

Embedded Common Libraries

All Common Libraries used by all Microcontrollers in the Rover

- [Overview](#)
- [Result Library](#)
- [Logging](#)
- [Priority Queue](#)
- [Key-Value Pool](#)

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

Result Library

Purpose

The `result` module defines a **shared result code system** for the codebase.

Its job is to give functions a consistent way to report success and failure without inventing random local conventions like:

- `0` means success here
- `1` means success there
- negative values somewhere else
- and one cursed module returning `true` for failure because someone was feeling creative

Instead, functions return a `result_t`, which makes error handling:

- consistent
- readable
- easier to propagate upward
- easier to log
- easier to document

This module also provides:

- string conversion helpers for result codes
- helper macros for early-return error propagation
- optional logging-aware propagation macros when the logging module is included

What files belong to this module

This module consists of:

- `result.h`
- `result.c`

`result.h`

Defines:

- the `result_t` enum
- the public string conversion functions
- the helper macros:
 - `TRY`
 - `TRY_CLEAN`
 - `TRY_LOG`
 - `TRY_LOG_CLEAN`

`result.c`

Implements:

- `result_to_short_str()`
- `result_to_desc_str()`

What problem this solves

In an embedded system, functions fail for many reasons:

- invalid arguments
- timeouts
- communication issues
- bad packet formats
- state machine misuse
- lack of memory
- busy resources
- and the usual parade of avoidable pain

Without a shared result type, every module ends up inventing its own error style. This library creates one common language for reporting outcomes across modules.

That gives the codebase several benefits:

- function contracts are clearer
- errors can be passed up the call chain without translation
- logs can use the same human-readable error text
- helper macros reduce repetitive boilerplate

Core design

The design is intentionally simple:

- `RESULT_OK` means success
- every other `result_t` value represents a failure or exceptional condition
- functions return a single enum value
- callers decide whether to:
 - handle the error locally
 - return it upward
 - clean up before returning
 - log it before returning

This makes `result_t` a lightweight, shared error protocol.

Public API overview

The public API consists of:

- `result_t`
- `result_to_short_str()`
- `result_to_desc_str()`
- `TRY(expr)`
- `TRY_CLEAN(expr)`
- `TRY_LOG(expr)`
- `TRY_LOG_CLEAN(expr)`

String conversion functions

The module provides two functions for converting result codes into human-readable text.

These are useful for:

- logs
- diagnostics
- debug output
- CLI or terminal status messages
- test failure reporting

`result_to_short_str()`

```
const char *result_to_short_str(result_t code);
```

Purpose

Returns a short label for a result code.

Examples

- `RESULT_OK` -> `"OK"`
- `RESULT_ERR_TIMEOUT` -> `"Timeout"`
- `RESULT_ERR_INVALID_ARG` -> `"Invalid Argument"`

Default behavior

If the result code is unknown or unsupported, it returns:

```
"Unknown Error"
```

Intended use

This is best for compact output such as:

```
ERROR: Timeout  
ERROR: Invalid Packet  
ERROR: Buffer too small
```

`result_to_desc_str()`

```
const char *result_to_desc_str(result_t code);
```

Purpose

Returns a longer descriptive explanation of a result code.

Examples

- `RESULT_OK` -> `"The operation completed successfully."`
- `RESULT_ERR_TIMEOUT` -> `"An operation failed to complete within the allotted time."`
- `RESULT_ERR_INVALID_ARG` -> `"A provided argument is null, out of range, or otherwise invalid."`

Default behavior

If the code is unknown or unsupported, it returns:

```
"An unknown error code was encountered."
```

Intended use

This is useful when more context is needed, especially in logs:

Invalid Argument: A provided argument is null, out of range, or otherwise invalid.

Important implementation detail: mapping must stay synchronized

The enum in `result.h` and the switch statements in `result.c` must stay synchronized.

At the moment, they are **not fully synchronized**.

Why this matters

This causes:

- misleading logs
- incomplete diagnostics
- confusion for anyone trying to use those result codes

Maintenance rule

Whenever a new `result_t` value is added, both conversion functions must be updated in the same change.

This should be treated as mandatory.

Error propagation macros

The module provides a set of helper macros that reduce repetitive boilerplate when working with `result_t`.

These macros assume the common pattern:

- call a function returning `result_t`
- if it failed, stop current flow
- either return immediately or jump to cleanup

TRY(expr)

```
#define TRY(expr) \
do { \
    result_t _try_status = (expr); \
    if (_try_status != RESULT_OK) { \
        return _try_status; \
    } \
} while (0)
```

Purpose

Evaluates an expression returning `result_t`.

If the result is not `RESULT_OK`, the current function immediately returns that result.

Example

```
result_t motor_start(void) {
    TRY(motor_check_ready());
    TRY(motor_enable_power());
    TRY(motor_configure_pwm());

    return RESULT_OK;
}
```

Expanded behavior

This behaves roughly like:

```
result_t status = motor_check_ready();
if (status != RESULT_OK) {
    return status;
}
```

for each call.

When to use it

Use `TRY()` when:

- the current function also returns `result_t`
- no local cleanup is needed before returning
- you want simple upward propagation

TRY_CLEAN(expr)

```
#define TRY_CLEAN(expr) \  
do { \  
    result_t _try_status = (expr); \  
    if (_try_status != RESULT_OK) { \  
        goto cleanup; \  
    } \  
} while (0)
```

Purpose

Evaluates an expression returning `result_t`.

If the result is not `RESULT_OK`, execution jumps to a `cleanup:` label.

Example

```
result_t process_frame(void) {  
    result_t status = RESULT_OK;  
    void *buffer = NULL;  
  
    buffer = malloc(128);  
    if (buffer == NULL) {  
        return RESULT_ERR_NO_MEM;  
    }  
  
    TRY_CLEAN(step_one());  
    TRY_CLEAN(step_two());  
    TRY_CLEAN(step_three());  
  
    return RESULT_OK;  
  
cleanup:  
    free(buffer);  
    return RESULT_ERR;  
}
```

TRY_LOG(expr)

When `LOGGING_H` is defined, this macro becomes:

```
#define TRY_LOG(expr) \  
do { \  
    result_t _try_status = (expr); \  
    if (_try_status != RESULT_OK) { \  
        LOGE(TAG, "%s: %s", result_to_short_str(_try_status), \  
            result_to_desc_str(_try_status)); \  
        return _try_status; \  
    } \  
} while (0)
```

Purpose

Like `TRY()`, but also emits a log message before returning.

Required assumption

The surrounding scope must define `TAG`, because the macro calls:

```
LOGE(TAG, ...)
```

If `TAG` is not defined, compilation will fail.

