us. In fact, we want you to join us, even if you don't have anything you feel like you want to contribute. Just being there is enough!

You can find out more about our meetings here. You don't have to turn on your video. The first time you come, introducing yourself is more than enough. Over time, we hope that you feel comfortable voicing your opinions, giving feedback on others' ideas, and even sharing your own ideas, and experiences.

Issues

Issues are used as the primary method for tracking anything to do with the project.

Issue Types

There are 5 types of issues (each with their own corresponding label):

- `question/support`: These are support or functionality inquiries that we want to have a record of for future reference. Generally these are questions that are too complex or large to store in the chat (e.g. Slack channel) or have particular interest to the community as a whole. Depending on the discussion, these can turn into `feature` or `bug` issues.
- `proposal`: Used for items that propose a new idea or functionality that require a larger community discussion. This allows for feedback from others in the community before a feature is actually developed. This is not needed for small additions. Final word on whether a feature needs a proposal is up to the core maintainers. All issues that are proposals should both have a label and an issue title of "Proposal: [the rest of the title]." A proposal can become a `feature` and does not require a milestone.
- `feature`: These track specific feature requests and ideas until they are complete. They can evolve from a `proposal` or can be submitted individually depending on the size.
- `bug/bug::ci`: These track bugs with the code/infrastructure.
- `docs`: These track problems with the documentation (i.e. missing or incomplete).

Issue Lifecycle

The issue lifecycle is mainly driven by the core maintainers, but is good information for those contributing to OCUDU. All issue types follow the same general lifecycle. Differences are noted below.

1. **Issue creation**
2. **Triage**
   - The maintainer in charge of triaging will apply the proper labels for the issue. This includes labels for priority, type, and metadata (such as `good_first_issue`). The only issue priority we will be tracking is whether the issue is "critical." If additional levels are needed in the future, we will add them.
   - (If needed) Clean up the title to succinctly and clearly state the issue. Also ensure that proposals are prefaced with "Proposal: [the rest of the title]".
   - Add the issue to the correct milestone. If any questions come up, don't worry about adding the issue to a milestone until the questions are answered.
   - We attempt to do this process at least once per work day.
3. **Discussion**
   - Issues that are labeled `feature` or `proposal` and fall outside the official roadmap may require a formal OCUDU Improvement Proposal (OIP) depending on scope. See Proposing an Idea for when an OIP is needed. Smaller quality-of-life enhancements are exempt.
   - Issues that are labeled as `feature` or `bug` should be connected to the MR that resolves it.
   - Whoever is working on a `feature` or `bug` issue (whether a maintainer or someone from the community), should either assign the issue to themselves or make a comment in the issue saying that they are taking it.
   - `proposal` and `question/support` issues should stay open until resolved or if they have not been active for more than 30 days. This will help keep the issue queue to a manageable size and reduce noise. Should the issue need to stay open, the `keep open` label can be added.
4. **Issue closure**

Find an Issue

We have good first issues for new contributors and help wanted issues suitable for any contributor. Good first issue has extra information to help you make your first contribution.

Sometimes there won’t be any issues with these labels. That’s ok! There is likely still something for you to work on. If you want to contribute but you don’t know where to start or can't find a suitable issue, you can check the here and ask for suggestions.

Once you see an issue that you'd like to work on, please post a comment saying that you want to work on it. Something like "I want to work on this" is fine.

Proposing an Idea

OCUDU uses a two-tier process for proposing new ideas, scaled to the size of the change.

Tier 1: Lightweight proposal (most changes)

Open a GitLab issue in the main ocudu repository with the `proposal` label and start the discussion there. A maintainer will triage it within 5 business days. If the scope is small enough the issue is converted directly to a `feature` and no further process is required.

