

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* pkt = (ArmBoardMovementFeedback *)buffer; //Retrieve the packet
    pkt->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* pkt = (ArmBoardControlSignals *)buffer;
    pkt->control_base; //Get fields of protobuf
    pkt->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);
```

Revision #8

Created 2026-05-18 14:05:00 UTC by Lisa te Braak

Updated 2026-05-24 17:26:17 UTC by Lisa te Braak