RustNL

postcard

an unreasonably effective tool for machine to machine communication

🦋: @jamesmunns.com
🕸: onevariable.com
💬: @jamesmunns:beeper.com
🐦: @bitshiftmask

ONE VARIABLE

i'm james!

🦋: @jamesmunns.com
🕸️: onevariable.com
💬: @jamesmunns:beeper.com
🐦: @bitshiftmask

ONE VARIABLE

ONE VARIABLE

ONE VARIABLE

(consulting)

I work on
"embedded systems"

this is a vague term that covers a lot

computers you don't sit in front of

By James086 - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=3520225

or

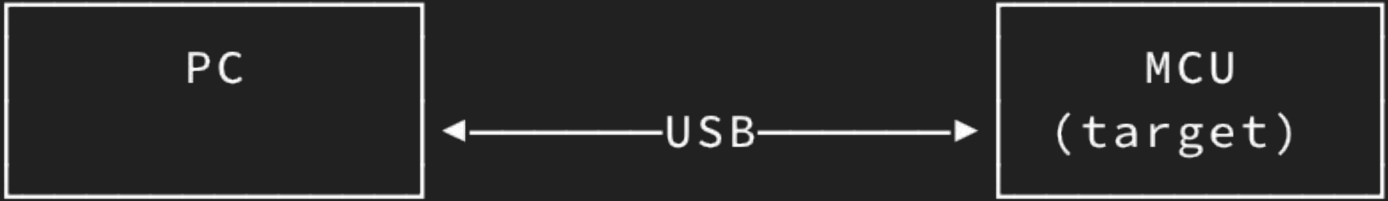
By Polimerek - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=425348 (one of) my most common tasks:

help computers talk to other computers

sometimes all the decisions are made

other times it's a **paralyzing void** of options stare into the void with me

what do you do when the whole plan looks like this?



PC

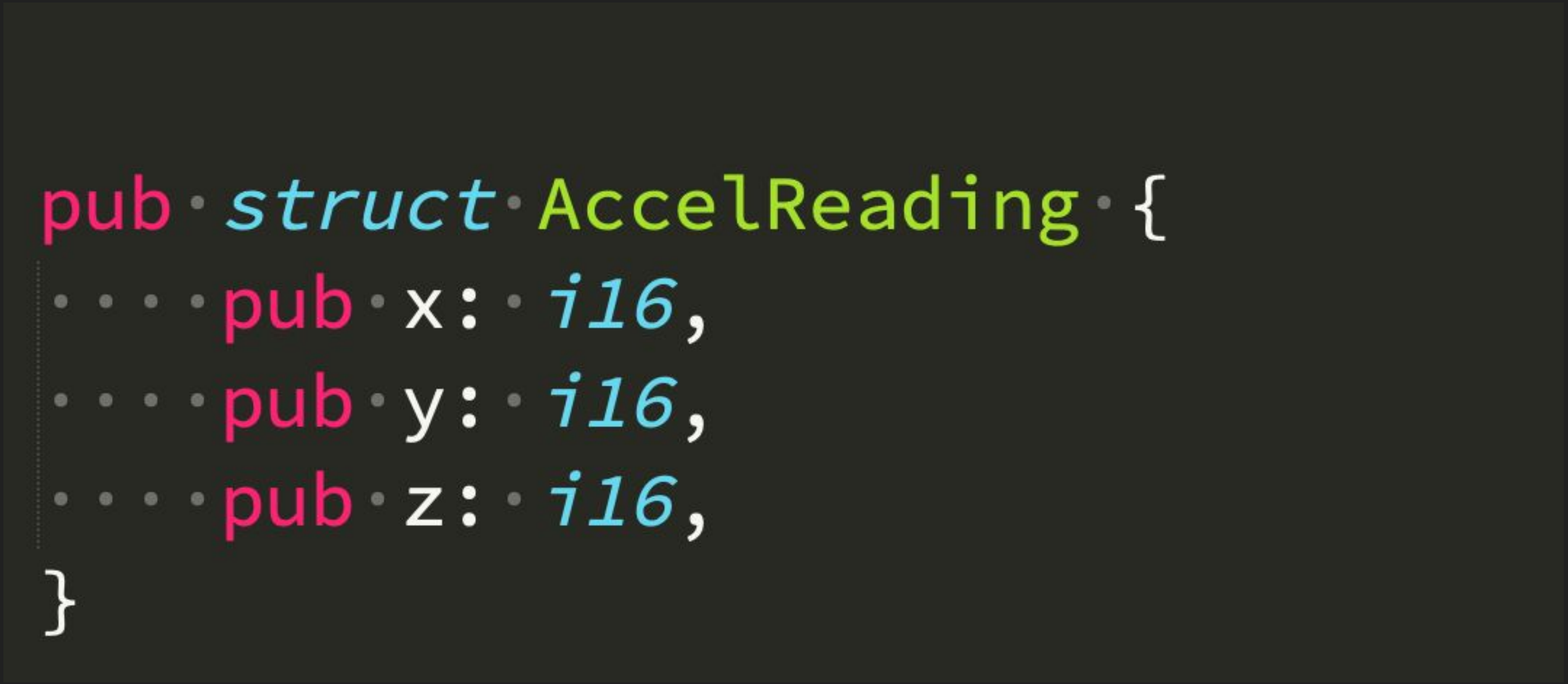
USB

MCU
(target)

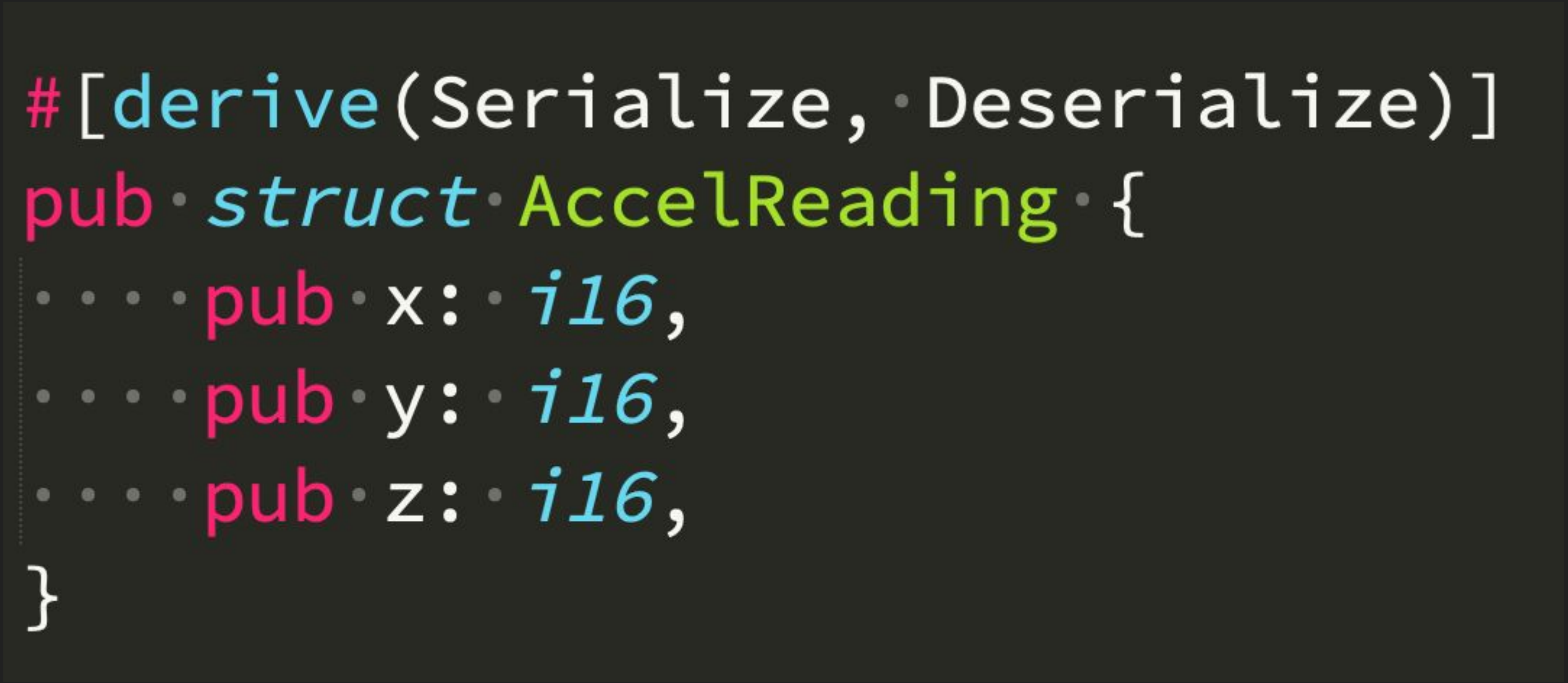
by the way: the first demo is in a week, better start quick first blessing: most of my customers are using Rust for both PCs and MCUs this makes everything so much easier.

second blessing: we have `serde`

“I want to put Rust data on the wire”



```
pub struct AccelReading {
    pub x: i16,
    pub y: i16,
    pub z: i16,
}
```



```rust
#[derive(Serialize, Deserialize)]
pub struct AccelReading {
    pub x: i16,
    pub y: i16,
    pub z: i16,
}
```

solved (halfway)

serde is the “front half”:
focused on Rust types

we need the other half:
a “data format”
or, “what we put on the wire”

embedded systems don't always have things like operating systems or allocators

```
#![no_std]
```

in 2019, there weren't many serde data formats that worked with `#![no_std]` postcard

I wrote down five design goals:
(lightly edited)

1

design primarily for `#![no_std]` usage: it has to work on the tiniest chips

ONE VARIABLE

2

make it as easy to use as any other `serde` format

3

no “special code”:
same on your PC and MCU

4

be resource efficient:
memory usage, code size,
developer time, and CPU time;
in that order

5

allow for context-specific customization options

I made the format as dumb as I could

no bit packing trickery, only bytes

competing against transmuting to/from bytes and calling `memcpy`

(don't do that lightly, btw)

"non self-describing":
only send raw values, nothing more

everything is little endian in postcard

structs and tuples just go in lexical order slices are prefixed by their length

enums are prefixed by their discriminant originally: slice lengths and discriminants were “varints” - variable length integers

today: all integers are “varints” for portability reasons

provide “flavors”:
stackable combinators on ser/de

works a lot like iterators

with helper functions to hide the generics:

```
postcard::to_slice(&data, &mut buf)
```

```
postcard::to_slice_cobs(&data, &mut buf)
```

2019-2022: I used `postcard` a lot!