Tier 2: OCUDU Improvement Proposal (significant changes)

A formal **OCUDU Improvement Proposal (OIP)** is required when a change:

- introduces or modifies a public API or wire protocol
- adds a new major subsystem or component
- has cross-layer or cross-team impact
- involves a breaking change for existing users
- requires a TSC decision before work begins

OIPs live in the dedicated OIPs repository. The process is:

```
1. Discuss informally    →  GitLab issue (label: proposal) in the main repo
2. Write an OIP          →  copy the template, open an MR in community/oips
3. Community review      →  minimum 2-week discussion window on the MR
4. Decision              →  TSC/maintainers accept, reject, or request changes
5. Implementation        →  accepted OIP is linked to the code MR(s)
6. Close out             →  OIP moved to accepted/ or rejected/ with rationale
```

The OIP template and full process details are in the community/oips README.

After your proposal has been approved, follow the developer's guide to get started.

Merge Requests

Like any good open source project, we use Merge Requests (MRs) to track code changes.

When contributing to this repository, please first discuss the change you wish to make via issue, email, or any other method with the TSC before making a change.

Please note we have a code of conduct, please follow it in all your interactions with the project.

How to Contribute a Patch

1. Identify or create the related issue.
2. Fork the desired repo; develop and test your code changes.
3. Submit a merge request, making sure to sign your work and link the related issue.

Coding conventions and standards are explained in the official developer docs.

Lifecycle

1. **MR creation**
   - MRs are usually created to add a feature or fix a particular issue.
   - It is preferred, but not required, to have a MR tied to a specific issue. There can be circumstances where if it is a quick fix then an issue might be overkill. The details provided in the MR description would suffice in this case.
   - MR naming should follow the following scheme "layer/component: title". For example a MR adding MIMO functionality to the PHY layer could be named "phy: add MIMO functionality".
   - We more than welcome MRs that are currently in progress. They are a great way to keep track of important work that is in-flight, but useful for others to see. If a MR is a work in progress, it **must** be prefaced with "Draft: title". Once the MR is ready for review, remove "Draft:" from the title.
2. **Triage**
   - The maintainer in charge of triaging will apply the proper labels for the issue. This should include at least a size label, `bug` or `feature`, and `awaiting_review` once all labels are applied. See the Labels section for full details on the definitions of labels.
   - Add the MR to the correct milestone. This should be the same as the issue the MR closes.
3. **Assigning reviews**
   - Once a review has the `awaiting_review` label, maintainers will review them as schedule permits. The maintainer who takes the issue should self-request a review.
   - MRs from a community member with the label `size/S` or larger requires 2 review approvals from maintainers before it can be merged. Those with `size/XS` are per the judgement of the maintainers. For more detail see the Size Labels section.
4. **Reviewing/Discussion**
   - All reviews will be completed using GitLab review tool.
   - A "Comment" review should be used when there are questions about the code that should be answered, but that don't involve code changes. This type of review does not count as approval.
   - A "Changes Requested" review indicates that changes to the code need to be made before they will be merged.
   - Reviewers should update labels as needed (such as `needs_rebase`)
5. **Address comments by answering questions or changing code**
6. **LGTM (Looks good to me)**
   - Once a Reviewer has completed a review and the code looks ready to merge, an "Approve" review is used to signal to the contributor and to other maintainers that you have reviewed the code and feel that it is ready to be merged.
7. **Merge or close**
   - MRs should stay open until merged or if they have not been active for more than 30 days. This will help keep the MR queue to a manageable size and reduce noise. Should the MR need to stay open (like in the case of a WIP), the `keep_open` label can be added.
   - Before merging a MR, refer to the topic on Size Labels below to determine if the MR requires more than one LGTM to merge.
   - If the owner of the MR is listed in the `OWNERS` file, that user **must** merge their own MRs or explicitly request another OWNER do that for them.

- If the owner of a MR is *not* listed in `OWNERS`, any core maintainer may merge the MR.

GitLab CI

All merge requests must pass CI/CD pipeline checks before they can be merged into OCUDU. The pipeline automatically runs tests, builds, and validation checks to ensure code quality.

After forking the repository, you need to enable CI/CD runners:

1. Go to your fork's **Settings → CI/CD → Runners**
2. Under the **Instance** tab, enable instance runners (free GitLab Shared Runners)
3. If you encounter issues enabling runners or see a banner stating `Identity verification is required in order to run CI jobs`, follow the account verification instructions

Shared runners are convenient, but they are subject to GitLab's compute minute limits and may be insufficient for large or frequent builds. In that case, using your own runner gives you more stable capacity and control over the build environment.