Example

```
#define TAG "NET"  
  
result_t net_start(void) {  
    TRY_LOG(net_hw_init());  
    TRY_LOG(net_link_up());  
  
    return RESULT_OK;  
}
```

If `net_link_up()` returns `RESULT_ERR_TIMEOUT`, the log might look like:

```
[ERROR] NET: Timeout: An operation failed to complete within the allotted time.
```

and then the function returns that result.

TRY_LOG_CLEAN(expr)

When `LOGGING_H` is defined, this macro becomes:

```
#define TRY_LOG_CLEAN(expr) \
do { \
    result_t _try_status = (expr); \
    if (_try_status != RESULT_OK) { \
        LOGE(TAG, "%s: %s", result_to_short_str(_try_status), \
            result_to_desc_str(_try_status)); \
        goto cleanup; \
    } \
} while (0)
```

Purpose

Like `TRY_CLEAN()`, but logs before jumping to `cleanup`.

Behavior when logging is not available

The logging-aware macros depend on whether `LOGGING_H` is defined.

This means they behave differently depending on whether the logging header has been included before `result.h`.

That is an important design detail.

When `LOGGING_H` is defined

If the logging header has already been included, `TRY_LOG` and `TRY_LOG_CLEAN` perform logging through `LOGE`.

This couples the macros to the logging module without hard-including it from `result.h`.

That keeps `result.h` lightweight, but also makes behavior depend on include order.

When `LOGGING_H` is not defined

The code falls back to compiler-specific warning behavior.

GCC / Clang

The macros emit a compile-time warning via `_Pragma(...)` and then degrade to:

- `TRY(expr)`
- `TRY_CLEAN(expr)`

MSVC

They emit a compiler message and also degrade to the non-logging versions.

Other compilers

A general `#warning` is emitted and the macros degrade to the non-logging versions.

Practical meaning

If logging is not available, the macros still work for flow control. They just do not log.

Recommended usage guidelines

Prefer specific result codes

Use the most precise `result_t` value that matches the failure.

Prefer:

- `RESULT_ERR_INVALID_ARG`
- `RESULT_ERR_TIMEOUT`
- `RESULT_ERR_NOT_INITIALIZED`

over generic:

- `RESULT_FAIL`

when possible.

Keep `RESULT_FAIL` as fallback only

`RESULT_FAIL` should mean:

“something failed, but no existing specific code fits cleanly.”

It should not become the default.

Use `TRY()` only in functions returning `result_t`

Otherwise the generated `return _try_status;` is wrong.

Use `TRY_CLEAN()` only when a cleanup label exists

And only when you understand whether the error code is preserved.

Define `TAG` before using logging-aware macros

Without it, `TRY_LOG` and `TRY_LOG_CLEAN` are not valid.

Keep conversion functions updated

Whenever a new enum value is added, update:

- `result_to_short_str()`
- `result_to_desc_str()`

in the same commit.

This should be treated as mandatory maintenance.

Suggested mental model

Think of this module as:

“The project-wide language for function outcomes.”

It is not just a list of enum values.

It defines how modules communicate success and failure to each other, and the helper macros define the common patterns for passing those outcomes upward through the call stack.

That makes it foundational infrastructure, even if the code itself is small and visually innocent.

Logging

Purpose

The logging library provides a simple UART-based logging interface for embedded firmware.

Its main job is to let the application print formatted log messages such as:

```
[INFO] MOTOR: Initialization complete  
[WARNING] SENSOR: Value out of range: 8123  
[ERROR] CAN: Failed to transmit frame
```

The module is intentionally small:

- it supports **three log levels**
- it formats messages in a consistent style
- it sends all output over a single UART
- it provides convenience macros for compile-time log filtering

This is a **printf-style logging system**, not a structured logger, ring buffer, or asynchronous trace system.

What problem this solves

Without this module, code would need to:

- manually format strings
- manually select a UART
- manually prepend log level tags
- duplicate formatting logic across the project

This library centralizes that behavior so all logs:

- use the same format
- use the same transport
- can be filtered by log level at compile time
- remain easy to call from application code

Output format

Every log produced by `LOG()` is formatted as:

```
[LEVEL] TAG: message\r\n
```

Example

```
LOG(LOG_INFO, "IMU", "Sensor ready");
```

produces:

```
[INFO] IMU: Sensor ready\r\n
```

Another example:

```
LOG(LOG_ERROR, "ETH", "TX failed with code %d", err);
```

produces something like:

```
[ERROR] ETH: TX failed with code -3\r\n
```

Components

A log line contains:

- opening bracket `[`
- level string such as `INFO`
- closing bracket and space `]`
- tag string
- separator `:`
- user message string
- CRLF line ending `\r\n`

The line ending is Windows/terminal friendly and also common for UART console output.

Public API overview

The public interface consists of:

- LogLevel
 - log_level_to_string()
 - LOG_init()
 - LOG()
 - convenience macros:
 - LOGE
 - LOGW
 - LOGI
-

Log levels

Enum definition

```
typedef enum {  
    LOG_INFO,  
    LOG_WARNING,  
    LOG_ERROR,  
    _LOG_LAST_LEVEL_DONT_EDIT  
} LogLevel;
```

Meaning

The library supports three levels:

- LOG_INFO
- LOG_WARNING
- LOG_ERROR

These are stored as enum values starting at 0.

The final enum value:

```
_LOG_LAST_LEVEL_DONT_EDIT
```

is not an actual log level. It is a sentinel used to:

- count how many log levels exist
- validate the string table size

Why that sentinel exists

The library keeps a parallel array of strings:

```
static const char *LOG_LEVEL_STRINGS[] = {  
    "INFO",  
    "WARNING",  
    "ERROR",  
};
```

The enum and string table must stay aligned.

The sentinel lets the code verify that automatically with `static_assert`.

log_level_to_string()

```
static inline const char *log_level_to_string(LogLevel logLevel);
```

Purpose

Converts a `LogLevel` enum into its corresponding string.

Valid conversions

- `LOG_INFO` -> `"INFO"`
- `LOG_WARNING` -> `"WARNING"`
- `LOG_ERROR` -> `"ERROR"`

If the value is outside the valid range, it returns:

```
"NoLevel"
```

Notes

This function is declared `static inline` in the header, so each translation unit including the header gets its own inline copy.

It also contains a compile-time check:

```
static_assert((sizeof(LOG_LEVEL_STRINGS) / sizeof(LOG_LEVEL_STRINGS[0])) ==  
              _LOG_LAST_LEVEL_DONT_EDIT,  
              "Mismatch in number of log level strings!");
```

This prevents someone from adding or removing enum levels without updating the string table.

That is one of the few parts of this module behaving like it has trust issues, which is correct.

Backend independence of the API

Although the current implementation sends logs over **UART**, the **public API itself is not inherently UART-specific**.

From the perspective of code using the logger, the interface is simply:

- initialize the logging system with `LOG_init(...)`
- emit logs with `LOG(...)`
- or use the convenience macros `LOGI`, `LOGW`, and `LOGE`

Nothing in normal application code needs to know how the log is actually transported.