2019-2022: so did other people!

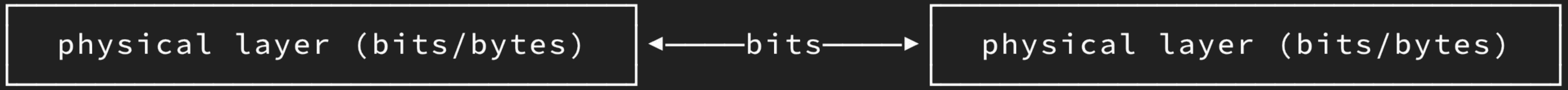
2022: time for 1.0

clean up the last portability issues, stabilize the wire format

2022-2024: I kept using `postcard` a lot!

it has gotten to the point where MOST of my projects use it somehow but I had a problem

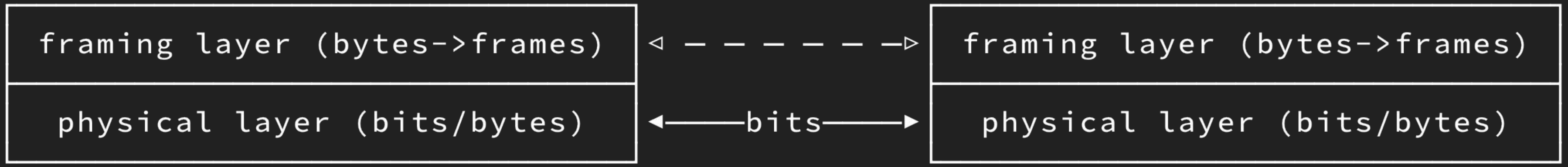
sometimes good tools can make the wrong thing **TOO** easy to do let's look at the stack:



physical layer (bits/bytes)

bits

physical layer (bits/bytes)



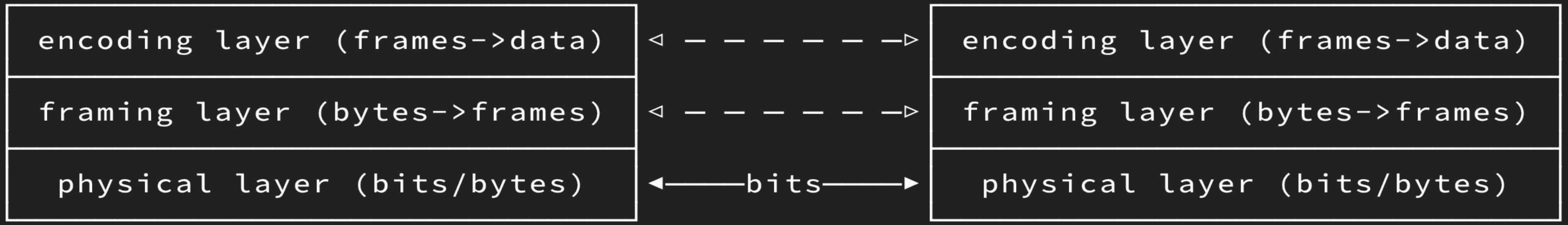
framing layer (bytes->frames)

physical layer (bits/bytes)

bits

framing layer (bytes->frames)

physical layer (bits/bytes)



encoding layer (frames->data)

framing layer (bytes->frames)

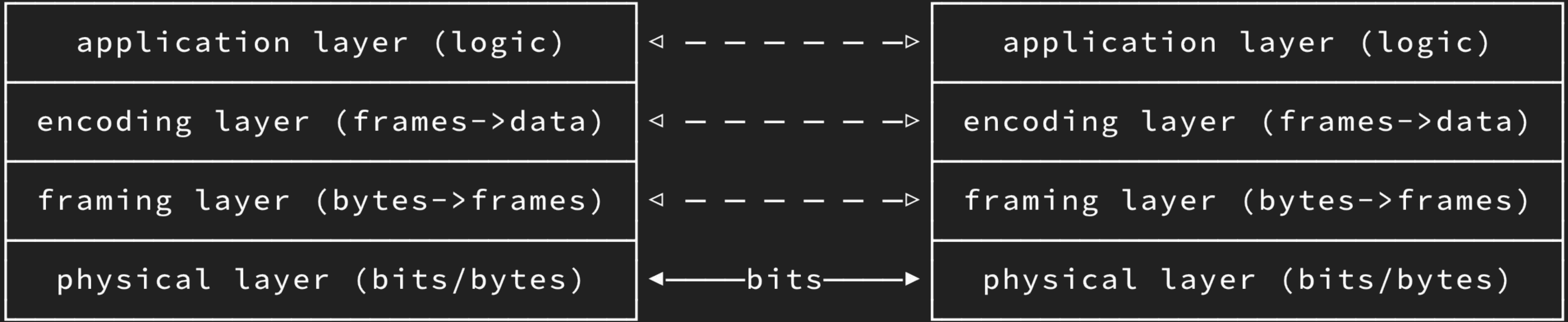
physical layer (bits/bytes)

bits

encoding layer (frames->data)

framing layer (bytes->frames)

physical layer (bits/bytes)



application layer (logic)

encoding layer (frames->data)

framing layer (bytes->frames)

physical layer (bits/bytes)

bits

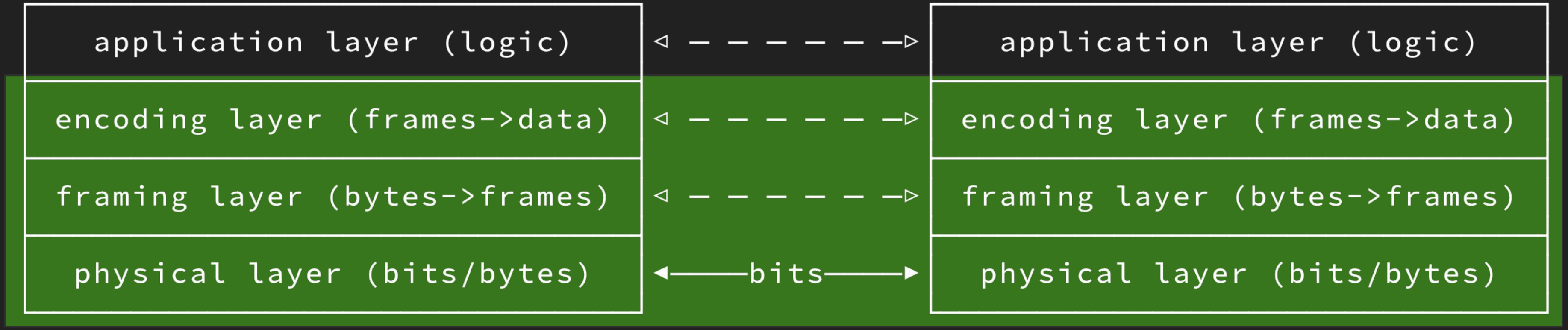
application layer (logic)

encoding layer (frames->data)

framing layer (bytes->frames)

physical layer (bits/bytes)

ONE VARIABLE



application layer (logic)

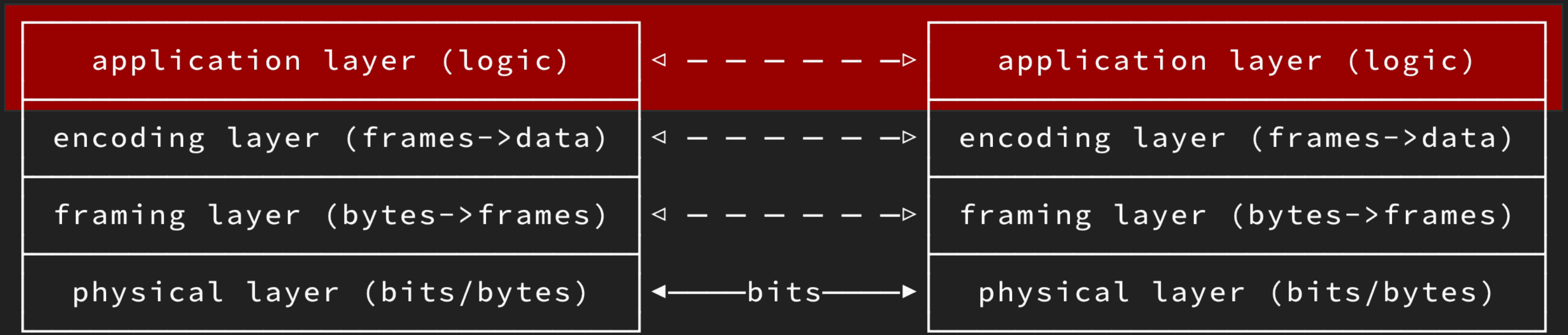
encoding layer (frames->data)

framing layer (bytes->frames)

physical layer (bits/bytes)

easy!