If you want to use your own runner, register it with GitLab (**in your fork**):

1. In **Settings → CI/CD → Runners**, expand the dropdown menu (or kebab menu with three dots) and choose "Show runner installation and registration instruction."
2. Copy the registration token for your fork.
3. On the machine where the runner will run, install GitLab Runner and execute `gitlab-runner register`.
4. When prompted, enter the GitLab instance URL, the registration token, a description, tags (for example, `saas-linux-medium-amd64`), and executor type (usually `docker` is a good choice).
5. After registration, make sure the runner is **active** and available for your fork.

See GitLab Runner install at https://docs.gitlab.com/runner/install/.

For builds that use ThreadSanitizer or other privileged Docker features, the runner may need `security_opt = ["seccomp=unconfined"]` configured under the `[runners.docker]` section in `/etc/gitlab-runner/config.toml`. The default log output limit (`output_limit`) may need to be increased when builds produce large logs.

See GitLab Runner configuration docs for `config.toml` at https://docs.gitlab.com/runner/configuration/advanced-configuration.html.

Licensing

Licensing is important to open source projects. It provides some assurances that the software will continue to be available based under the terms that the author(s) desired. We require that contributors sign off on commits submitted to our project's repositories. The Developer Certificate of Origin (DCO) is a way to certify that you wrote and have the right to contribute the code you are submitting to OCUDU. See also here for some additional reading.

Any contribution requiring a patent license beyond what is already required under relevant 3GPP standards must be disclosed with the contribution. Contributions requiring additional license requirements must be approved by the TSC committee or a designated subcommittee of the TSC prior to acceptance into any OCUDU codebase.

SPDX License Identifiers

All new OCUDU project source code files must have an SPDX License Identifier in the header. SPDX license identifiers provide a standardized, machine-readable way to declare a file's license, making it easy for automated tools to detect, audit, and verify licensing across large codebases. This improves legal clarity, reduces ambiguity, and simplifies compliance for developers, companies, and downstream users. Reference OCUDU SPDX license identifiers are included below. Files incorporated into OCUDU from other existing open source licensed projects will require using an appropriate license identifier based on that project's license.

For files not implementing 3GPP specifications the following file header shall be used:

```
// SPDX-License-Identifier: BSD-3-Clause-Open-MPI
```

For files implementing 3GPP specifications the following file header shall be used:

```
// SPDX-License-Identifier: BSD-3-Clause-Open-MPI
// Portions of this file may implement 3GPP specifications, which may be subject to additional
```

Sign Your Commits

You sign-off by adding the following to your commit messages. Your sign-off must match the git user and email associated with the commit.

```
This is my commit message

Signed-off-by: Your Name <your.name@example.com>
```

Git has a `-s` command line option to do this automatically:

```
git commit -s -m 'This is my commit message'
```

If you forgot to do this and have not yet pushed your changes to the remote repository, you can amend your commit with the sign-off by running:

```
git commit --amend -s
```

We encourage contributors to use their normal, publicly recognized name in the DCO sign-off to ensure clear attribution, traceability, and long-term accountability of contributions. For example something like `Preferred Firstname Lastname <stable email>`. Using a consistent real-world identity helps maintain transparency in the project's history and simplifies compliance, auditing, and collaboration across a broad community.

That said, we understand that security researchers or contributors in sensitive situations may have legitimate reasons not to disclose their full legal name. In such cases, a consistent and professional pseudonym may be used, provided the contributor can validly certify the DCO and maintain a stable contact email.

Please note that the project maintains the right to reject contributions - especially security-related changes - if the source cannot be reasonably trusted or verified. Please also review our requirements for submitting security related fixes here.

Use of AI Coding Assistants

OCUDU follows the same pragmatic stance as the Linux kernel (Kernel AI Coding Assistants): AI coding tools are permitted, but their use must be disclosed and the human contributor remains fully responsible for every line submitted.