What this means in practice

The current board-specific implementation uses:

- a UART handle passed into `LOG_init()`
- `HAL_UART_Transmit()` for output
- `_write()` retargeting for stdout

But that is only one possible backend.

A different board or firmware target could keep the same header/API and provide a different implementation, for example:

- USB CDC logging
- SWO / ITM logging
- RTT logging
- CAN or Ethernet debug output
- buffered logging to memory
- semihosting during development

Why this matters

This separation means the API should be understood as a **logical logging interface**, not as a UART contract.

In this codebase, each board can provide its own implementation behind the same header, as long as it preserves the expected external behavior of the API.

That makes the module portable across boards without forcing higher-level application code to care about the physical logging transport, which is one of the few times abstraction is actually doing something useful instead of just breeding paperwork.

Maintenance guidance

If a future board needs a different logging transport, the preferred approach is:

- keep `logging.h` stable if possible
- replace or adapt the implementation file for that board
- preserve the meaning of:
 - `LOG_init()`
 - `LOG()`
 - `LOGI/LOGW/LOGE`

This allows application code to remain unchanged while the backend changes per target.

Initialization

`LOG_init()`

```
void LOG_init(void *arg);
```

Purpose

Initializes the logging system by providing the UART handle that will be used for all later output.

Expected argument

`arg` must point to a valid `UART_HandleTypeDef`.

In practice:

```
LOG_init(&huart2);
```

or whichever UART handle should be used for logging.

What it does internally

`LOG_init()` performs these steps:

1. Casts `args` to `UART_HandleTypeDef`
2. copies the pointed-to UART handle into a private static variable
3. sets an internal `initialized` flag
4. writes a boot banner directly using `_write()`
5. emits an info log saying logging was initialized

Important requirements

`LOG_init()` must be called before any normal logging is expected to work.

If `LOG()` is called before initialization, it silently returns without output.

Main logging function

`LOG()`

```
void LOG(LogLevel level, const char *TAG, const char *log_message, ...);
```

Purpose

Formats and transmits a log line over UART.

Parameters

`level`

The severity level of the message.

Expected values:

- `LOG_INFO`
- `LOG_WARNING`
- `LOG_ERROR`

If the value is invalid, the implementation falls back to:

```
"UNKNOWN"
```

for formatting.

TAG

A short text label identifying the source of the log.

Typical examples:

- "IMU"
- "CAN"
- "DISPATCHER"
- "ETH"

This appears in the formatted output after the log level.

log_message

A printf-style format string.

Examples:

- "Init done"
- "Received packet %u"
- "Voltage too high: %d mV"

Optional variadic arguments used by the format string.

Example usage

```
LOG(LOG_INFO, "MOTOR", "Started with speed %u", speed);  
LOG(LOG_WARNING, "TEMP", "High temperature: %d", temp);  
LOG(LOG_ERROR, "FLASH", "Write failed");
```

Behavior when not initialized

If logging has not been initialized yet, the function returns immediately:

```
if (initialized == 0) {  
    return;  
}
```

No output is produced.

This is deliberate.

Internal formatting process

The implementation builds the final message in two phases.

Phase 1: Build a full format string

It first constructs a format string like:

```
[INFO] MOTOR: Started with speed %u\r\n
```

This is stored in dynamically allocated memory called `format_message`.

Phase 2: Format variadic arguments into final output

It then uses `vsnprintf()` twice:

1. once to calculate the final required length
2. once to write the fully formatted message into another dynamically allocated buffer

That final message is transmitted using:

```
HAL_UART_Transmit(&huart_handler, (uint8_t *)total_message, total_len, HAL_MAX_DELAY);
```

After transmission, both heap allocations are freed.

Retargeted `_write()`

```
int _write(int file, char *ptr, int len);
```

Purpose

This function retargets standard output to the configured UART.

Behavior

If the file descriptor is `1`:

```
if (file == 1)
```

the function transmits the provided buffer over UART using `HAL_UART_Transmit()`.

It then returns `len`.

Why this exists

On many embedded toolchains, overriding `_write()` allows C library output functions such as `printf()` to write to UART.

That means this module is not only a custom logging module. It also partially redirects stdout.

Important note

This implementation only handles file descriptor `1`, which is typically stdout.

It does not distinguish stderr or other descriptors.

Interaction with `LOG()`

`LOG()` does not actually use `printf()` or `_write()` for its main output path. It calls `HAL_UART_Transmit()` directly after formatting its message.

`LOG_init()` does use `_write()` once to print the boot banner.

So `_write()` exists mainly for stdout retargeting and the boot line, not as the core mechanism used by `LOG()` itself.

Convenience macros

The header defines these macros:

- `LOGE`
- `LOGW`
- `LOGI`

These call `LOG()` with a fixed level, but only if that level is enabled by `CONFIG_LOG_LEVEL`.

Default log level configuration

If `CONFIG_LOG_LEVEL` is not defined by the build system, the header sets:

```
#define CONFIG_LOG_LEVEL LOG_INFO
```

This means all log levels are enabled by default.

Macro behavior

LOGE

```
#define LOGE(TAG, format, ...) LOG(LOG_ERROR, TAG, format, ##__VA_ARGS__)
```

Enabled when:

```
(CONFIG_LOG_LEVEL <= LOG_ERROR)
```

Because `LOG_ERROR` is the highest enum value in this setup, this macro is enabled for all current supported configurations.

LOGW

```
#define LOGW(TAG, format, ...) LOG(LOG_WARNING, TAG, format, ##__VA_ARGS__)
```

Enabled when:

```
(CONFIG_LOG_LEVEL <= LOG_WARNING)
```

LOGI

```
#define LOGI(TAG, format, ...) LOG(LOG_INFO, TAG, format, ##__VA_ARGS__)
```

Enabled when:

```
(CONFIG_LOG_LEVEL <= LOG_INFO)
```

Filtering semantics

Since the enum values are ordered:

- `LOG_INFO = 0`
- `LOG_WARNING = 1`
- `LOG_ERROR = 2`

a lower configured value means **more logs enabled**.

Examples

```
CONFIG_LOG_LEVEL = LOG_INFO
```

Enabled:

- info
- warning
- error

```
CONFIG_LOG_LEVEL = LOG_WARNING
```

Enabled:

- warning
- error

Disabled:

- info

```
CONFIG_LOG_LEVEL = LOG_ERROR
```

Enabled:

- error only

Disabled:

- warning
- info

Disabled macro behavior

When disabled, the macro expands to:

```
(void)0
```

So the call is compiled out.

This is compile-time filtering, not runtime filtering.

That matters because disabled log calls impose essentially no runtime cost.

Typical usage pattern

Initialization

At system startup, once the UART peripheral is ready:

```
LOG_init(&huart2);
```

This should happen before any code that expects logging output.