hard!



application layer (logic)	◁ – – – – – –▷	application layer (logic)
encoding layer (frames->data)	◁ – – – – – –▷	encoding layer (frames->data)
framing layer (bytes->frames)	◁ – – – – – –▷	framing layer (bytes->frames)
physical layer (bits/bytes)	◀———bits———▶	physical layer (bits/bytes)

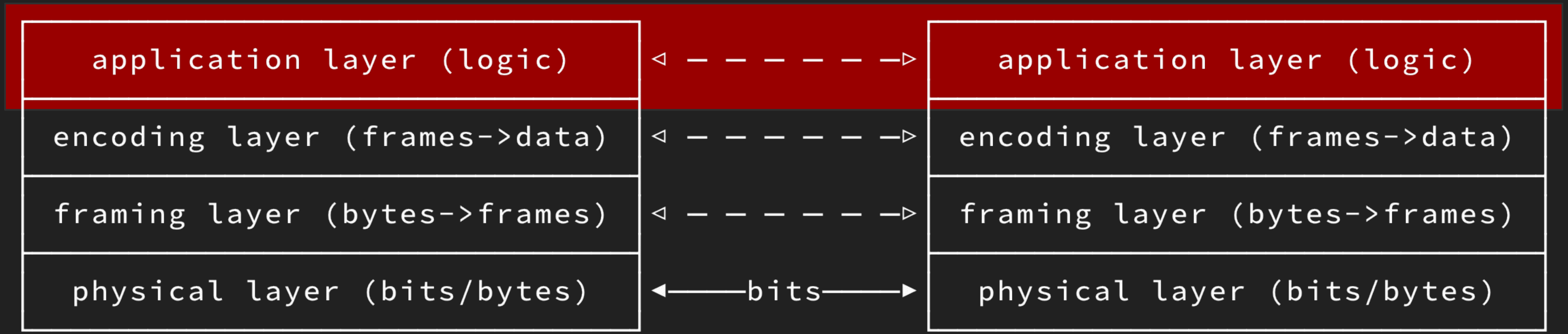
(at least for nontrivial projects)

every project required a lot of custom work not *hard* work, just *tedious* work

(usually a lot of copy and pasting)

I couldn't figure out how to abstract this

hard!



application layer (logic)
encoding layer (frames->data)
framing layer (bytes->frames)
physical layer (bits/bytes)
bits
application layer (logic)
encoding layer (frames->data)
framing layer (bytes->frames)
physical layer (bits/bytes)

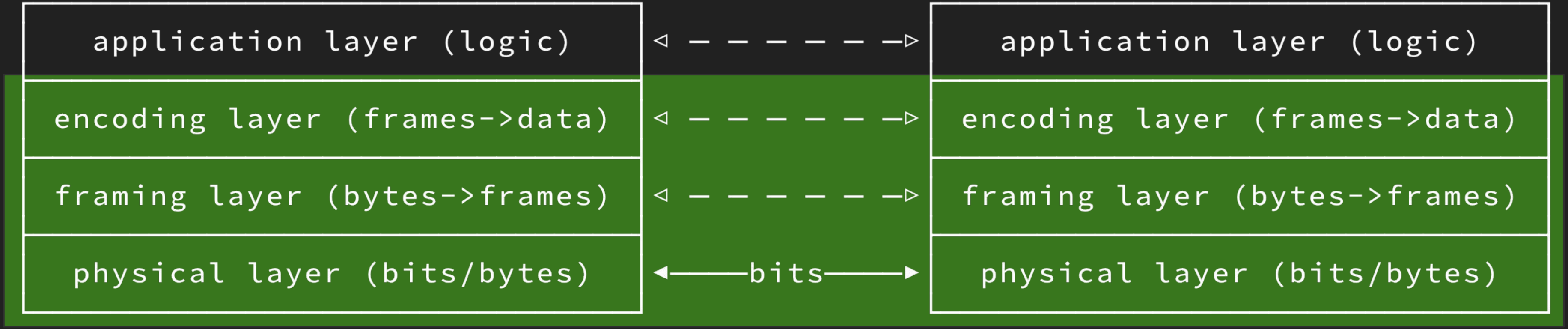
(at least for nontrivial projects)

until I realized that I was missing a defined “protocol” layer

application: “how to behave”
or "business logic”

protocol: “how to communicate”

I was reinventing a protocol for *every* project postcard made it TOO easy to build bespoke protocols - but I had to do it every time what if there was a protocol available *out of the box?*



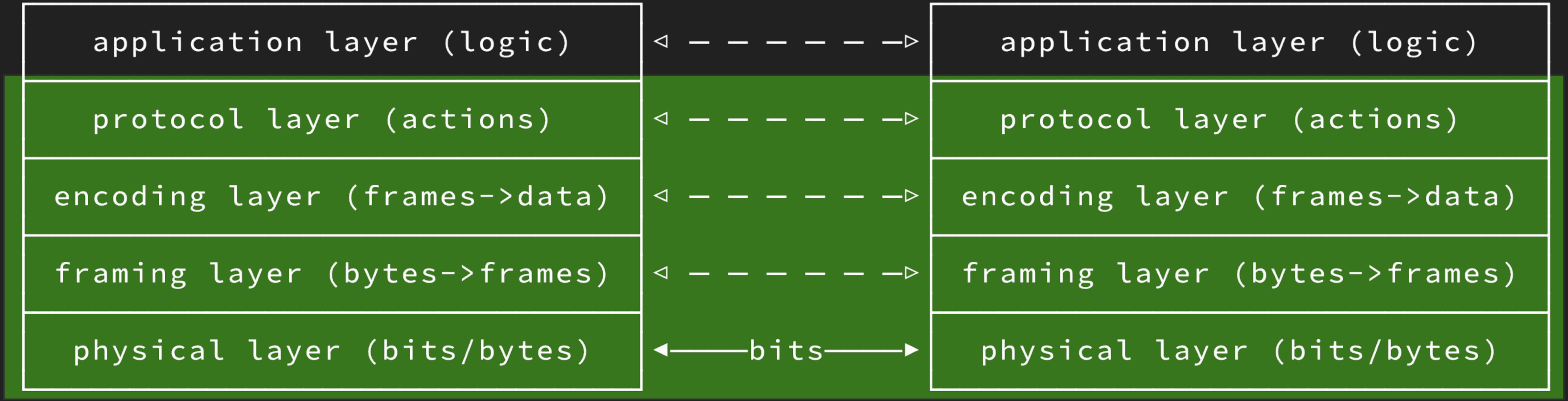
application layer (logic) ◁ – – – – – –▷ application layer (logic)

encoding layer (frames->data) ◁ – – – – – –▷ encoding layer (frames->data)

framing layer (bytes->frames) ◁ – – – – – –▷ framing layer (bytes->frames)

physical layer (bits/bytes) ◀——bits——▶ physical layer (bits/bytes)

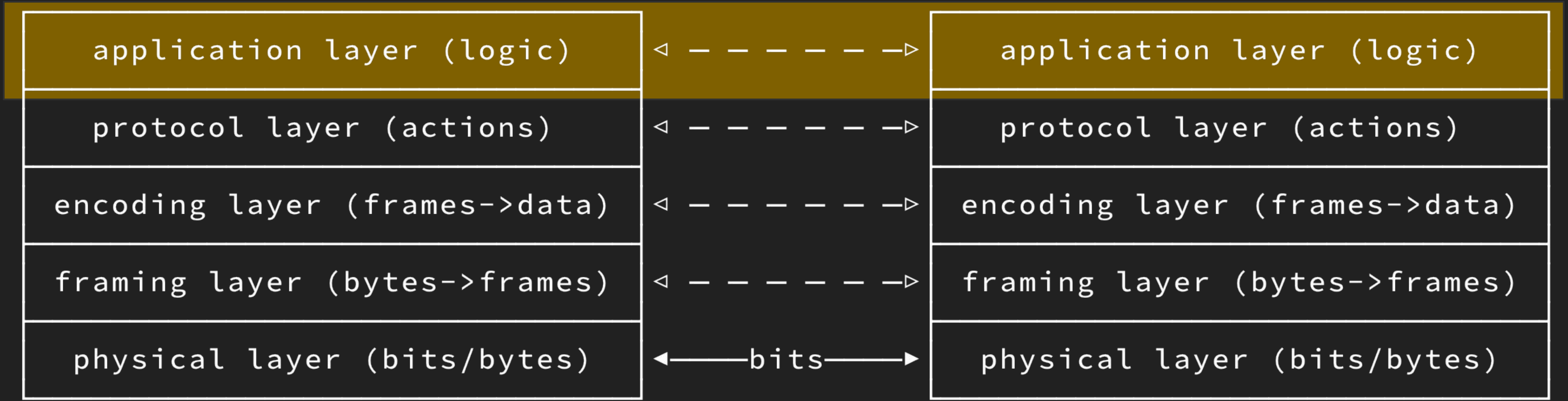
easy!



