Packet Dispatcher

High Level Overview

This Page

Purpose

The packet dispatcher is used to decode protobuf frames.

Application code usually wants:

- **strongly typed** decoded payloads
- **one handler** per packet type
- **decoupling** between input reception and packet processing

This module solves that by:

1. **receiving** a raw protobuf `receive_frame`
2. **decoding** it into `PBEnvelope`
3. **determining** `which_payload`
4. **finding** the corresponding handler
5. **copying** the decoded payload **into** that handler's (freeRTOS) **queue**
6. letting a dedicated task **call callback** for this handler

In short, each packet type gets its own handler callback, queue and task. That makes the system modular and easy to extend, at least conceptually. So the module acts as a bridge between **transport-level bytes** and **application-level packet handler.**

> In practical terms, it is a **decode-and-dispatch layer** between an **input source** that receives raw bytes and **a set of application handlers** that want already-decoded payloads

>

> The implementation has **some assumptions and hazards** that absolutely need to be understood before you start messing with its internal structure.

High-level design

The design has three major parts:

- Global handler registry
  The global handler registry contains an **array** of packet handler tasks (see packet_handler_config_t). A packet handler task configures (amongst other things) the callback function for a certain type of packet.
  The array of configs is given by the caller at initialization time. This array is stored globally and **used by dispatch logic for packet type lookup**.

> What we call a **packet** is a raw protobuf.
> What we call a **handler** is a (configuration of a) callback function for a specific protobuf/packet.

- One queue & task per packet type
  The dispatcher takes each handler configuration and creates **1 FreeRTOS queue** and **1 FreeRTOS task**. When receiving messages, the dispatcher enqueues decoded payloads into the corresponding queue. **The corresponding task blocks that queue and calls the handler callback** (which saved in the registry).

> The task takes the **correspoding** payload out of the queue and calls the specified handler/callback function. By corresponding we mean that each type of packet has their own queue.

- Shared decode step
  Incoming frames are decoded into a global static `PBEnvelope` object: `static PBEnvelope DecodingEnvelopeCurrent;`.
  The dispatcher then copies `DecodingEnvelopeCurrent.payload` into a handler queue. **This detail matters a lot for concurrency and payload sizing.**

NOTE on handler task lifecycles

>

> Each handler task is intended to live forever.

A task is responsible for passing a specific packet type from the corresponding queue to the correct callback. As stated above, a handler task **gets created by the dispatcher** according to the configuration (see packet_handler_config_t) done by the caller when initializing the dispatcher.

Lifecycle

1. **Created by** `PacketHandlerStart()`
   As part of PacketDispatcherInit().
2. **Validate configuration**
   Task_name, handler and queue need to be present for it to work. These params are set in packet_handler_config_t. If you use the macros, this should be fine.
3. **Allocate local packet buffer**
4. **Block forever on queue receive**
   So, when we receive a packet in the corresponding queue, we wait for it to be handled.
5. **Process packets as they arrive**
   The processing is done by the callback specified in the handler.

Terminates only if...

- config is **invalid**
- queue is **null**
- heap **allocation fails** for packet buffer

In those cases it deletes itself.

> At the moment, there is no restart or supervision mechanism in this module!

External dependencies

> This is not a standalone module. It sits in the middle of RTOS tasking, protobuf decoding, and transport reception.

This module depends on:

	specifically used pieces

FreeRTOS	• `xQueueCreateStatic` • `xQueueReceive` • `xQueueSend` • `xTaskCreate` • `vTaskDelete`
nanopb / protobuf decoding	• `pb_istream_from_buffer` • `pb_decode`
`PBEnvelope` generated protobuf definitions	• `PBEnvelope_fields` • `PBEnvelope_size`
Logging library	
Result Library	
`stm/ethernet_udp.h`	• `receive_frame`

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.

Helper Macros for Handler Config

Purpose

To reduce repetitive boilerplate when defining packet handlers, the module also provides a set of helper macros in `packet_dispatcher_macros.h`.

These macros generate:

- a statically allocated queue buffer
- a fully initialized `packet_handler_config_t`

> They are especially useful because they automatically derive the correct queue item size from the selected `PBEnvelope` payload member, which helps avoid one of the easiest mistakes in this module: mismatching `item_size` with the actual decoded protobuf payload type.

Why these macros are useful

Without these macros, every handler config has to manually specify:

- queue storage buffer
- queue length
- item size
- task name
- default priority
- default stack depth
- queue initialization fields

That is tedious and error-prone.