Logging from application code

Use one of the convenience macros in normal code:

```
LOGI("NET", "Ethernet initialized");  
LOGW("ADC", "Reading outside expected range: %u", sample);  
LOGE("FLASH", "Erase failed at sector %u", sector);
```

This is the intended public usage style.

Using `LOG()` directly is also valid when needed.

Tag conventions

The module does not enforce tag format, so the team should adopt a convention.

A good pattern is to use short subsystem names, such as:

- "ETH"
- "CAN"
- "SCHED"
- "MOTOR"

- "UI"
- "SENSOR"

Keep tags short enough for readable UART logs.

Since this logger is plain text over UART, bloated tags just make the output harder to scan.

Priority Queue

Summary

`bucketed_pqueue` is a simple, efficient strict-priority queue for FreeRTOS systems where:

- items fall into a small fixed set of priority levels
- producers may be tasks or ISRs
- a single consumer drains work in priority order

Its design is intentionally lightweight:

- one FreeRTOS queue per priority
- one bitmap to track which priorities are active
- optional task notification for efficient wakeup

Used correctly, it is a clean fit for embedded event dispatch and deferred work handling.

Used incorrectly, mainly with multiple consumers or inconsistent bucket item types, it becomes a fine little trap with excellent timing and poor manners.

Purpose

The `bucketed_pqueue` module implements a **strict-priority queue** on top of standard FreeRTOS queues.

Instead of storing all items in one queue, it uses **multiple FIFO queues**, called **buckets**, where each bucket represents one priority level.

- **Bucket 0** = lowest priority
- **Bucket** `num_buckets - 1` = highest priority

When consuming items, the module always returns an item from the **highest non-empty priority bucket**.

Within the same priority bucket, ordering remains **FIFO**, because each bucket is an ordinary FreeRTOS queue.

This gives the system:

- **strict priority across buckets**
- **FIFO ordering within each bucket**
- support for **task producers**
- support for **ISR producers**
- a lightweight way to wake a consumer task when new work arrives

When to use this module

Use this module when:

- work items have a **small fixed number of priority levels**
- **higher-priority items must always be processed first**
- you want to keep FreeRTOS queue semantics
- producers may run in both **task** and **interrupt** context
- only **one consumer** is responsible for draining the queue

Typical use cases:

- event dispatching where alarms/errors must preempt normal work
- deferred interrupt processing with different urgency levels
- message handling where command classes have discrete priority bands

When to **not** use this module

This module is **not** a general-purpose concurrent priority queue.

Do **not** use it if:

- you need **multiple consumer tasks** calling `pop()` or `peek()` concurrently
- you need **arbitrary numeric priorities** rather than a small fixed bucket range
- you need **blocking pop** behavior built into the module
- you need stable ordering across different priorities based on timestamp or insertion order

The implementation is designed around **multiple producers, single consumer**.

High-Level Design

The queue is made from:

- an array of FreeRTOS queue handles
- a count of how many buckets exist
- a 32-bit bitmap tracking which buckets are currently believed to be non-empty
- an optional task handle to notify when something is pushed

Core idea

Each priority level has its own FreeRTOS queue.

The module maintains a bitmap:

- bit n set = bucket n is believed to contain at least one item
- bit n clear = bucket n is believed to be empty

This bitmap lets the consumer avoid blindly probing every queue all the time.

Example

If there are 4 buckets:

- bucket 0 = low
- bucket 1 = medium
- bucket 2 = high
- bucket 3 = critical

And the bitmap is:

```
non_empty_mask = 0b1010
```

then bucket 1 and bucket 3 are non-empty.

A call to `bucketed_pqueue_pop()` will check from highest to lowest, so it will try:

1. bucket 3
2. bucket 2
3. bucket 1
4. bucket 0

and return the first available item it finds.

Data Structure

```
typedef struct {
    QueueHandle_t *buckets;
    uint8_t num_buckets;
    uint32_t non_empty_mask;
    TaskHandle_t notifier;
} bucketed_pqueue_t;
```

Fields

buckets

Pointer to an array of `QueueHandle_t`.

Each entry is a FreeRTOS queue representing one priority bucket.

This array is **not owned** by the module. The caller must create the queues and ensure the array remains valid for the entire lifetime of the priority queue.

num_buckets

Number of buckets in the `buckets` array.

Valid range: **1 to 32**.

The upper limit exists because `non_empty_mask` is a 32-bit bitmap.

non_empty_mask

Bitmap used as a fast summary of which buckets contain items.

- bit `0` corresponds to bucket `0`
- bit `1` corresponds to bucket `1`
- etc.

This bitmap is updated on push, and cleared when the consumer determines a bucket is empty.

notifier

Optional task to notify when an item is successfully pushed.

If not `NULL`, a push sets the notification bit:

```
1UL << prio
```

in the target task's notification value.

This is useful when the consumer task waits on task notifications instead of polling.

Important behavioral guarantees

This module provides the following behavior:

Strict priority

Higher-priority buckets are always preferred over lower-priority buckets.

If bucket 3 and bucket 1 both contain items, `pop()` will always return from bucket 3 first.

FIFO within one priority

Because each bucket is a FreeRTOS queue, items in the same bucket are processed in insertion order.

Non-blocking pop/peek

`pop()` and `peek()` do not wait. If no item is available, they return `RESULT_ERR_NOT_FOUND`.

Multi-context producers

There are separate APIs for:

- task context: `bucketed_pqueue_push()`
- ISR context: `bucketed_pqueue_push_from_isr()`

Single-consumer design

The implementation assumes only one consumer performs `pop()` and `peek()`.

That is not just a suggestion. It is a design constraint.

Required setup

This module does **not** create FreeRTOS queues itself.

The caller must:

1. create one FreeRTOS queue for each priority level
2. store those queue handles in an array
3. initialize a `bucketed_pqueue_t` using that array

Example setup

```
#define NUM_BUCKETS 4

static QueueHandle_t bucket_handles[NUM_BUCKETS];

bucketed_pqueue_t pq;

void app_init(void) {
    bucket_handles[0] = xQueueCreate(8, sizeof(my_msg_t));
    bucket_handles[1] = xQueueCreate(8, sizeof(my_msg_t));
    bucket_handles[2] = xQueueCreate(8, sizeof(my_msg_t));
    bucket_handles[3] = xQueueCreate(8, sizeof(my_msg_t));

    bucketed_pqueue_init(&pq, bucket_handles, NUM_BUCKETS, consumer_task_handle);
}
```

Strong recommendation

All buckets should use the **same item type or at least size**.

Technically the module does not enforce that. Practically, mixing queue item sizes makes the API awkward and error-prone, because `pop()` and `peek()` write into a single `out` buffer and the caller has no type information at that point.

Usage model

Producer side

A producer decides the priority and pushes into the matching bucket.