application layer (logic)	◁ – – – – – –▷	application layer (logic)
protocol layer (actions)	◁ – – – – – –▷	protocol layer (actions)
encoding layer (frames->data)	◁ – – – – – –▷	encoding layer (frames->data)
framing layer (bytes->frames)	◁ – – – – – –▷	framing layer (bytes->frames)
physical layer (bits/bytes)	◀——bits——▶	physical layer (bits/bytes)

easy!

still custom, but that's life



application layer (logic) ◁ — — — — — —▷ application layer (logic)

protocol layer (actions) ◁ — — — — — —▷ protocol layer (actions)

encoding layer (frames->data) ◁ — — — — — —▷ encoding layer (frames->data)

framing layer (bytes->frames) ◁ — — — — — —▷ framing layer (bytes->frames)

physical layer (bits/bytes) ◀——bits——▶ physical layer (bits/bytes) meet `postcard-rpc`:

a *protocol* on top of `postcard`

ONE VARIABLE

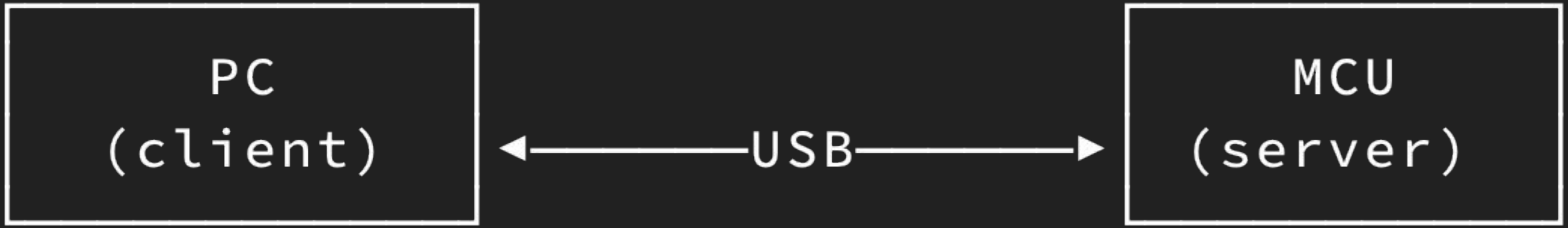
what behaviors does the `postcard-rpc` protocol have?

ONE VARIABLE

1

defined "client" and "server" roles

1



PC
(client)

USB

MCU
(server)

2

rpc: "remote procedure call"

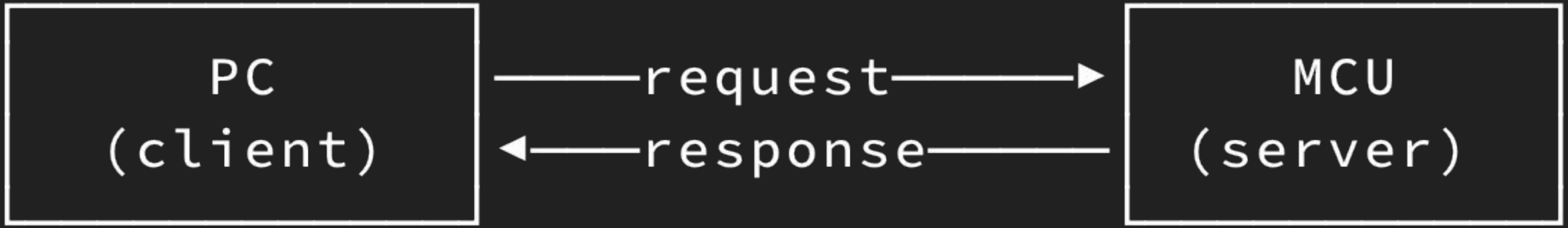
2

"every request gets a response"

2

client initiates, server responds

2



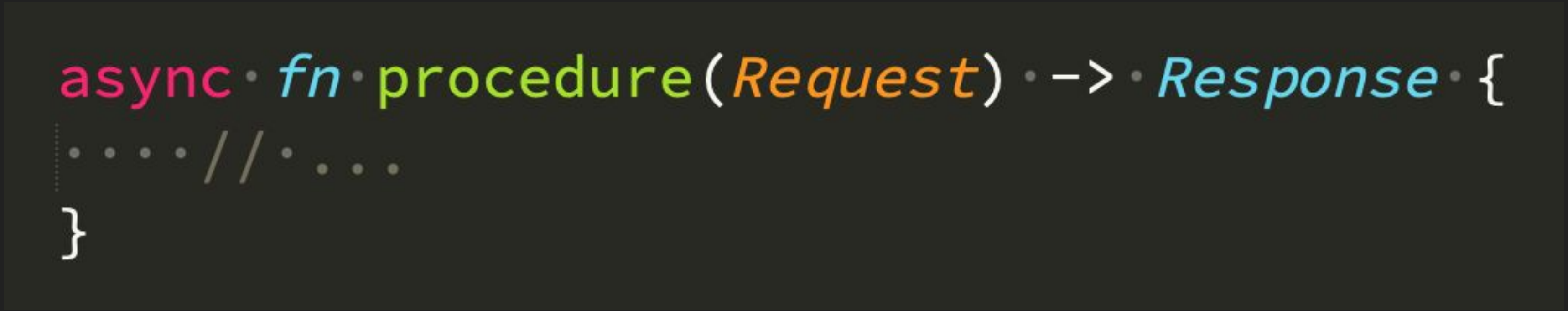
PC
(client)

request

response

MCU
(server)

2



```rust
async fn procedure(Request) -> Response {
    // ...
}
```

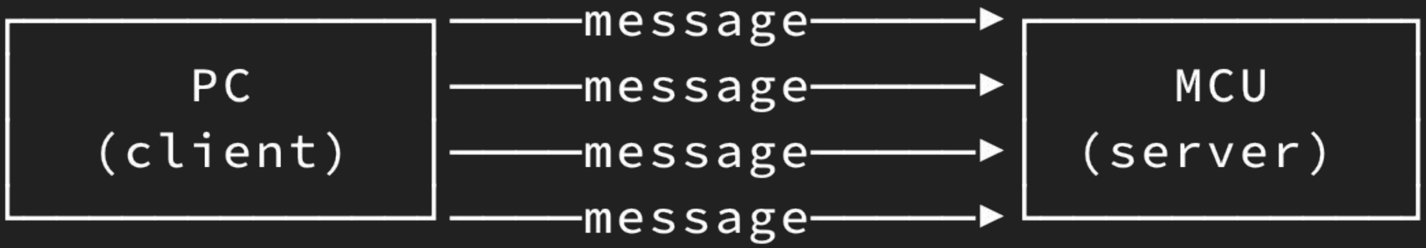
3

"topics", for streaming or notifications

3

can go in either direction, NO response

3



PC (client)

message

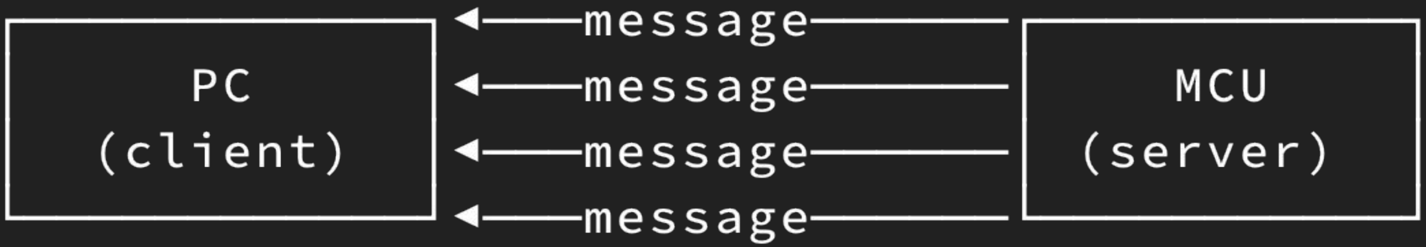
message

message

message

MCU (server)

and/or



PC (client)

message

message

message

message

MCU (server)

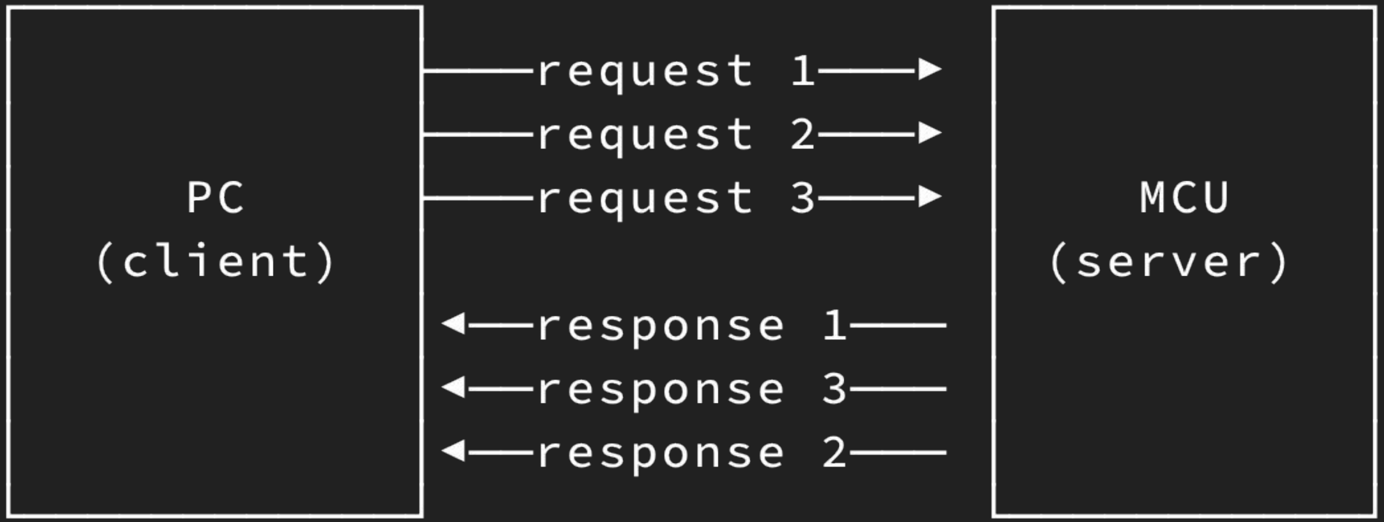
how does `postcard-rpc` do it?