I) They derive `item_size` automatically

Each macro uses: `sizeof(((PBEnvelope*)0)->payload.payload_member)` to compute the exact size of the selected envelope payload member at compile time. This removes the need to manually write `.item_size = sizeof(MyPayloadType)` and reduces the chance of queue item size mismatches.

II) They allocate queue storage automatically

Each macro also declares:

```
static uint8_t name##_queue_buffer[...];
```

with the correct total size based on:

- payload member size
- selected queue length

So the queue backing storage is generated alongside the config object.

Important consequence of these macros

> These macros define **static objects**.

That means each use creates:

- a static queue buffer
- a static `packet_handler_config_t`

> This is generally what you want for a dispatcher configuration that should live for the full lifetime of the system.

It also means:

- they should normally be used at file scope
- using the same `name` twice in one translation unit will cause symbol redefinition
- they are not runtime factory macros, they are compile-time object definition helpers

---

Shared Functionality

For **all** of these macros, the generated config uses:

```
#define PACKET_HANDLER_CONFIG_STATIC(name, packet_tag, payload_member_size, handler_fn)


.handler = (handler_fn)
.task_name = #name
.packet_type = (packet_tag)
```

```
.item_size = payload_member_size
.queue_buffer = name##_queue_buffer
.queue_struct = {0}
.queue = NULL
```

This is helpful for two reasons:

- `task_name` is automatic
  The task name becomes **the same as the symbol (handler config itself) name**, which keeps config definitions compact and readable.
- Queue internals are initialized consistently
  The queue control structure is zero-initialized, and the runtime queue handle starts as `NULL`, matching the expectations of the dispatcher startup code.

IMPORTANT NOTE on `payload_member`

> The `payload_member` argument is not the packet type name. It is the **member name inside `PBEnvelope.payload`!**

This matters because the macros compute size using direct member access syntax: `sizeof(((PBEnvelope*)0)->payload.payload_member)`. **So, if the wrong member name is used, compilation will fail, which is actually helpful for once.**

The member names are defined in `envelope.pb.h` .
For example, currently `envelope.pb.h` contains the following:

```
typedef struct _PBEnvelope {
    pb_size_t which_payload;

    union _PBEnvelope_payload {
            /* Sensorboard messages */
            SensorBoardPHInfo ph_info;

            /* Armboard messages */
            ArmBoardControlSignals arm_ctrl;
            ArmBoardDiagnostics arm_diag;

            //etc etc...
    }
}
```

> So, the macro must be called with the member name **matching the rest of the config**, such as `ph_info` or `arm_ctrl` and **NOT** the protobuf struct type name!

Available macros

1) Default configuration macros

The header defines these default values:

```
#define PACKET_HANDLER_DEFAULT_PRIORITY (tskIDLE_PRIORITY + 2U)
#define PACKET_HANDLER_DEFAULT_QUEUE_LENGTH (5U)
#define PACKET_HANDLER_DEFAULT_STACK_DEPTH (0U)
```

- `PACKET_HANDLER_DEFAULT_PRIORITY`
  Default FreeRTOS task priority assigned to handler tasks created with the simpler macros.
- `PACKET_HANDLER_DEFAULT_QUEUE_LENGTH`
  Default number of queued packets per handler.
- `PACKET_HANDLER_DEFAULT_STACK_DEPTH`
  Default stack depth field stored in the config.
  A value of `0U` is intentional here. In the dispatcher implementation, a task stack depth of zero is treated as “use the dispatcher default,” which becomes: `PACKET_HANDLER_TASK_STACK_DEPTH_DEFAULT`. So this macro does **not** mean “zero stack.” It means “defer to the runtime default chosen by the dispatcher.”

---

2) Basic config: `PACKET_HANDLER_CONFIG_STATIC`

```
#define PACKET_HANDLER_CONFIG_STATIC(name, packet_tag, payload_member_size, handler_fn)
```

This is the simplest form. Creates a handler config using:

- **default** priority
- **default** queue length
- **default** stack depth behaviour

Parameters

- `name`
  User defined name, go crazy.

- `packet_tag`
  This is the Nanopb generated tag for the packet type. They follow the pattern `PBEnvelope_[payload_member]_tag`. So for example: PBEnvelope_arm_ctrl_tag
- `payload_member`
  See important note on payload_member. **Needs to match the packet_tag and the buffer type the callback is specified for!**
- `handler_fn`
  Callback function. Type signature packet_handler_t.