Disclosure

If any part of a contribution was written or materially shaped by an AI tool, add an `Assisted-by:` trailer to the commit message:

```
Assisted-by: GitHub Copilot
```

```
Assisted-by: Claude Sonnet (Anthropic)
```

Use one trailer per tool. Place it alongside the other trailers (`Signed-off-by:`, `Co-developed-by:`, etc.) at the bottom of the commit message.

Human responsibility

The `Signed-off-by:` tag certifies the Developer Certificate of Origin (DCO). AI tools cannot sign off - only the human submitting the patch can. By signing off you take full responsibility for the correctness, security, and licensing of the contribution, regardless of how it was produced.

This means:

- Review AI-generated code as carefully as you would review anyone else's patch.
- Do not submit code you do not understand simply because a tool produced it.

- Security-sensitive changes receive extra scrutiny; the origin of the code does not reduce that bar.

What does not need disclosure

Routine autocomplete suggestions, grammar fixes in documentation, or one-line completions do not require an `Assisted-by:` tag. The intent is transparency about substantive AI involvement, not accounting for every keystroke.

Labels

The following tables define the main label types used for OCUDU. They are split up by category.

Common

Label	Description
`bug`	Marks an issue in the main OCUDU codebase as a bug or a MR as a bugfix
`bug::ci`	Marks an issue in the CI/testing code or infrastructure as a bug or a MR as a bugfix
`critical`	Marks an issue or MR as critical. This means that addressing the MR or issue is top priority and must be addressed as soon as possible
`docs`	Indicates the issue or MR is a documentation change
`feature`	Marks the issue as a feature request or a MR as a feature implementation
`keep_open`	Denotes that the issue or MR should be kept open past 30 days of inactivity
`refactor`	Indicates that the issue is a code refactor and is not fixing a bug or adding additional functionality

Issue Specific

Label	Description
`help_wanted`	Marks an issue needs help from the community to solve
`proposal`	Marks an issue as a proposal
`question/support`	Marks an issue as a support request or question
`good_first_issue`	Marks an issue as a good starter issue for someone new to OCUDU
`wont_fix`	Marks an issue as discussed and will not be implemented (or accepted in the case of a proposal)

MR Specific

Label	Description
`awaiting_review`	Indicates a MR has been triaged and is ready for someone to review
`breaking`	Indicates a MR has breaking changes (such as API changes)
`in_progress`	Indicates that a maintainer is looking at the MR, even if no review has been posted yet
`needs_rebase`	Indicates a MR needs to be rebased before it can be merged
`needs_pick`	Indicates a MR needs to be cherry-picked into a feature branch (generally bugfix branches). Once it has been, the `picked` label should be applied and this one removed
`picked`	This MR has been cherry-picked into a feature branch
`docs_needed`	Tracks MRs that introduces a feature/change for which documentation update would be desirable (non-blocking). Once a suitable documentation MR has been created, then this label should be removed

Size labels

Size labels are used to indicate how "dangerous" a MR is. The guidelines below are used to assign the labels, but ultimately this can be changed by the maintainers. For example, even if a MR only makes 30 lines of changes in 1 file, but it changes key functionality, it will likely be labeled as `size/L` because it requires sign off from multiple people. Conversely, a MR that adds a small feature, but requires another

150 lines of tests to cover all cases, could be labeled as `size/S` even though the number of lines is greater than defined below.

Any changes from the community labeled as `size/S` or larger should be thoroughly tested before merging and always requires approval from 2 core maintainers. MRs submitted by a core maintainer, regardless of size, only requires approval from one additional maintainer. This ensures there are at least two maintainers who are aware of any significant MRs introduced to the codebase.

Label	Description
`size/XS`	Denotes a MR that changes 0-9 lines, ignoring generated files. Very little testing may be required depending on the change.
`size/S`	Denotes a MR that changes 10-29 lines, ignoring generated files. Only small amounts of manual testing may be required.
`size/M`	Denotes a MR that changes 30-99 lines, ignoring generated files. Manual validation should be required.
`size/L`	Denotes a MR that changes 100-499 lines, ignoring generated files.
`size/XL`	Denotes a MR that changes 500-999 lines, ignoring generated files.
`size/XXL`	Denotes a MR that changes 1000+ lines, ignoring generated files.