every packet gets a **header** with two things:

1

a sequence number

1


PC
(client)
request 1
request 2
request 3
response 1
response 3
response 2
MCU
(server)

2

a unique Key

2

a way to identify what KIND of message this is

2

```
key = hash("path") + hash(schema(Message));
```

2

keys are 8 bytes (for now)

2

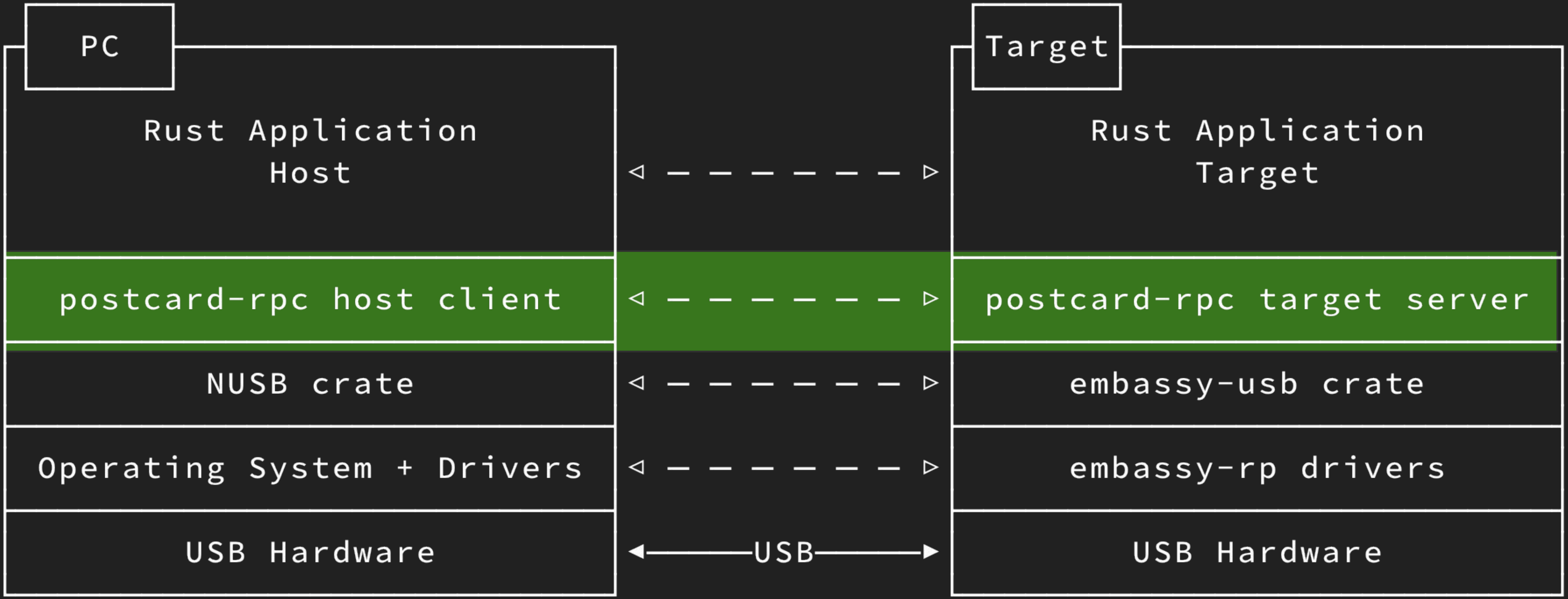
keys are generated at compile time

now that the protocol is standard, we can provide reusable protocol code

USB:

`nusb` on the host side

`embassy-usb` on the target side



PC

Rust Application
Host

postcard-rpc host client

NUSB crate

Operating System + Drivers

USB Hardware

Target

Rust Application
Target

postcard-rpc target server

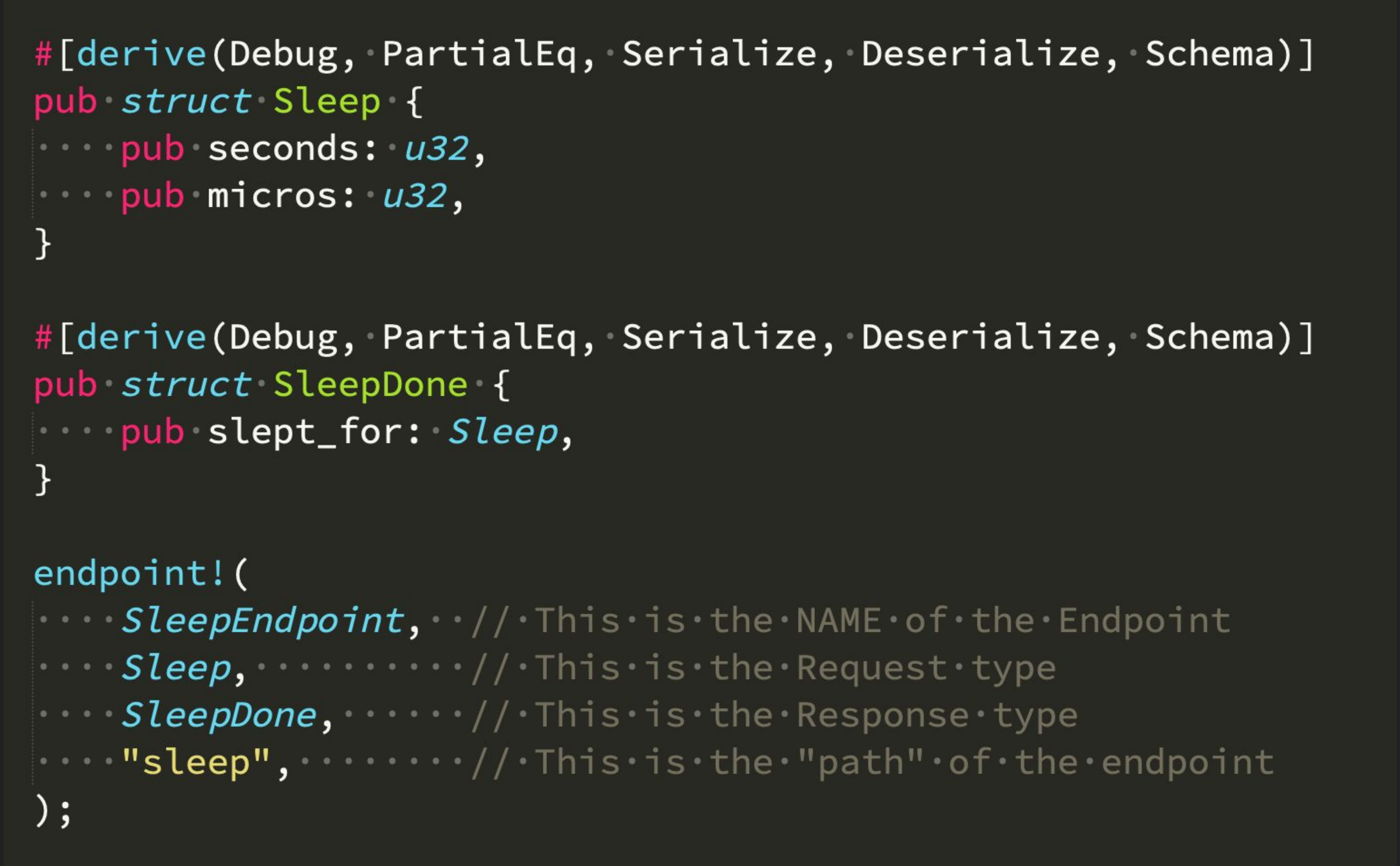
embassy-usb crate

embassy-rp drivers

USB Hardware

what does the code look like?