Example

```
/* Config for: ArmBoardMovementFeedback */

//Define the callback function with the specified signature
static result_t Callback_ArmBoardMovementFeedback(void *buffer) {
  if (buffer == NULL) {
    return RESULT_ERR_INVALID_ARG;
  }

  //Retreive the packet
  ArmBoardMovementFeedback* pckt = (ArmBoardMovementFeedback *)buffer;
  //Get all fields
  pckt->arm_error;

  /*
  Go wild...
  */
  return RESULT_OK;
}

PACKET_HANDLER_CONFIG_STATIC(
  Handler_ArmBoardMovementFeedback,   // NOTE: This name is USER DEFINED, let your imagination
run
  PBEnvelope_arm_feedback_tag,        //  Make sure these...
  arm_feedback,                       //                   ... MATCH!
  Callback_ArmBoardMovementFeedback); // Callback as above
```

3) Full config: `PACKET_HANDLER_CONFIG_STATIC_EX`

```
#define PACKET_HANDLER_CONFIG_STATIC_EX(name, packet_tag, payload_member, handler_fn,
                                        priority_, stack_depth_, queue_length_)
```

Full explicit version. Lets you set:

- name, packet_tag, payload_member, handler_fn as above
- **custom** priority
- **custom** stack depth
- **custom** queue length

Best used when

- the handler needs a non-default stack size
- you want fully explicit resource configuration

Example

```
PACKET_HANDLER_CONFIG_STATIC_EX(vision_handler_cfg,
                                PBEnvelope_detected_object_tag,
                                detected_object,
                                handle_detected_object,
                                tskIDLE_PRIORITY + 3U,
                                768U,
                                16U);
```

---

these r not in the code lol

begin here

II) `PACKET_HANDLER_CONFIG_STATIC_QUEUE`

```
#define PACKET_HANDLER_CONFIG_STATIC_QUEUE(name, packet_tag, payload_member_size,
handler_fn, queue_length_)
```

Same as the basic macro, but lets you override queue length.

Best used when

- handler needs a longer or shorter queue
- default priority is still fine

Example

```
PACKET_HANDLER_CONFIG_STATIC_QUEUE(sensor_handler_cfg,
                                   PBEnvelope_sensor_diag_tag,
                                   sensor_diag,
                                   handle_sensor_diag,
                                   12);
```

III) `PACKET_HANDLER_CONFIG_STATIC_PRIO`

```
#define PACKET_HANDLER_CONFIG_STATIC_PRIO(name, packet_tag, payload_member, handler_fn,
priority_)
```

Same as the basic macro, but lets you override task priority.

Best used when

- one handler must run at a different RTOS priority
- default queue length is still fine

Example

```
PACKET_HANDLER_CONFIG_STATIC_PRIO(emergency_handler_cfg,
                                  PBEnvelope_arm_obstructions_tag,
                                  arm_obstructions,
                                  handle_arm_obstructions,
                                  tskIDLE_PRIORITY + 4U);
```

IV) `PACKET_HANDLER_CONFIG_STATIC_PRIO_QUEUE`

```
#define PACKET_HANDLER_CONFIG_STATIC_PRIO_QUEUE(    name, packet_tag, payload_member,
handler_fn, queue_length_, priority_)
```

Lets you override both:

- queue length
- task priority

Best used when

- a handler has non-default scheduling needs
- and also non-default backlog requirements

Example

```
PACKET_HANDLER_CONFIG_STATIC_PRIO_QUEUE(nav_handler_cfg,
                                        PBEnvelope_ph_info_tag,
                                        ph_info,
                                        handle_ph_info,
                                        10,
                                        tskIDLE_PRIORITY + 3U);
```

---

end here

Recommended Usage Pattern

> More information on the mentioned steps can be found in Functions of the Packet Dispatcher

---

Typical Usage Model

Intended setup

1. Define one handler function per packet type
2. define one packet_handler_config_t entry per packet type (using the macros)
3. provide queue storage buffers

   (When using the macros, you do not need to do this manually)
4. call PacketDispatcherInit(...)
5. whenever a frame arrives, call DispatchPacket()

Flow after setup

1. Ethernet/UDP receives raw frame
2. networking code builds `receive_frame`
3. `DispatchPacket()` decodes it
4. payload type is matched
5. decoded payload is copied into target queue
6. matching handler task wakes
7. the callback processes typed payload

IMPORTANT configuration rules

> This module is heavily configuration-driven. Several things must match exactly.

I. `packet_type` must match the protobuf discriminator

Each handler's `packet_type` must be the exact value used by `PBEnvelope.which_payload`. If this is wrong, packets will never reach that handler.

II. `item_size` must match the decoded payload type

The queue copies bytes from `&DecodingEnvelopeCurrent.payload` into a queue item of size `item_size`.