Contents:

Software Architecture Guide

4 items

Codebase Guide

2 items

C++ Code Guide

7 items

Logging Style Guide

Log levels, subsystem tags, and formatting rules for consistent, readable OCUDU log output.

Testing Policy

Test coverage expectations for OCUDU contributions: unit, integration, and system tests across the development lifecycle.

Code Contribution Guide

Ways to contribute, how to propose ideas, opening merge requests, and licensing requirements for OCUDU.

Docs Contribution Guide

How to contribute to the OCUDU documentation: repository structure, adding pages, writing style, and submitting a merge request.

Using Claude Code

Using Claude Code to work on the OCUDU documentation with a pre-configured agent context file.

Reporting Issues

How to report bugs and feature requests via GitLab, and how to disclose security vulnerabilities.

Doxygen

Docs Contribution Guide

This guide is for anyone who wants to improve the OCUDU documentation: adding a new integration guide, writing a tutorial, fixing an error, or expanding existing content. It covers everything you need to go from a local clone to a submitted merge request.

For contributing code to OCUDU, see the Code Contribution Guide.

Repository Structure

The documentation lives in the ocudu_docs repository. The top-level structure is:

```
ocudu_docs/
├── .gitlab/ # GitLab project configuration and templates
├── .reuse/ # REUSE licence compliance configuration
├── LICENSES/ # Licence texts referenced by SPDX headers
├── docs/ # All documentation content as Markdown files
│ ├── user_manual/ # Installation, configuration, running, outputs, troubleshooting
│ ├── tutorials/ # Step-by-step tutorials
│ ├── integrations/ # Third-party hardware and core integration guides
│ ├── knowledge_base/ # Background and reference material
│ └── dev_guide/ # Developer and contributor documentation
├── plugins/ # Custom Docusaurus plugins
├── scripts/ # Build scripts
├── src/ # Custom React components and CSS
│ ├── css/ # Custom stylesheets
│ ├── pages/ # Custom React pages
│ └── theme/ # Docusaurus theme component overrides
├── static/ # Static assets served directly
│ ├── doxygen/ # Doxygen HTML output (mounted from the OCUDU source repo)
│ └── img/ # Images used in the documentation
├── .env # Environment variables for docker-compose
├── .gitignore # Git ignore rules
├── .gitlab-ci.yml # GitLab CI/CD pipeline configuration
├── LICENSE # Project licence
├── README.md # Repository overview
├── REUSE.toml # REUSE/SPDX licence compliance configuration
├── docker-compose.yml # Docker services for the documentation site
├── docusaurus.config.js # Site configuration, including the top navbar
├── package-lock.json # Locked dependency versions (auto-generated)
├── package-lock.json.license # SPDX licence sidecar for package-lock.json
├── package.json # Node.js dependencies
├── package.json.license # SPDX licence sidecar for package.json
└── sidebars_extended.js # Sidebar navigation for all sections
```

The `node_modules/` and `.docusaurus/` directories are generated automatically and do not need to be committed or modified.

Setting Up a Local Preview

Docker must be installed before running the site. See the Docker installation guide if you have not set it up yet.

Clone the ocudu_docs repository:

```
git clone https://gitlab.com/ocudu/ocudu_docs.git
```

If you do not have write access, fork the repository on GitLab first and clone your fork instead. See Submitting Your Contribution for the full workflow.

From the root of the repository, start the preview server:

```
docker compose -f docker-compose.yml up --build
```

The `--build` flag forces Docker to rebuild the container image before starting. Use it the first time you run the site and whenever dependencies change. Without it, Docker reuses a cached image that may be out of date.

The site is available at http://localhost:3001. If port 3001 is already in use, update the port mapping in `docker-compose.yml`.

Content changes to Markdown files are reflected automatically without restarting the server. If you modify any build files (`docusaurus.config.js`, `sidebars_extended.js`, `package.json`, or the `src/` directory), stop the server and run the command again with `--build` to pick up the changes.

How the site works

Docusaurus renders all `.md` files in the repository automatically:

- Every Markdown file in `docs/` is collected and made accessible by URL.
- Files are excluded only if their path matches an exclusion rule in `docusaurus.config.js`.
- `README.md` files become index pages for their directory.