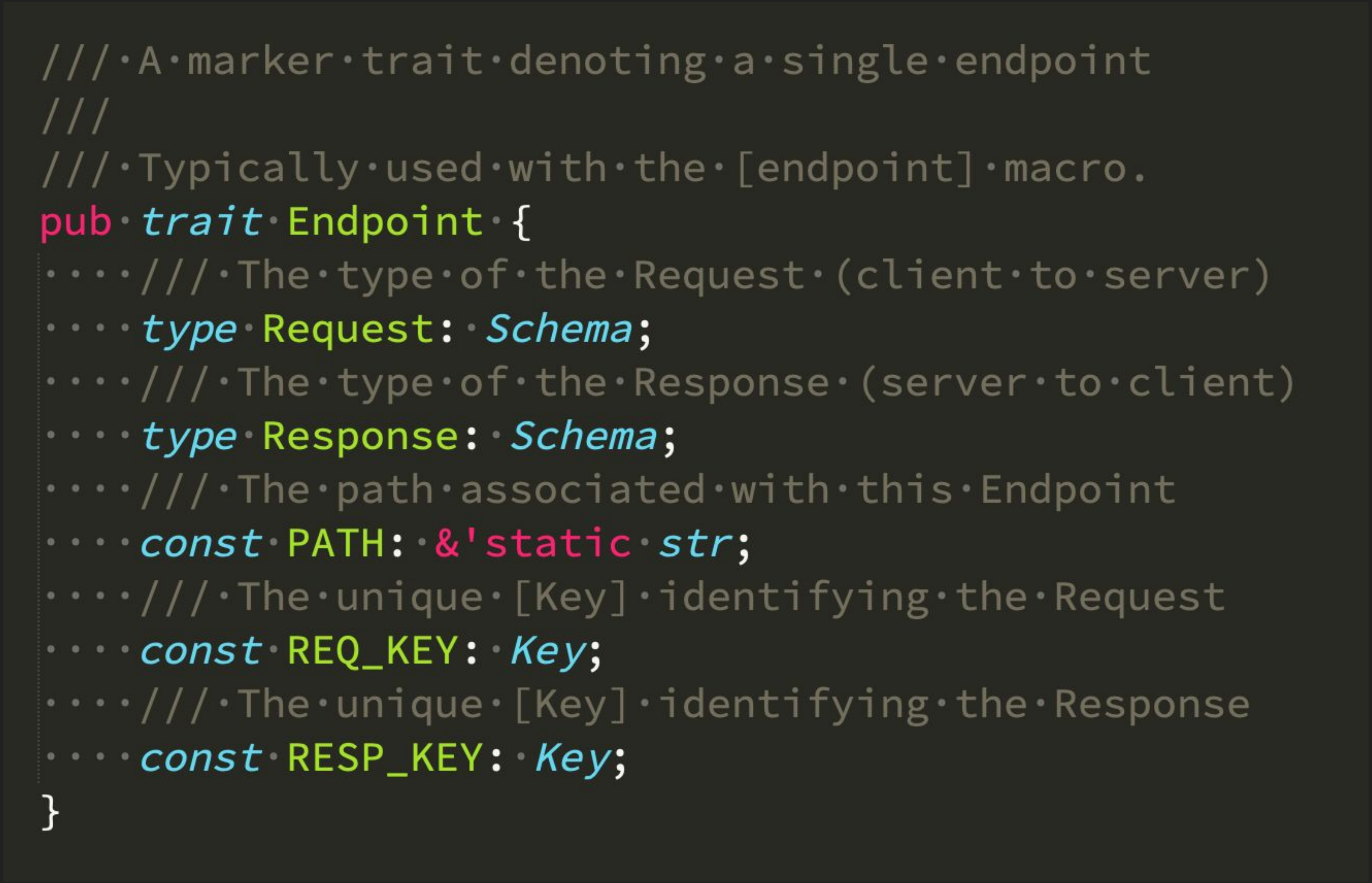
defining shared protocol definitions:



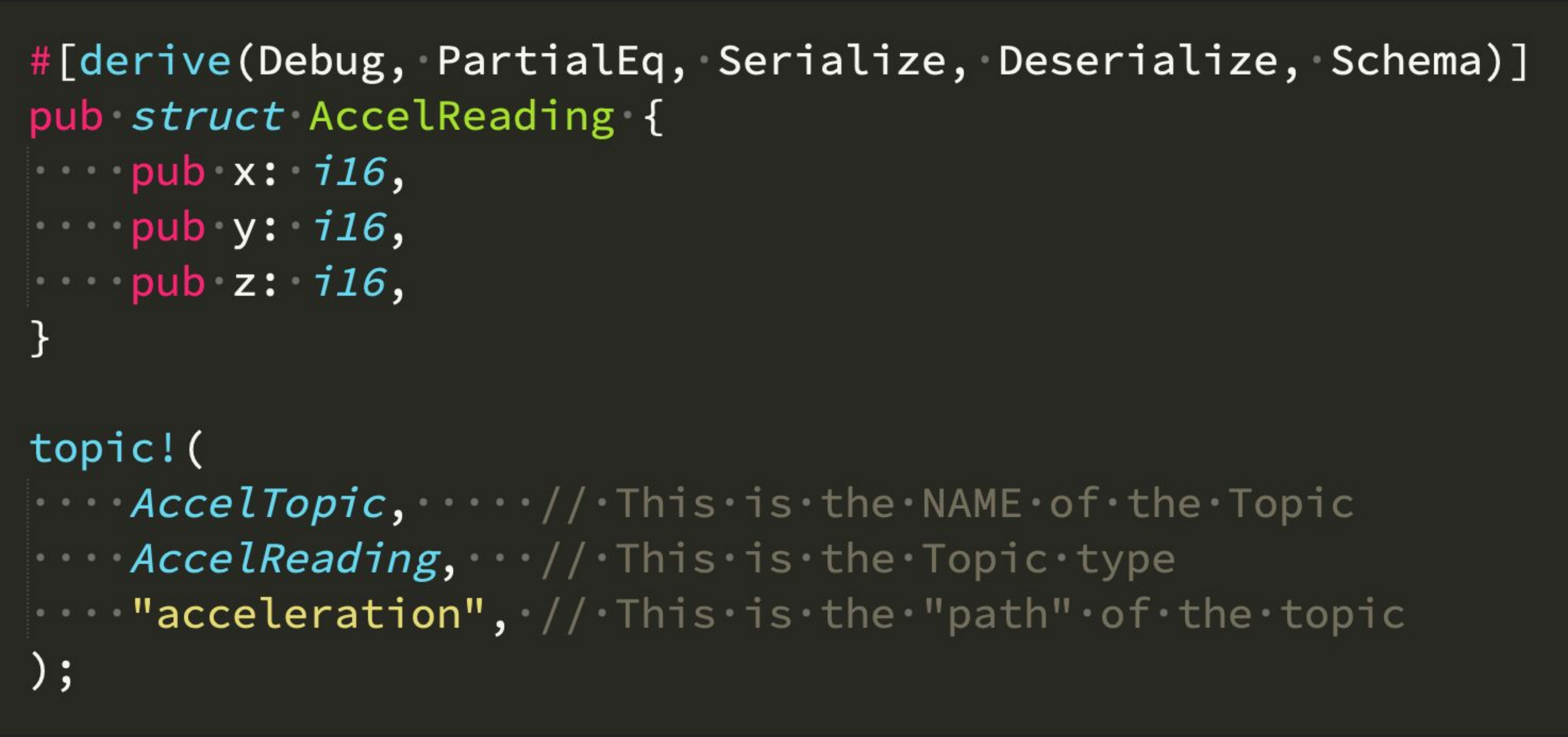
```rust
#[derive(Debug, PartialEq, Serialize, Deserialize, Schema)]
pub struct Sleep {
    pub seconds: u32,
    pub micros: u32,
}

#[derive(Debug, PartialEq, Serialize, Deserialize, Schema)]
pub struct SleepDone {
    pub slept_for: Sleep,
}

endpoint!(
    SleepEndpoint,  // This is the NAME of the Endpoint
    Sleep,          // This is the Request type
    SleepDone,      // This is the Response type
    "sleep",        // This is the "path" of the endpoint
);
```

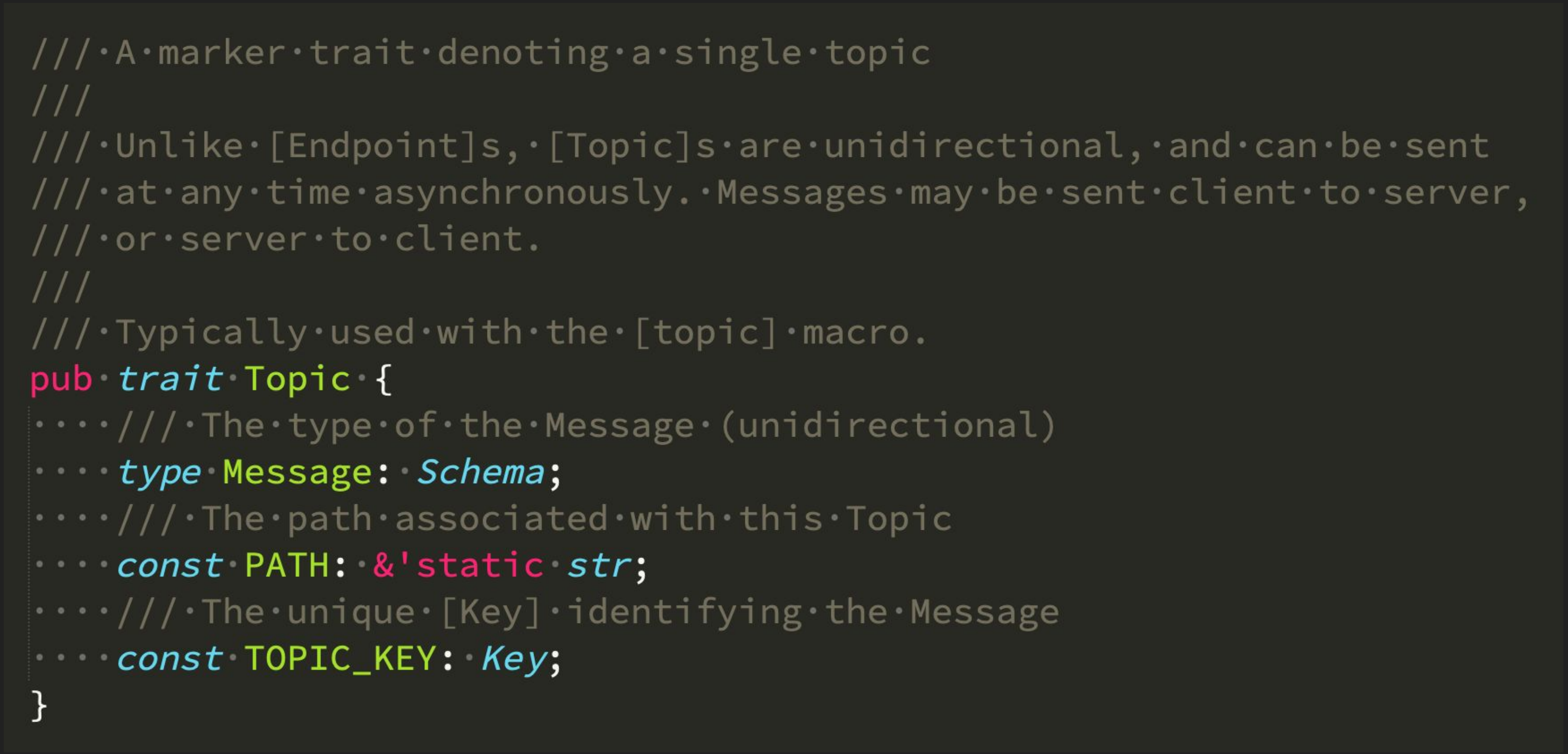


```rust
/// A marker trait denoting a single endpoint
///
/// Typically used with the [endpoint] macro.
pub trait Endpoint {
    /// The type of the Request (client to server)
    type Request: Schema;
    /// The type of the Response (server to client)
    type Response: Schema;
    /// The path associated with this Endpoint
    const PATH: &'static str;
    /// The unique [Key] identifying the Request
    const REQ_KEY: Key;
    /// The unique [Key] identifying the Response
    const RESP_KEY: Key;
}
```