Example:

```
my_msg_t msg = { ... };  
bucketed_pqueue_push(&pq, PRIORITY_HIGH, &msg, pdMS_TO_TICKS(10));
```

From an ISR:

```
BaseType_t higher_woken = pdFALSE;  
my_msg_t msg = { ... };  
  
bucketed_pqueue_push_from_isr(&pq, PRIORITY_HIGH, &msg, &higher_woken);  
portYIELD_FROM_ISR(higher_woken);
```

Consumer side

The consumer repeatedly pops the highest-priority item available.

Example:

```
my_msg_t msg;  
  
while (bucketed_pqueue_pop(&pq, &msg) == RESULT_OK) {  
    process_msg(&msg);  
}
```

Because `pop()` is non-blocking, a typical design is:

1. consumer task blocks on a task notification
2. on wakeup, it calls `pop()` in a loop until `RESULT_ERR_NOT_FOUND`

Example pattern:

```
for (;;) {  
    uint32_t notified_bits = 0;  
    xTaskNotifyWait(0, UINT32_MAX, &notified_bits, portMAX_DELAY);  
  
    my_msg_t msg;  
    while (bucketed_pqueue_pop(&pq, &msg) == RESULT_OK) {  
        process_msg(&msg);  
    }  
}
```

This pattern works well with the module's `notifier` mechanism.

Notification behavior

If `notifier` is provided during initialization, every successful push sends a task notification with:

```
1UL << prio
```

using `eSetBits`.

This means:

- pushes do **not overwrite** previous notification bits
- multiple buckets can be represented in the task notification value
- repeated pushes to the same priority keep the same bit set

What the notification means

The notification indicates that at least one push occurred into that bucket.

It does **not** guarantee:

- how many items are in that bucket
- that the bucket is still non-empty by the time the task wakes
- that the bitmap and queues are perfectly synchronized at all times

It is a wakeup hint, not a count.

That is fine. The consumer should drain the queue with repeated `pop()` calls rather than assuming one notification equals one item.

Concurrency model and assumptions

This section matters more than the function list.

Supported access pattern

Supported

- multiple producer tasks
- ISR producers
- one consumer task calling `pop()` and/or `peek()`

Not supported

- multiple concurrent consumers calling `pop()`
- one task peeking while another task pops in a way that assumes strong synchronization guarantees
- external code modifying the underlying bucket queues directly behind this module's back

The module uses a bitmap plus queue operations, but it does **not** implement a full multi-consumer synchronization scheme around dequeue behavior.

Why single-consumer matters

The bitmap is read, scanned, and repaired in steps.

That is acceptable with one consumer, because any race is limited to producer updates and queue state changes, and the consumer can repair stale bits safely.

With multiple consumers, two tasks could:

- observe the same bitmap snapshot
- both decide the same bucket has data
- one drains the queue
- the other sees an empty queue and clears the bit

That alone is survivable, but once multiple consumers are simultaneously probing and repairing, reasoning about ordering and fairness gets messy fast. This module avoids that entire circus by assuming a single consumer.

Critical sections

The bitmap is protected with FreeRTOS critical sections:

- `taskENTER_CRITICAL()` / `taskEXIT_CRITICAL()`
- `taskENTER_CRITICAL_FROM_ISR()` / `taskEXIT_CRITICAL_FROM_ISR()`

These critical sections protect **bitmap access**, not the whole queue operation sequence.

That means queue operations and bitmap updates are not one indivisible transaction.

This is intentional and mostly fine for the intended model, but maintainers should understand that the bitmap is a **best-effort summary**, not a perfect mirror of queue state.

Known implementation characteristics

Priority scan is linear in number of buckets

`pop()` and `peek()` scan from highest to lowest priority:

```
for (int prio = num_buckets - 1; prio >= 0; prio--)
```

So the dequeue cost is $O(\text{num_buckets})$ in the worst case.

Since `num_buckets <= 32`, this is usually acceptable in embedded systems.

If someone later decides to turn this into 128 priorities with a 32-bit bitmap, you might want to consider a better scanning system for all buckets.

That said, the loop is incredibly efficient doing only a bit-wise check to know if the bucket is populated or not.

Bitmap may be temporarily stale

The module can have these transient states:

- queue contains data but bitmap not yet updated
- bitmap says non-empty but queue is empty

The code handles the second case explicitly by clearing stale bits when `xQueueReceive()` or `xQueuePeek()` fails.

The first case is shorter-lived and happens between a successful queue send and the bitmap update, or if initialization starts with pre-filled queues.

This is why the bitmap should be viewed as a hint structure.

No ownership of queue storage

The module does not allocate or destroy the bucket queues.

It only stores the queue handles provided by the caller.

The caller is responsible for:

- creating queues before initialization
- keeping them alive while the priority queue is in use
- ensuring item types and queue lengths are appropriate

No reset/deinit API

There is no deinitialization function.

If needed, the caller must manage queue lifecycle itself.

If a reset feature is ever added, it must consider:

- draining or recreating each bucket
- resetting `non_empty_mask`
- possible interaction with producers still running

Practical example

Scenario

A system has three priorities:

- 0: telemetry
- 1: commands
- 2: emergency actions

All use the same message type:

```
typedef struct {  
    uint8_t type;  
    uint32_t value;  
} app_msg_t;
```

Setup

```
#define APP_NUM_PRIORITIES 3  
  
static QueueHandle_t app_buckets[APP_NUM_PRIORITIES];  
static bucketed_pqueue_t app_pq;  
  
void app_queue_init(TaskHandle_t consumer_task) {  
    app_buckets[0] = xQueueCreate(16, sizeof(app_msg_t));  
    app_buckets[1] = xQueueCreate(16, sizeof(app_msg_t));
```

```
app_buckets[2] = xQueueCreate(8, sizeof(app_msg_t));

bucketed_pqueue_init(&app_pq, app_buckets, APP_NUM_PRIORITIES, consumer_task);
}
```

Producer task

```
void send_command(uint32_t cmd) {
    app_msg_t msg = {
        .type = 1,
        .value = cmd,
    };

    (void)bucketed_pqueue_push(&app_pq, 1, &msg, 0);
}
```

ISR producer

```
void emergency_isr(void) {
    BaseType_t higher_woken = pdFALSE;

    app_msg_t msg = {
        .type = 2,
        .value = 0xDEADU,
    };

    (void)bucketed_pqueue_push_from_isr(&app_pq, 2, &msg, &higher_woken);
    portYIELD_FROM_ISR(higher_woken);
}
```

Consumer task

```
void consumer_task(void *arg) {
    (void)arg;

    for (;;) {
        uint32_t notify_bits;
        xTaskNotifyWait(0, UINT32_MAX, &notify_bits, portMAX_DELAY);
    }
}
```

```
app_msg_t msg;
while (bucketed_pqueue_pop(&app_pq, &msg) == RESULT_OK) {
    handle_message(&msg);
}
}
```

Result

If telemetry, commands, and emergency messages all arrive, the consumer will process:

1. emergency
2. commands
3. telemetry

Within each class, messages remain FIFO.

