Functions of the Packet Dispatcher

Public API

The following functions are available for the boards to use **outside of** the library.

The public API consists of: `packet_handler_t`, `packet_handler_config_t`, `PacketDispatcherInit(...)`, `DispatchPacket()`
There are also stack depth macros: `PACKET_HANDLER_TASK_STACK_DEPTH_DEFAULT`, `PACKET_DISPATCHER_TASK_STACK_DEPTH`

1) Stack depth macros

> **NOTE:** `PACKET_DISPATCHER_TASK_STACK_DEPTH` is currently defined but not actually used in the provided implementation!

```
#define PACKET_HANDLER_TASK_STACK_DEPTH_DEFAULT ((configSTACK_DEPTH_TYPE)512U)
#define PACKET_DISPATCHER_TASK_STACK_DEPTH ((configSTACK_DEPTH_TYPE)1024U)
```

---

2) packet_handler_t (callback)

```
typedef result_t (*packet_handler_t)(void* buffer);
```

This type represents the **callback function** invoked by a **handler task** when a packet of its type is received.

Parameters

- `buffer`

Pointer to the **decoded packet payload** copied from the queue. The actual type of `buffer` depends on the registered `packet_type` in the config for the handler (see packet_handler_config_t).

For example, if a handler is registered for one specific protobuf payload type, the handler should cast `buffer` to the corresponding generated struct type.

Example

```
static result_t Callback_ArmBoardControlSignals(void *buffer) {
    ArmBoardControlSignals* pckt = (ArmBoardControlSignals *)buffer;
    }
```

Note on buffer typecasting

> The callback receives only a raw `void *`. That means type safety is **entirely dependent on correct configuration!**

- `packet_type` must match the actual protobuf payload member
- `item_size` must match the size of that decoded payload type
- handler must cast `buffer` to the correct struct type

If any of those mismatch, the code may compile while quietly doing something stupid (and it will be your fault :D).

Return value

Returns `result_t`. The handler task logs a warning if the return value is not `RESULT_OK`.

---

3) packet_handler_config_t (struct)

> **NOTE:** there exist macros to make the configuration easier! **See:** Helper Macros for Static Handler Config

```
typedef struct {
    packet_handler_t handler;
    const char* task_name;
    pb_size_t packet_type;
```

```
    UBaseType_t task_priority;
    configSTACK_DEPTH_TYPE task_stack_depth;

    size_t item_size;
    UBaseType_t queue_length;

    uint8_t* queue_buffer;
    StaticQueue_t queue_struct;
    QueueHandle_t queue;
} packet_handler_config_t;
```

Purpose

Describes one packet type and the task/queue resources needed to process it. **Each entry in the handler config array** (passed to PacketDispatcherInit(...)) **corresponds to one routed packet type!**

Fields

- `handler`
  Callback invoked when a packet of this type is received. Must not be `NULL`.
- `task_name`
  Name used when creating the FreeRTOS task. Must not be `NULL`.
- `packet_type`
  The protobuf discriminator value to match against `DecodingEnvelopeCurrent.which_payload`, which is the routing key.
- `task_priority`
  Priority of the FreeRTOS handler task. If set to zero, that is still a valid FreeRTOS priority value. There is no separate “unset” semantic here.
- `task_stack_depth`
  Stack depth for the handler task.
  If `<= 0`, the implementation replaces it with: `PACKET_HANDLER_TASK_STACK_DEPTH_DEFAULT`. Since this type is typically unsigned, the `<= 0` check effectively means “zero” in practice.
- `item_size`
  Size of **one** queued item.

> This must match the size of the decoded payload type copied into the queue.

- `queue_length`
  Number of items the queue can hold.
- `queue_buffer`
  Backing storage for static queue data.

Must be large enough for `queue_length * item_size`

- `queue_struct`
  Static queue control structure used internally by `xQueueCreateStatic()`. Caller provides storage but should not manually initialize runtime content.
- `queue`
  Queue handle written internally during initialization.

> Caller should not pre-fill it!

---

4) PacketDispatcherInit(...)

```
result_t PacketDispatcherInit(packet_handler_config_t* handlers,
                              size_t handler_count);
```

Initializes the dispatcher by...

- storing the handler registry
- creating **one queue** and **one task** per handler entry

Parameters

- `handlers`
  Pointer to an array of handler configurations (see above: packet_handler_config_t). The implementation stores a global pointer to it and passes individual entries to tasks.
- `handler_count`
  Number of entries in the array.

> **IMPORTANT:** The `handlers` array **must remain valid for the full lifetime** of the system. Do **NOT** allocate this array on a temporary stack frame unless you are into being abused by segfaults :)

---

5) DispatchPacket(...)

```
void DispatchPacket(receive_frame* incoming_packet);
```

Decodes one incoming raw frame and **routes** its decoded payload to the appropriate handler queue.

Internal functioning

1. validates basic frame properties
2. creates a nanopb input stream from the raw bytes
3. decodes into the global static `DecodingEnvelopeCurrent`
4. scans the registered handler list
5. finds the first handler whose `packet_type` matches `which_payload`
6. sends `DecodingEnvelopeCurrent.payload` to that handler's queue
7. returns

If no matching handler is found, it logs a warning. If decode fails, it logs an error.

> **NOTE:** This function returns `void`, so dispatch failure is **only observable through logs**.

Parameters

- `incoming_packet`
  Pointer to a transport frame containing `payload`, `len` of the incoming packet.

Internal (private) task model

`PacketHandlerTask()`

> Also see note on handler task lifecycles !

Each handler config gets its own task (and corresponding queue, remember ladies?) running this loop:

1. validate config and resources
2. allocate one packet buffer using `malloc(conf->item_size)`
3. block forever on `xQueueReceive()`
4. when a packet arrives:
   - call `conf->handler(packet_buffer)`
   - log if handler returns error

Purpose of per-task buffer

The queue copies incoming items into the task's local `packet_buffer`. That means the handler callback receives a stable task-local buffer for the duration of the callback. The callback does **not** receive a pointer directly into the global decode object.

> The task allocates its buffer dynamically with `malloc()` once at startup and never frees it, because the task is intended to live forever.

Macros

> There exist macros to make the configuration of a handler easier! See: Helper Macros for Static Handler Config.

Revision #17
Created 2026-05-18 14:00:41 UTC by Lisa te Braak
Updated 2026-05-24 17:04:28 UTC by Lisa te Braak