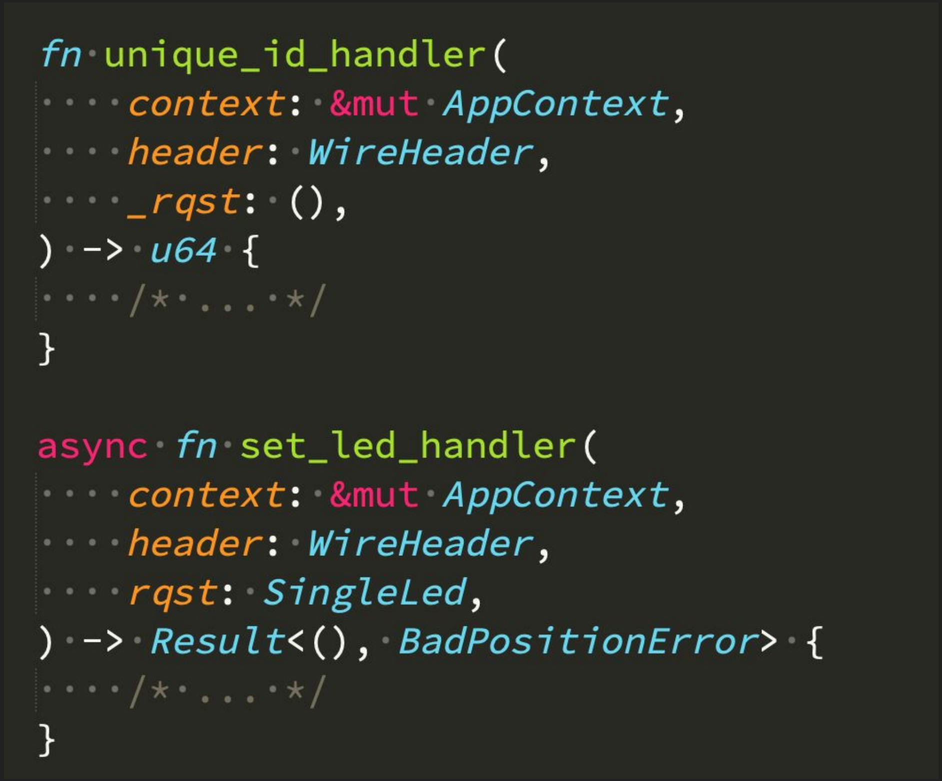
```rust
#[derive(Debug, PartialEq, Serialize, Deserialize, Schema)]
pub struct AccelReading {
    pub x: i16,
    pub y: i16,
    pub z: i16,
}

topic!(
    AccelTopic,     // This is the NAME of the Topic
    AccelReading,   // This is the Topic type
    "acceleration", // This is the "path" of the topic
);
```



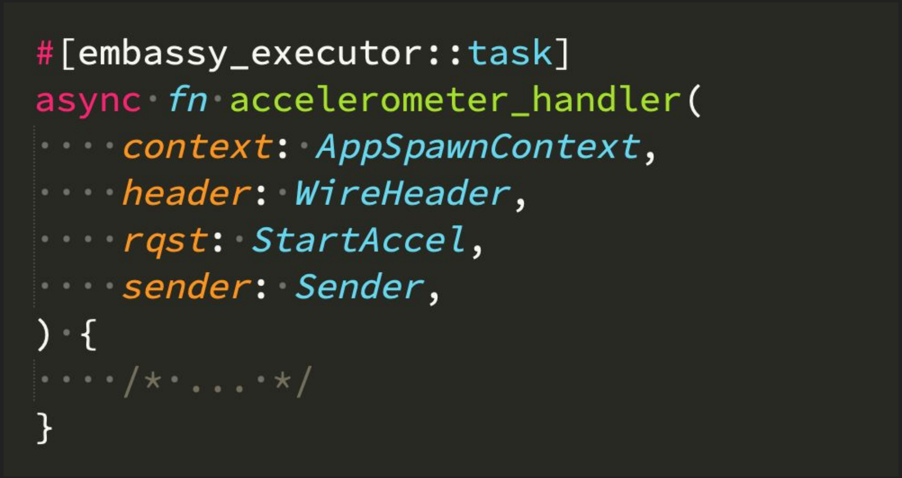
```rust
/// A marker trait denoting a single topic
///
/// Unlike [Endpoint]s, [Topic]s are unidirectional, and can be sent
/// at any time asynchronously. Messages may be sent client to server,
/// or server to client.
///
/// Typically used with the [topic] macro.
pub trait Topic {
    /// The type of the Message (unidirectional)
    type Message: Schema;
    /// The path associated with this Topic
    const PATH: &'static str;
    /// The unique [Key] identifying the Message
    const TOPIC_KEY: Key;
}
```

defining a server's endpoint handlers



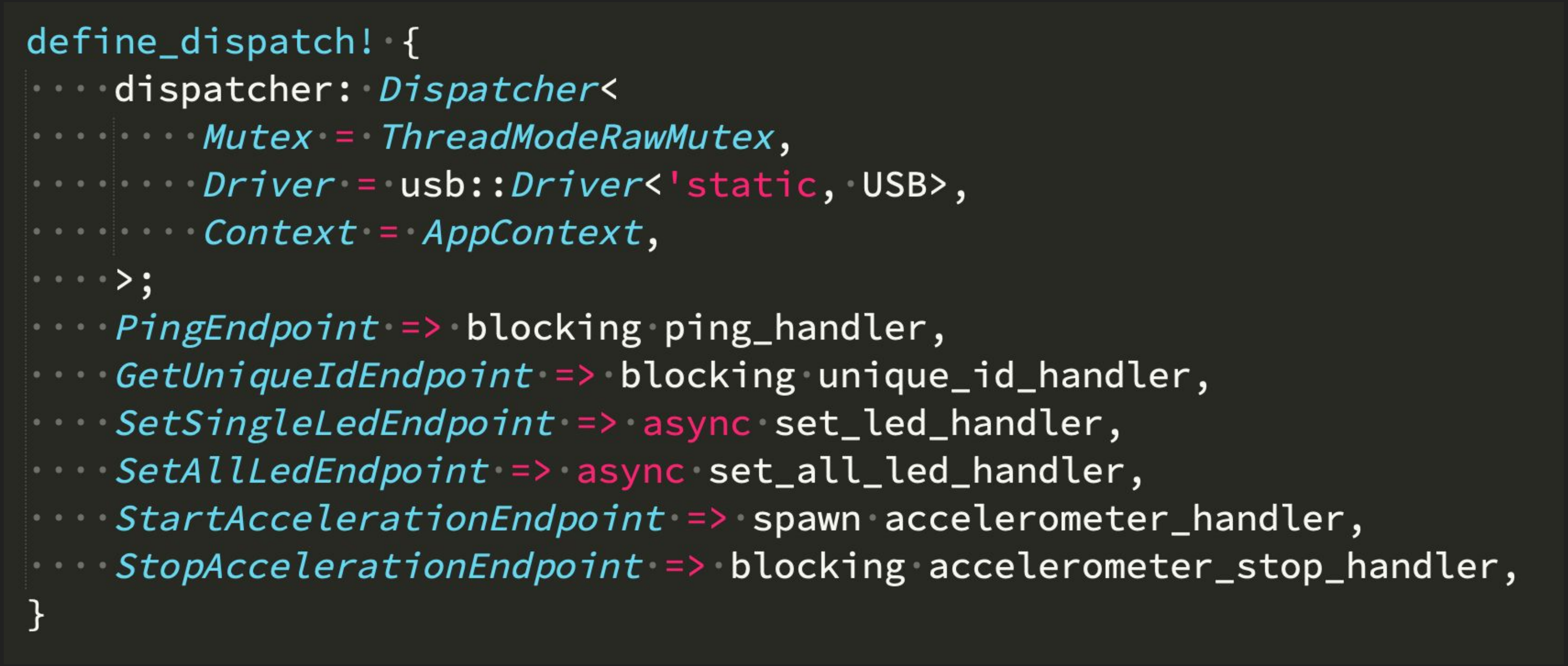
```rust
fn unique_id_handler(
    context: &mut AppContext,
    header: WireHeader,
    _rqst: (),
) -> u64 {
    /* ... */
}

async fn set_led_handler(
    context: &mut AppContext,
    header: WireHeader,
    rqst: SingleLed,
) -> Result<(), BadPositionError> {
    /* ... */
}
```



```rust
#[embassy_executor::task]
async fn accelerometer_handler(
    context: AppSpawnContext,
    header: WireHeader,
    rqst: StartAccel,
    sender: Sender,
) {
    /* ... */
}
```