Error handling

The module uses `result_t`, which is defined elsewhere.

Based on the implementation, these results are used:

RESULT_OK

Operation succeeded.

RESULT_ERR_INVALID_ARG

Returned when the caller passes invalid arguments, such as null pointers or out-of-range priority.

RESULT_ERR_OVERFLOW

Returned by push functions when the selected bucket queue cannot accept the item.

This usually means the bucket queue is full, or the send timed out.

RESULT_ERR_NOT_FOUND

Returned by `pop()` or `peek()` when no item is available.

This is not necessarily an error in the usual sense. It is the normal result for an empty priority queue.

Maintenance notes

If you change the number of buckets beyond 32

You must also change:

- the bitmap type
- all bit shift logic
- input validation
- notification assumptions based on `1UL << prio`

Right now, 32 buckets is a **hard architectural limit**.

If you add blocking pop behavior

Be careful.

A naive implementation that blocks on each bucket in order would break strict priority or become awkward and expensive.

A better approach is usually to keep the existing design:

- producers notify a task
- consumer waits on notification
- consumer drains with non-blocking `pop()`

If you still add a blocking API, document its wakeup semantics very clearly.

If you want multiple consumers

This requires redesign.

You would need to revisit:

- bitmap synchronization
- dequeue race handling

- semantics of `peek()`
- fairness between consumers
- whether the notifier model still makes sense

Do not label it “thread-safe” just because critical sections exist.

If different buckets need different item types

The current API is not a good fit for that.

`pop()` and `peek()` return into one generic `out` buffer with no explicit type metadata.

If you need heterogeneous payloads, safer patterns are:

- use a tagged union message type
- store pointers to separately managed objects
- wrap payloads in a common envelope struct

If queues are pre-filled before init

The bitmap starts at zero during `bucketed_pqueue_init()`.

So pre-filled queues will not be visible until something later sets the relevant bits, or until code is changed to rebuild the bitmap.

If supporting pre-filled queues matters, one possible improvement is for `init()` to inspect each bucket using `uxQueueMessagesWaiting()` and initialize `non_empty_mask` accordingly.

That behavior does **not** exist today.

Key-Value Pool

Purpose

The `kv_pool` module implements a **fixed-key key-value store backed by a custom memory pool**.

It is designed for systems where:

- keys are known as integer indices in a fixed range
- values are variable-sized blobs of bytes
- dynamic allocation from the general heap is undesirable or unavailable
- memory must come from a caller-provided region
- multiple threads may access the store concurrently

The module combines two things:

- a **lookup table** that maps keys to stored values
- an internal **heap allocator** that manages the memory used by those values

The result is a storage system where each key corresponds to one slot, and each valid slot points to a block allocated from the pool's private heap.

This is not a dictionary in the desktop-software sense. Keys are not hashed, compared, or discovered dynamically. A key is just an **index into a preallocated slot table**.

What problem this solves

This module exists to store variable-sized values in a memory-constrained system without relying on the standard heap.

It solves these problems:

- fixed set of logical keys, but variable-sized data per key
- need to provide memory externally
- need to safely read and update values across threads
- need to reclaim storage when a key is removed
- need predictable ownership of all stored data

Instead of calling `malloc()` and `free()` from the general runtime allocator, the module manages a private heap inside caller-provided memory.

That gives the application explicit control over:

- where memory lives
- how large the pool is
- how many keys exist
- whether metadata and data live together or in different memory regions

High-Level Design

The module consists of two main parts.

Lookup table

Each key corresponds to one `kv_slot`.

A slot contains:

- a per-slot lock
- a pointer to the stored data block
- the size of that block
- a validity flag

If a slot is valid, the key currently has stored data.

If a slot is invalid, the key is empty.

Internal heap

Actual data bytes are stored inside a custom heap managed by the module.

This heap:

- lives in caller-provided memory
- uses a free-list allocator
- allocates variable-sized blocks
- coalesces adjacent free blocks when freeing

This means the slot table stores metadata only. The actual value bytes are stored elsewhere in the pool heap.

Memory model

The pool manages two logical memory regions:

- **lookup table memory**
- **data heap memory**

These can be provided in two ways:

Contiguous mode

One big memory block is given to `kv_pool_init()`. The module splits it into:

1. lookup table
2. data heap

Fragmented mode

Separate memory regions are given to `kv_pool_init_fragmented()`. This allows the lookup table and data heap to live in different memory banks.

That can be useful when, for example:

- metadata should live in fast SRAM
- bulk value storage should live in larger but slower RAM

Public API overview

The public API consists of:

- data structures:
 - `kv_slot`
 - `kv_header`
 - `kv_pool`
- macros:
 - `KV_ALIGNMENT`
 - `ALIGN(x)`
 - `MINIMUM_BLOCK_SIZE`
 - `LOOKUP_TABLE_SIZE(x)`
- initialization:
 - `kv_pool_init_fragmented()`
 - `kv_pool_init()`

- key operations:
 - `kv_pool_get()`
 - `kv_pool_write()`
 - `kv_pool_insert()`
 - `kv_pool_remove()`
 - `kv_pool_is_index_valid()`
- allocator functions:
 - `kv_pool_allocate()`
 - `kv_pool_free()`

The last two are currently exposed in the header, although they behave more like internal allocator primitives than ordinary user-facing API.

Data structures

`kv_slot`

```
typedef struct {  
    atomic_flag slot_lock;  
    void *data_ptr;  
    size_t data_size;  
    bool is_valid;  
} kv_slot;
```

Purpose

Represents the metadata for one key.

Fields

`slot_lock`

A spinlock protecting this slot's metadata and associated data access.

Used to synchronize operations on a single key.

`data_ptr`

Pointer to the allocated data block in the internal heap.

The caller must not free or reallocate this pointer directly.

`data_size`

Size in bytes of the stored value.

`is_valid`

Whether this slot currently contains valid data.

If false, the key is considered empty.

Important note

The key itself is not stored in the slot. The key is simply the slot's index in the lookup table.

`kv_header`

```
typedef struct kv_header {
    size_t size;
    union {
        struct kv_header *next_free;
        char data[1];
    } as;
} kv_header;
```

Purpose

Header for blocks in the internal heap.

Role

When a block is free, `as.next_free` links it into the free list.

When a block is allocated, `as.data` is the start of the user-visible payload.

Meaning of `size`

`size` is the total size of the block, including the header and payload region.