Static pages such as the Doxygen API reference are served directly and accessible at `baseUrl/doxygen/index.html`. Doxygen output is only built during deployment and will not be available in the local development server.

Search is powered by `@easyops-cn/docusaurus-search-local` and indexes all documentation pages locally without any external service. The search index is only built during a production build and will not work in the local development server.

Adding a New Page

1. Create the file

Place the file in the appropriate subdirectory under `docs/`. Use lowercase with underscores for filenames. For a new tutorial, the file goes in `docs/tutorials/<topic>/index.md`.

Use a `.md` extension by default. Use `.mdx` only if the page requires JSX components or React imports. The JSX pitfall in Common Pitfalls explains what breaks if you mix these up.

2. Add frontmatter

Every page requires a frontmatter block at the top:

```
---
description: "A short description of this page. Quote the value if it contains a colon."
displayed_sidebar: userDocsSidebar
---
```

`description` is used for search results and page metadata. If the value contains a colon followed by a space, wrap it in double quotes. Unquoted colons cause a YAML parse error and break the build.

`displayed_sidebar` tells Docusaurus which sidebar to show when this page is open. Use `userDocsSidebar` for user-facing content, `devSidebar` for developer documentation, and `knowledgeBaseSidebar` for knowledge base pages. Without this field, the sidebar may not display correctly on pages that are not in an autogenerated directory.

These are the only required frontmatter fields. Other Docusaurus frontmatter keys (`title`, `slug`, `sidebar_label`, `id`) are supported but optional.

Every page should open with an H1 heading (`# Page Title`). Docusaurus uses this as the visible page title in the browser and table of contents.

3. Register the page in the sidebar

This step is required. A new file that is not registered in `sidebars_extended.js` will not appear in the navigation, even though the page itself is accessible by URL.

Open `sidebars_extended.js` and add an entry in the correct section. For example, to add a new integration guide for a radio unit:

```
items: [
'integrations/radio_units/benetel',
'integrations/radio_units/your_new_guide',  // add this line
'integrations/radio_units/foxconn',
],
```

Use the file path relative to `docs/`, without the `.md` extension. Keep entries in alphabetical order within their section. This is a readability convention, not a build requirement.

Some subdirectories under `docs/` are marked as `autogenerated` in `sidebars_extended.js`. Files added to those directories appear in the sidebar automatically without a manual entry. Most tutorial subdirectories and user manual sections use this mode. Integration guides and developer documentation do not — entries must be added manually.

4. Check the navbar

Most pages do not require a navbar change. The navbar links to section index pages, not individual documents. Only add a navbar entry if you are creating a new top-level section of the site.

Adding a New Integration Guide

Integration guides are the most common community contribution. The process is:

1. Create the file at `docs/integrations/radio_units/<vendor>.md`, `docs/integrations/5g_cores/<vendor>/index.md`, or `docs/integrations/switches_and_timing/<vendor>.md` depending on the type.
2. Follow the structure of an existing guide in the same category. The Benetel guide is a good reference for a radio unit guide.
3. Add the file to `sidebars_extended.js` in alphabetical order within the correct category (see step 3 in Adding a New Page above).
4. Add any images to `static/img/` and include a `.license` sidecar alongside each one (see License sidecars below).

Writing Style

Follow these rules when writing documentation:

- **Active voice.** Write "run the command" not "the command should be run."
- **Imperative for instructions.** Write "clone the repository" not "you should clone the repository."
- **One topic per sentence.** Split complex sentences rather than joining them with "and" or "but."
- **No em-dashes (—).** Use a colon, semicolon, or a new sentence instead.
- **Quote all commands and file paths** using inline code formatting.

For a full style reference, see the Hitchhiker's Guide to Documentation and the Diataxis framework for guidance on which type of document to write.