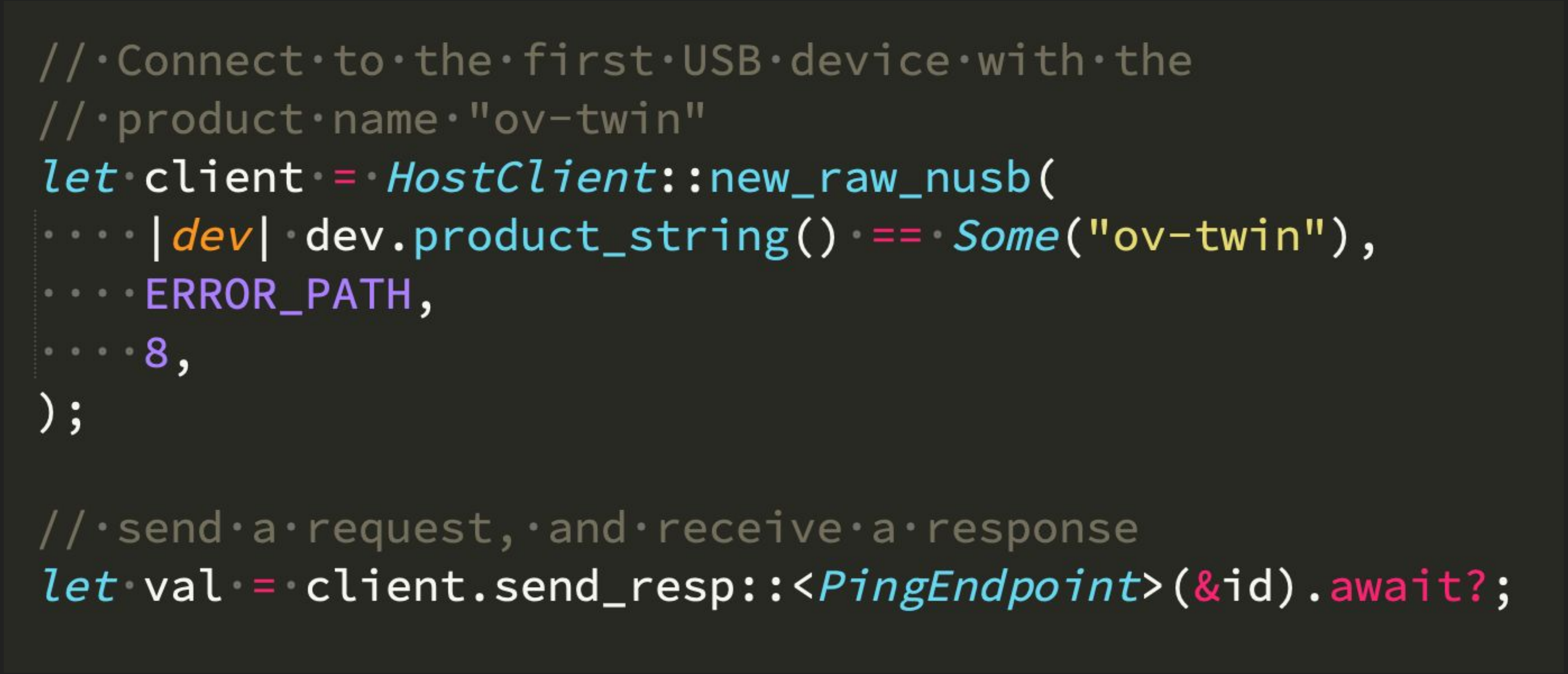
defining the “routing” or “dispatching” of our server



```
define_dispatch! {
    dispatcher: Dispatcher<
        Mutex = ThreadModeRawMutex,
        Driver = usb::Driver<'static, USB>,
        Context = AppContext,
    >;
    PingEndpoint => blocking ping_handler,
    GetUniqueIdEndpoint => blocking unique_id_handler,
    SetSingleLedEndpoint => async set_led_handler,
    SetAllLedEndpoint => async set_all_led_handler,
    StartAccelerationEndpoint => spawn accelerometer_handler,
    StopAccelerationEndpoint => blocking accelerometer_stop_handler,
}
```

on the PC side

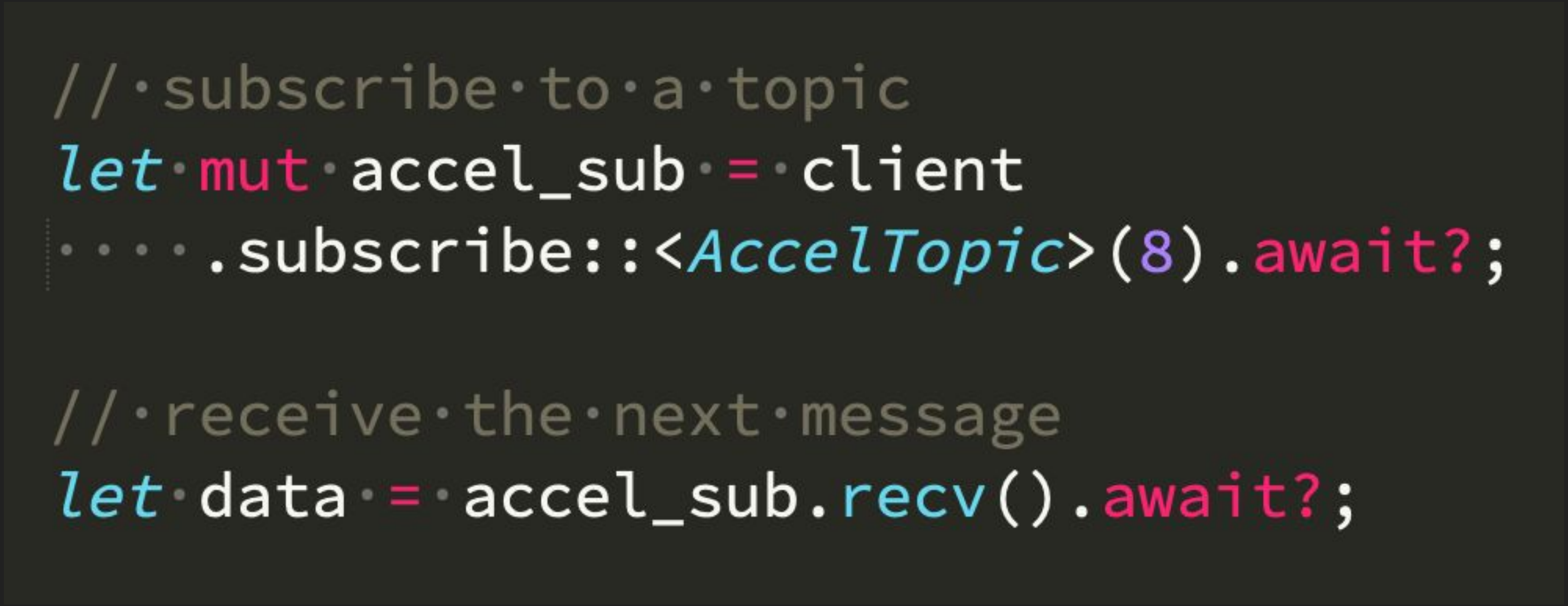
making an rpc/`Endpoint` request



```rust
// Connect to the first USB device with the
// product name "ov-twin"
let client = HostClient::new_raw_nusb(
    |dev| dev.product_string() == Some("ov-twin"),
    ERROR_PATH,
    8,
);

// send a request, and receive a response
let val = client.send_resp::<PingEndpoint>(&id).await?;
```

subscribing to incoming `Topic` messages



```rust
// subscribe to a topic
let mut accel_sub = client
    .subscribe::<AccelTopic>(8).await?;

// receive the next message
let data = accel_sub.recv().await?;
```

concerns: separated.

`postcard`

make getting data from here to there effortless

postcard-rpc

make a conversation between computers effortless

ONE VARIABLE

no matter how you connect to your devices they're never more than a postcard away.

RustNL

postcard

an unreasonably effective tool for machine to machine communication

🦋: @jamesmunns.com
🕸️: onevariable.com
💬: @jamesmunns:beeper.com
🐦: @bitshiftmask

ONE VARIABLE

no matter how you connect to your devices, they're never more than a `postcard` away.

onevariable.com
docs.rs/postcard
docs.rs/postcard-rpc

ONE VARIABLE

Q: where does the name "postcard" come from?
A: The song "Postcards From Hell" by "The Wood Brothers"

ONE VARIABLE