This is important for:

- pointer arithmetic
 - block splitting
 - coalescing adjacent free blocks
-

kv_pool

```
typedef struct {  
    void *pool_start;  
    size_t pool_size;  
  
    atomic_flag heap_lock;  
    void (*delay)(void);  
    kv_header *free_list_head;  
  
    size_t max_keys;  
    kv_slot *lookup_table;  
} kv_pool;
```

Purpose

Represents the entire key-value pool.

Fields

`pool_start`

Start address of the data heap region.

`pool_size`

Size of the data heap region in bytes.

`heap_lock`

Spinlock protecting heap allocator operations.

`delay`

Callback invoked while waiting for a lock.

This is used during busy-waiting to avoid a pure tight spin.

`free_list_head`

Head of the free-list allocator.

`max_keys`

Maximum number of keys supported by this pool.

Valid keys are:

```
0 <= key < max_keys
```

`lookup_table`

Pointer to the slot array.

Alignment and size macros

KV_ALIGNMENT

```
#define KV_ALIGNMENT 16
```

All allocations are aligned to this boundary.

This affects:

- payload placement
- allocator block sizes
- pointer arithmetic

If this value changes, allocator behavior changes with it.

ALIGN(x)

```
#define ALIGN(x) (((x) + (KV_ALIGNMENT - 1)) & ~(KV_ALIGNMENT - 1))
```

Rounds a size up to the next `KV_ALIGNMENT` boundary.

Used by the allocator.

MINIMUM_BLOCK_SIZE

```
#define MINIMUM_BLOCK_SIZE sizeof(kv_header) + 1
```

Minimum block size allowed in the heap.

Used to decide whether a free block can be split.

The allocator will not create a leftover free fragment smaller than this.

LOOKUP_TABLE_SIZE(x)

```
#define LOOKUP_TABLE_SIZE(x) (sizeof(kv_slot) * (x))
```

Returns the number of bytes required for a lookup table supporting `x` keys.

Used during initialization and memory size validation.

Initialization APIs

kv_pool_init_fragmented()

```
result_t kv_pool_init_fragmented(void *lookup_table,  
                                size_t lookup_table_size,  
                                size_t max_keys,  
                                void *pool_data,  
                                size_t pool_size,  
                                kv_pool *pool,  
                                void (*delay)(void));
```

Purpose

Initializes a pool from two separate memory regions:

- one for slot metadata
- one for heap storage

Parameters

`lookup_table`

Memory for the `kv_slot` array.

`lookup_table_size`

Size of the lookup table memory region in bytes.

`max_keys`

Maximum number of keys.

`pool_data`

Memory for the heap region.

`pool_size`

Size of the heap region in bytes.

`pool`

Output pool structure to initialize.

`delay`

Function called while waiting for spinlocks.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if required pointers are null or `max_keys == 0`
- `RESULT_ERR_BUFFER_TOO_SMALL` if:
 - heap is too small
 - lookup table region is too small

What it does

1. validates inputs
2. clears both memory regions with `memset`
3. sets up the lookup table
4. sets up the free-list heap as one large free block
5. clears locks
6. marks all slots invalid

Important lifetime rule

The memory regions provided to this function must remain valid for the entire lifetime of the pool.

The module does not copy them.

```
kv_pool_init()
```

```
result_t kv_pool_init(void *data,
                      size_t data_size,
                      size_t max_keys,
                      kv_pool *pool,
                      void (*delay)(void));
```

Purpose

Initializes a pool from one contiguous memory block.

Parameters

`data`

Start of the full memory region.

`data_size`

Total size of the region.

`max_keys`

Number of keys.

`pool`

Output pool structure.

`delay`

Lock wait callback.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_BUFFER_TOO_SMALL` if total memory is insufficient
- any error returned by `kv_pool_init_fragmented()`

What it does

It computes:

- lookup table size = `LOOKUP_TABLE_SIZE(max_keys)`
- heap start = `data + LOOKUP_TABLE_SIZE(max_keys)`
- heap size = remaining bytes

Then it delegates to `kv_pool_init_fragmented()`.

When to use it

Use this function when you want simple setup from one static buffer.

Use `kv_pool_init_fragmented()` when memory placement matters.

Key operations

kv_pool_get()

```
result_t kv_pool_get(kv_pool *pool, int key, void *buffer, size_t *buffer_size);
```

Purpose

Copies the value for a key into a caller-provided buffer.

Parameters

`pool`

Initialized pool.

`key`

Key index to read.

`buffer`

Destination buffer for copied data.

`buffer_size`

Input/output parameter.

- on input: capacity of `buffer`
- on output:
 - actual copied size on success
 - required size if buffer is too small

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if `pool == NULL` or `buffer_size == NULL`
- `RESULT_ERR_NOT_FOUND` if key is out of range or not valid
- `RESULT_ERR_BUFFER_TOO_SMALL` if destination buffer is too small

Behavior

The function:

1. validates inputs
2. checks key bounds
3. locks the slot
4. verifies the slot is valid
5. checks whether the provided buffer is large enough
6. copies the stored data with `memcpy`
7. unlocks the slot

Important note

The function does **not** require `buffer` to be non-null when the buffer is too small path is taken first, but in practice a null `buffer` with a large enough `buffer_size` would lead to invalid `memcpy`. So callers should always provide a valid buffer unless they intentionally use this as a size query pattern and know what they are doing.

The implementation does not explicitly validate `buffer != NULL`. Why? No clue.

`kv_pool_write()`

```
result_t kv_pool_write(kv_pool *pool, int key, void *buffer, size_t buffer_size);
```

Purpose

Overwrites the existing value for a valid key without reallocating memory.

Parameters

`pool`

Initialized pool.

`key`

Key index to overwrite.

`buffer`

Source data to copy from.

`buffer_size`

Size of new data.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if `pool == NULL`, `buffer == NULL`, or size mismatches existing allocation
- `RESULT_ERR_NOT_FOUND` if key is invalid or not currently in use

Behavior

The function:

1. validates arguments
2. checks that the key refers to a valid existing slot
3. locks the slot
4. checks that `buffer_size` exactly matches the stored size
5. copies new bytes over existing allocation
6. unlocks the slot

Important constraint

This function does **not** resize.

The size must exactly match the existing allocation.

If the caller wants to store a different-sized value, they must:

- remove the key
- insert again with the new size

or implement a resize API in the future.

`kv_pool_insert()`

```
result_t kv_pool_insert(kv_pool *pool, int key, void *data, size_t data_size);
```

Purpose

Allocates heap space and stores new data for a key.

Parameters

`pool`

Initialized pool.

`key`

Key index to populate.

`data`

Source bytes to copy into pool storage.

`data_size`

Size of the data to store.

Returns

Actual implementation returns:

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if:
 - `pool == NULL`
 - `data == NULL`
 - key is out of range
- `RESULT_ERR_NO_MEM` if allocation fails

Inserting on an already allocated slot

If `kv_pool_insert()` is called on a slot that is already valid, the old allocation is orphaned and leaked from the pool heap.

