Overview

This page gives a high-level overview of the shared libraries described so far, what each one is for, how they fit together, and how they are meant to be used in the codebase.

> The goal is not to replace the detailed documentation for each module.

These libraries are all small, but they are not random utilities. Together they form a set of shared infrastructure for building embedded application code that is:

- more consistent
- easier to reason about
- easier to extend
- less likely to devolve into every subsystem inventing its own incompatible habits

Which, naturally, is what happens the moment shared infrastructure is missing.

Design philosophy of the shared library layer

Before going into the individual libraries, it helps to understand the common design pattern behind them.

> These libraries are not trying to be a giant framework. They are trying to provide **targeted, reusable building blocks** for recurring embedded problems:

- reporting success and failure consistently
- logging runtime behavior in a uniform way
- storing shared variable-sized state in a controlled memory region
- scheduling work by discrete priority
- decoding and dispatching incoming protocol messages to the correct subsystem

The philosophy behind them is mostly this:

Centralize recurring patterns

If every module invents its own result codes, logging style, queueing scheme, and packet dispatching logic, the system becomes harder to maintain very quickly.

These libraries centralize those patterns so the rest of the application can focus on subsystem logic instead of re-solving the same infrastructure problems over and over.

Keep APIs small and practical

The libraries generally expose narrow APIs with very specific purposes.

Prefer explicit ownership and caller-provided resources

Several of these modules rely on the caller to provide memory, configuration, or queue storage.

> That is not accidental. It keeps ownership visible and lets the application control where resources live.

Separate policy from mechanism where useful

A few libraries expose a generic interface while allowing board-specific or implementation-specific backends.

Examples:

- the logging API is conceptually transport-agnostic even though the current implementation uses UART
- the packet dispatcher API is conceptually about routing decoded packets even though the current implementation uses FreeRTOS queues and tasks
- the KV pool lets callers decide where metadata and storage memory live

Be honest about constraints

These are embedded libraries and we are not that good at coding.

A lot of their usefulness depends on respecting their assumptions:

- some are single-consumer by design
- some are not ISR-safe
- some assume static lifetime of configuration objects
- some have concurrency limitations that matter a lot

>

> That is why documentation matters here. These modules are only “simple” if you already know their rules.

How the libraries fit together

At a system level, the libraries can be thought of as falling into a few categories.

Core utility infrastructure

These are foundational and broadly reusable:

- `result`
- `logging`

> They define how modules report status and how the system reports runtime information.

Storage and local scheduling infrastructure

These are reusable building blocks for internal system behavior:

- `bucketed_pqueue`
- `kv_pool`

They solve internal resource management and work scheduling problems.

Communication and protocol infrastructure

These are more application-flow oriented:

- packet dispatcher / decoding task
- packet dispatcher macros

> They take incoming protocol messages and move them to the right processing logic.

A good mental model is:

- `result` defines how functions communicate success and failure
- `logging` defines how the system communicates information outward
- `bucketed_pqueue` defines how work can be prioritized internally
- `kv_pool` defines how variable-sized keyed data can be stored safely in managed memory

- the dispatcher layer defines how external messages enter the application and get routed to the correct handlers

---

Revision #1
Created 2026-04-15 08:34:11 UTC by Nikolaos Diamantopoulos
Updated 2026-04-15 08:39:00 UTC by Nikolaos Diamantopoulos