If `item_size` is:

- too small -> payload will be truncated
- too large -> copied data may include unrelated union bytes or layout assumptions
- wrong type entirely -> handler sees garbage with confidence

III. `queue_buffer` must be sized correctly

The backing storage must be at least: `queue_length * item_size`. **If not, queue creation or runtime behavior is invalid.**

IV. Handler must cast `void *` correctly

The callback receives a raw buffer pointer. It must cast to the correct generated protobuf type.

V. Handlers array must be an array of structs

The current `PacketDispatcherInit()` API expects:

```
packet_handler_config_t* handlers
```

meaning a contiguous array of structs, not an array of pointers.

So with the current implementation, the final array should actually be:

```
static packet_handler_config_t* handlers[] = {
    drive_handler_cfg,
    sensor_diag_handler_cfg,
};
```

NOT an array of pointers.

Examples

1) Using macros

```
//Imports
#include "packet_dispatcher.h"
#include "packet_dispatcher_macros.h"

/*Define handler callbacks*/
//Callback for protobuf of type ArmBoardMovementFeedback
static result_t Callback_ArmBoardMovementFeedback(void *buffer) {
  if (buffer == NULL) {
        return RESULT_ERR_INVALID_ARG;
    }

  ArmBoardMovementFeedback* pckt = (ArmBoardMovementFeedback *)buffer; //Retreive the packet
  pckt->arm_error; //Get fields of protobuf
  //Do something...

  return RESULT_OK;
}

//Config using most basic macro
PACKET_HANDLER_CONFIG_STATIC(Handler_ArmBoardMovementFeedback, PBEnvelope_arm_feedback_tag,
arm_feedback, Callback_ArmBoardMovementFeedback);

//Callback for protobuf of type ArmBoardControlSignals
static result_t Callback_ArmBoardControlSignals(void *buffer) {
  if (buffer == NULL) {
        return RESULT_ERR_INVALID_ARG;
    }

    ArmBoardControlSignals* pckt = (ArmBoardControlSignals *)buffer;
    pckt->control_base; //Get fields of protobuf
    pckt->control_gripper_pitch;
    //... etc etc
    //Do something...

    return RESULT_OK;
}

//Config using most basic macro
```

```
PACKET_HANDLER_CONFIG_STATIC(Handler_ArmBoardControlSignals, PBEnvelope_arm_ctrl_tag,
arm_ctrl, Callback_ArmBoardControlSignals);

//Add configs to the list of configs
static packet_handler_config_t* handlers[] = {Handler_ArmBoardMovementFeedback,
Handler_ArmBoardControlSignals};

//HERE WE PUT ETH_init(...) and the creation of queues from the networking board
//See respective documentation

PacketDispatcherInit(handlers, 2);
ETH_udp_init(2, queues, DispatchPacket); //Passing DispatchPacket to ETH_udp_init makes sure
it gets called upon receiving msgs

//Once again, after this we can use networking and do ETH_add_arp(...) and ETH_udp_send(...)
```

2) Manual configuration

```
//Imports
#include "packet_dispatcher.h"

static result_t handle_drive_cmd(void* buffer) {
    PBDriveCommand* msg = (PBDriveCommand*)buffer;
    return drive_process(msg);
}

static result_t handle_arm_cmd(void* buffer) {
    PBArmCommand* msg = (PBArmCommand*)buffer;
    return arm_process(msg);
}

static uint8_t drive_queue_storage[8 * sizeof(PBDriveCommand)];
static uint8_t arm_queue_storage[4 * sizeof(PBArmCommand)];

static packet_handler_config_t handlers[] = {
    {
        .handler = handle_drive_cmd,
```

```
        .task_name = "drive_pkt",
        .packet_type = PBEnvelope_drive_cmd_tag,
        .task_priority = 3,
        .task_stack_depth = 512,
        .item_size = sizeof(PBDriveCommand),
        .queue_length = 8,
        .queue_buffer = drive_queue_storage,
    },
    {
        .handler = handle_arm_cmd,
        .task_name = "arm_pkt",
        .packet_type = PBEnvelope_arm_cmd_tag,
        .task_priority = 3,
        .task_stack_depth = 512,
        .item_size = sizeof(PBArmCommand),
        .queue_length = 4,
        .queue_buffer = arm_queue_storage,
    },
};
```

Then during startup:

```
result_t res = PacketDispatcherInit(handlers, ARRAY_LEN(handlers));
```

And during frame reception:

```
DispatchPacket(&rx_frame);
```