So the safe usage rule today is:

- only call `kv_pool_insert()` on an empty key
- remove existing keys first before reinserting

The implementation does not enforce that, but callers must.

Internal behavior

The function:

1. validates inputs
 2. locks the slot
 3. marks the slot invalid and clears metadata
 4. allocates a heap block
 5. copies data into the new block
 6. marks the slot valid
 7. unlocks and returns
-

kv_pool_remove()

```
result_t kv_pool_remove(kv_pool *pool, int key);
```

Purpose

Removes a key and frees its allocated data block.

Parameters

`pool`

Initialized pool.

`key`

Key index to remove.

Returns

Actual implementation returns:

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if `pool == NULL`
- `RESULT_ERR_NOT_FOUND` if key is out of bounds or not valid

Behavior

The function:

1. validates the pool pointer
2. verifies the key is valid with `kv_pool_is_index_valid()`
3. calls `kv_pool_free()` on the stored pointer

kv_pool_is_index_valid()

```
result_t kv_pool_is_index_valid(kv_pool *pool, int key);
```

Purpose

Checks whether a key currently contains valid data.

Parameters

`pool`

Initialized pool.

key

Key index to check.

Returns

- `RESULT_OK` if key is in range and valid
- `RESULT_ERR_INVALID_ARG` if pool is null or key is out of range
- `RESULT_ERR_NOT_FOUND` if slot is currently invalid

Behavior

The function:

1. validates arguments
2. locks the slot
3. checks `is_valid`
4. unlocks and returns a snapshot result

Important concurrency note

This is only a snapshot.

A slot that is valid at the time of the check may become invalid immediately afterward.

So callers must not do:

1. `kv_pool_is_index_valid()`
2. assume later access is now guaranteed safe forever

They must still handle failure from the actual operation.

Allocator APIs

These are declared in the header and can be called externally, but they behave like internal heap primitives.

Using them directly requires understanding the allocator and slot ownership rules.

`kv_pool_allocate()`

```
result_t kv_pool_allocate(kv_pool *pool, size_t size, void **out_ptr);
```

Purpose

Allocates a block from the pool heap.

Parameters

`pool`

Initialized pool.

`size`

Payload size requested.

`out_ptr`

Output pointer for the allocated payload address.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` for null pointers or zero size
- `RESULT_ERR_NO_MEM` if no free block can satisfy the request

Allocation strategy

The allocator uses **first fit** on the free list.

It scans from the head until it finds the first block large enough.

Block sizing

The allocator computes:

- header offset to payload
- total required size including header
- alignment rounding
- minimum block size enforcement

Splitting

If the selected block is larger than needed and the remainder is big enough, it splits the block and leaves the remainder on the free list.

Otherwise it consumes the whole block.

kv_pool_free()

```
result_t kv_pool_free(kv_pool *pool, void *ptr);
```

Purpose

Returns a previously allocated block to the pool heap.

Parameters

`pool`

Initialized pool.

`ptr`

Pointer previously returned by `kv_pool_allocate()` or stored in a slot.

Returns

- `RESULT_OK` on success
- `RESULT_ERR_INVALID_ARG` if arguments are null

Behavior

The function:

1. derives the block header from the payload pointer
2. inserts the block back into the sorted free list
3. coalesces with adjacent free neighbors when possible
4. scans the slot table for a matching `data_ptr`
5. if found, clears that slot's metadata

Important consequence

`kv_pool_free()` does not merely free heap memory. It also tries to invalidate any slot pointing at that memory.

That means allocator state and key-value metadata are coupled.

Safe usage implication

External callers should not casually use `kv_pool_allocate()` and `kv_pool_free()` unless they understand this coupling.

If you free a pointer that a slot still references, the function will clear that slot.

If you allocate memory manually and never attach it to a slot, the allocator still works, but you are now using the pool partly as a raw allocator and partly as a key-value store, which increases maintenance complexity.

Role of `delay()`

The caller supplies the delay function during pool initialization.

This allows board or environment-specific waiting behavior, such as:

- yielding
- sleeping
- short pause loop
- RTOS task delay
- platform-specific backoff

The pool does not define how delay behaves. That is the caller's responsibility.

Example usage

Contiguous initialization

```
static uint8_t kv_memory[2048];
static kv_pool pool;

static void pool_delay(void) {
    /* platform-specific wait/yield */
}

result_t app_kv_init(void) {
    return kv_pool_init(kv_memory, sizeof(kv_memory), 16, &pool, pool_delay);
}
```

This creates a pool with:

- 16 keys
- lookup table at the start of `kv_memory`
- heap in the remaining bytes

Insert value

```
uint32_t value = 1234;
result_t res = kv_pool_insert(&pool, 3, &value, sizeof(value));
```

This stores 4 bytes at key 3.

Read value

```
uint32_t value = 0;
size_t size = sizeof(value);

result_t res = kv_pool_get(&pool, 3, &value, &size);
```

On success:

- `res == RESULT_OK`
- `value` contains the stored bytes
- `size == sizeof(uint32_t)`

Handle too-small buffer

```
uint8_t small_buf[4];
size_t size = sizeof(small_buf);

result_t res = kv_pool_get(&pool, key, small_buf, &size);
if (res == RESULT_ERR_BUFFER_TOO_SMALL) {
    /* size now contains required size */
}
```

This is the intended size negotiation pattern.

Overwrite same-sized value

```
uint32_t new_value = 5678;
result_t res = kv_pool_write(&pool, 3, &new_value, sizeof(new_value));
```

This succeeds only if key `3` already exists and has exactly 4 bytes allocated.

Remove key

```
result_t res = kv_pool_remove(&pool, 3);
```

This frees the associated heap block and invalidates the slot.

Backend and platform independence

The API is largely independent of where memory comes from and how waiting is implemented.

The caller provides:

- raw memory regions
- a delay function
- the `kv_pool` object itself

That means different boards or environments can use the same API with different backing strategies, for example:

- contiguous static RAM region on one board
- split metadata/data regions across different RAM banks on another
- RTOS yield in the delay callback on one target
- busy-wait pause or test hook in host-side simulation

The implementation is therefore **memory-placement agnostic** and **wait-strategy agnostic**, even though the current source file provides one specific allocator and lock implementation.

This is useful for portability, provided each target respects the same concurrency and memory lifetime contract.

Recommended usage rules for current code

Given the implementation as it exists today, these rules are the safest:

1. Initialize once before concurrent use.
2. Provide memory that remains valid for the full pool lifetime.
3. Use `insert()` only on empty keys.
4. Use `write()` only when new data size exactly matches old size.
5. Do not rely on `is_index_valid()` as a guarantee for later access.
6. Treat direct allocator calls as advanced/internal use.
7. Do not assume allocator concurrency is fully correct (there is definitely a bug or two in there).
8. Always pass a valid destination buffer to `get()` when copying data.