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`

Revision #9
Created 2026-05-18 13:59:24 UTC by Lisa te Braak
Updated 2026-05-24 17:03:30 UTC by Lisa te Braak