Common Pitfalls

Check all of the following before submitting a merge request:

- **Missing sidebar entry.** The most common mistake. Every new page must be added to `sidebars_extended.js` manually if its directory is not listed as `autogenerated`.
- **Missing `displayed_sidebar`.** Pages outside an autogenerated directory may show no sidebar at all. Add `displayed_sidebar: userDocsSidebar` (or the appropriate sidebar ID) to the frontmatter of every new page.
- **Unquoted YAML colons.** Any `description:` value that contains `:` must be wrapped in double quotes.
- **Broken relative links.** Links to other `.md` files use relative paths. Verify that the target file exists at the path you specify.
- **Broken anchor links.** An anchor like `#section-heading` will silently stop working if the target heading is renamed. Verify all anchors after any heading change.
- **Missing `.license` sidecar.** Every PNG, JPG, GIF, SVG, PDF, `.woff`, or other binary or generated file in `static/` requires a `.license` file alongside it. See License sidecars.
- **JSX in `.md` files.** JSX syntax (React components, `className=`, self-closing tags) is only valid in `.mdx` files. Using JSX in a `.md` file will break the build. If your page includes custom components, rename the file to `.mdx`.
- **Conflict markers.** Run `git diff` before committing to confirm no `<<<<<<<`, `=======`, or `>>>>>>>` markers remain in the files.

License sidecars

PNG, JPG, GIF, SVG, PDF, `.woff`, and other binary or generated files require a `.license` sidecar. Update the end year to the current year:

```
SPDX-FileCopyrightText: Copyright (C) 2021-<current year> Software Radio Systems Limited
SPDX-License-Identifier: BSD-3-Clause-Open-MPI
```

For example, an image at `static/img/my_diagram.png` requires a file at `static/img/my_diagram.png.license` with the above content.

Submitting Your Contribution

If you have write access to the repository, clone it directly and start from step 2. If you do not have write access, begin with step 1.

1. Fork the ocudu_docs repository on GitLab and clone your fork.
2. Create a branch with a descriptive name using underscores, for example `add_<vendor>_integration_guide` or `fix_<topic>_typos`.
3. Make your changes and verify the local preview looks correct.
4. Verify your Git identity is configured before committing. The DCO sign-off will be malformed without it:

```
git config user.name
git config user.email
```

Set either with `git config --global user.name "Your Name"` if blank.

5. Sign your commits with a Developer Certificate of Origin (DCO) sign-off. The DCO certifies that you wrote the change and have the right to contribute it. Use the area of work as the commit scope:

```
git commit --signoff -m "<area>: describe what you changed"
```

Common scopes: `tutorials`, `integrations`, `user_manual`, `dev_guide`, `knowledge_base`, `config`, `css`.

6. Open a merge request against the `main` branch. Describe what you changed and why.

When you open the MR, two automated checks run: a REUSE licence header check and a full Docusaurus build, which enforces broken link detection. Both must pass before the MR can be merged. The pipeline also produces a live preview URL, which appears in the CI job output and lets you review the rendered site before merge.

If you are unsure whether your contribution fits the documentation or have questions at any point, open an issue or ask in the community channels.

Using Claude Code

Claude Code is an AI coding assistant that reads a `CLAUDE.md` file from the root of a repository to understand project conventions before starting work. A pre-configured `CLAUDE.md` is available for the `ocudu_docs` repository. Support for the OCUDU source repository is in progress.

Documentation

The `CLAUDE.md` for `ocudu_docs` gives the agent the context it needs to work on the documentation: the repository structure, writing conventions, sidebar registration rules, frontmatter requirements, and common build pitfalls.

Download the file and save it as `CLAUDE.md` in the root of your cloned `ocudu_docs` repository. The file is served from `static/CLAUDE.md` in the repository:

Download CLAUDE.md

Open Claude Code from the root of the repository. It will read the file automatically. Some tasks you can ask it to handle:

- Add a new integration guide for a vendor, including the file, frontmatter, and sidebar entry
- Check a page for style issues against the rules in the Docs Contribution Guide
- Run through the common pitfalls checklist before submitting a merge request

Reporting Issues

Issues

For contributing code and reporting issues, use GitLab's merge request or issue tracking system.

Vulnerabilities

For issues or fixes that could potentially affect software security, please use the dedicated `security@ocudu.org` email for submissions. Our target initial response time for any vulnerability report is less than 7 days.