RUSTLAB 2026

FROM HARDWARE TO HUMAN INTERFACE: BUILDING HID DEVICES IN RUST

ONCE UPON A TIME...

GAMES HAD CUSTOM CONTROLLERS!

LIKE DUCK HUNT!

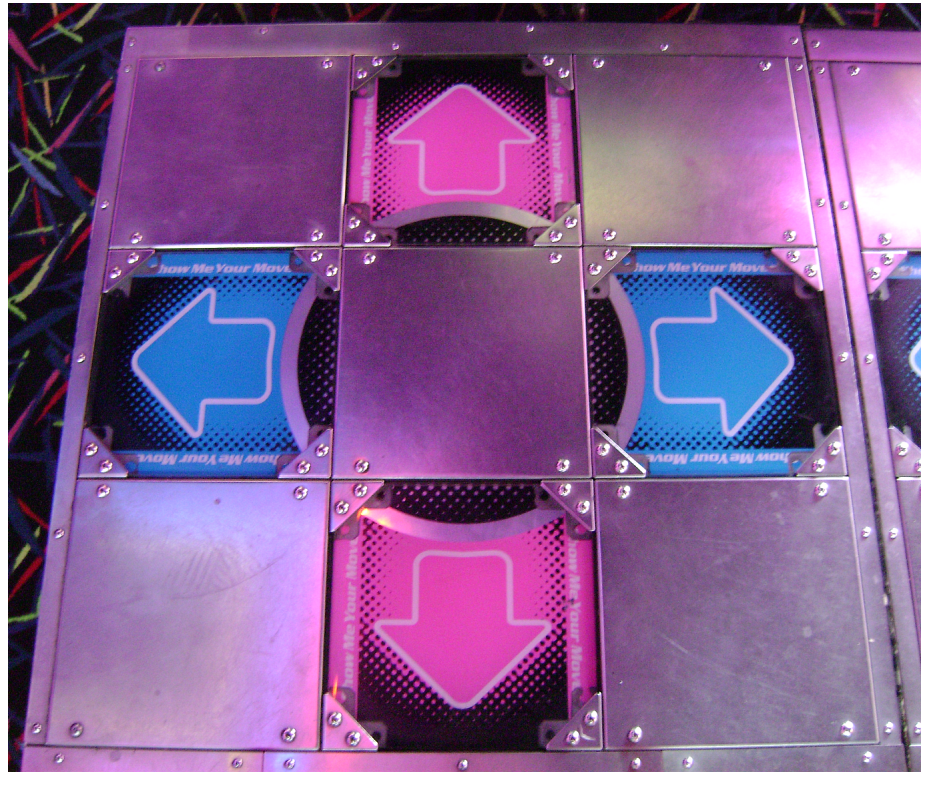


STEEL BATTALION


COMMUNICATION TUNER
TUNER DIAL
COMMUNICATION BUTTONS
FUNCTION SWITCHES
TANK DETACH
OVERRIDE MANIPULATOR
NIGHT SCOPE
LINE COLOR CHANGE
CENTER BLOCK
WASHING
EXTINGUISHER
CHAFF
WEAPON CONTROLS
MAIN
SUB
MAGAZINE CHANGE
ROTATION LEVER
TOGGLE SWITCHES
MAIN WEAPON
LOCK ON
AIMING LEVER
EJECT
MULTI MONITOR
MODE SELECT
SUB MONITOR MODE SELECT
MAIN MONITOR ZOOM
COCKPIT HATCH
IGNITION
START

MUSIC GAMES





CONTROLLER INNOVATION





INPUT DEVICES ARE 'DONE'

NO MEANINGFUL INNOVATION FOR DECADES:

- Keyboard
- Mouse
- Trackpad
- 'Xbox style' controller
- Even VR struggles

DIY FLIGHT / RACING SIM





VR

- Visual parts are useless
- Tactile even more useful
- ... but are you holding a VR controller?
- Hand tracking?

DARK CONTROLLERS

- Use in VR
- Fixed spot that matches VR
- Tactile feedback is important
- Authentikit

TODAY WE'LL MAKE A SMALL HAPTIC INPUT DEVICE

ABOUT ME

Frank Lyaruu



Architect / Software generalist / Tinkerer

Principal Innovator at Hightech Innovators

Definitely **not** an Embedded Engineer

HAPTIC FEEDBACK: SMART KNOB

- Rotary controller
- Useful feedback

BRUSHLESS ('BLDC') MOTOR



BRUSHLESS MOTOR AS INPUT DEVICE

- Out of the box a high-resolution jog wheel
- Sturdy and well built

HAPTIC FEEDBACK SUCH AS:

- Detents
- Multi rotation
- Spring loaded
- Barriers

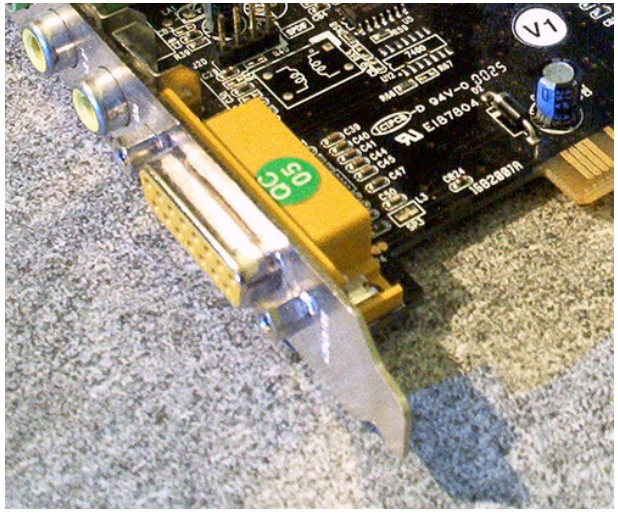
THREE PARTS:

- ~~Reading the Hall sensor (I2C connection)~~
- ~~Creating the force feedback (FOC motor driver)~~
- Writing the data to the host (USB HID)

ANOTHER (SMALL) HISTORY LESSON

BACK IN THE DAY, EVERY TYPE OF INPUT HAD DISTINCT CONNECTORS

Analog joystics



PS2 (Mouse / Keyboard)



Mouse / Serial



CC BY-SA 3.0

https://commons.wikimedia.org/w/index.php?curid=33732

ANALOG JOYSTICK

KIMSOFT INC.
2001

SERIAL PORTS

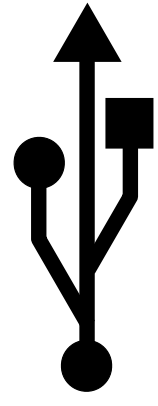
- Symmetrical
- Agnostic over wire format
- Unmanaged

SERIAL PORT CONFIGURATION

- Baud rate
- Data bits
- Stop bits
- Parity checks
- ~~Plug and Play~~

e.g. 9600 baud 8N1

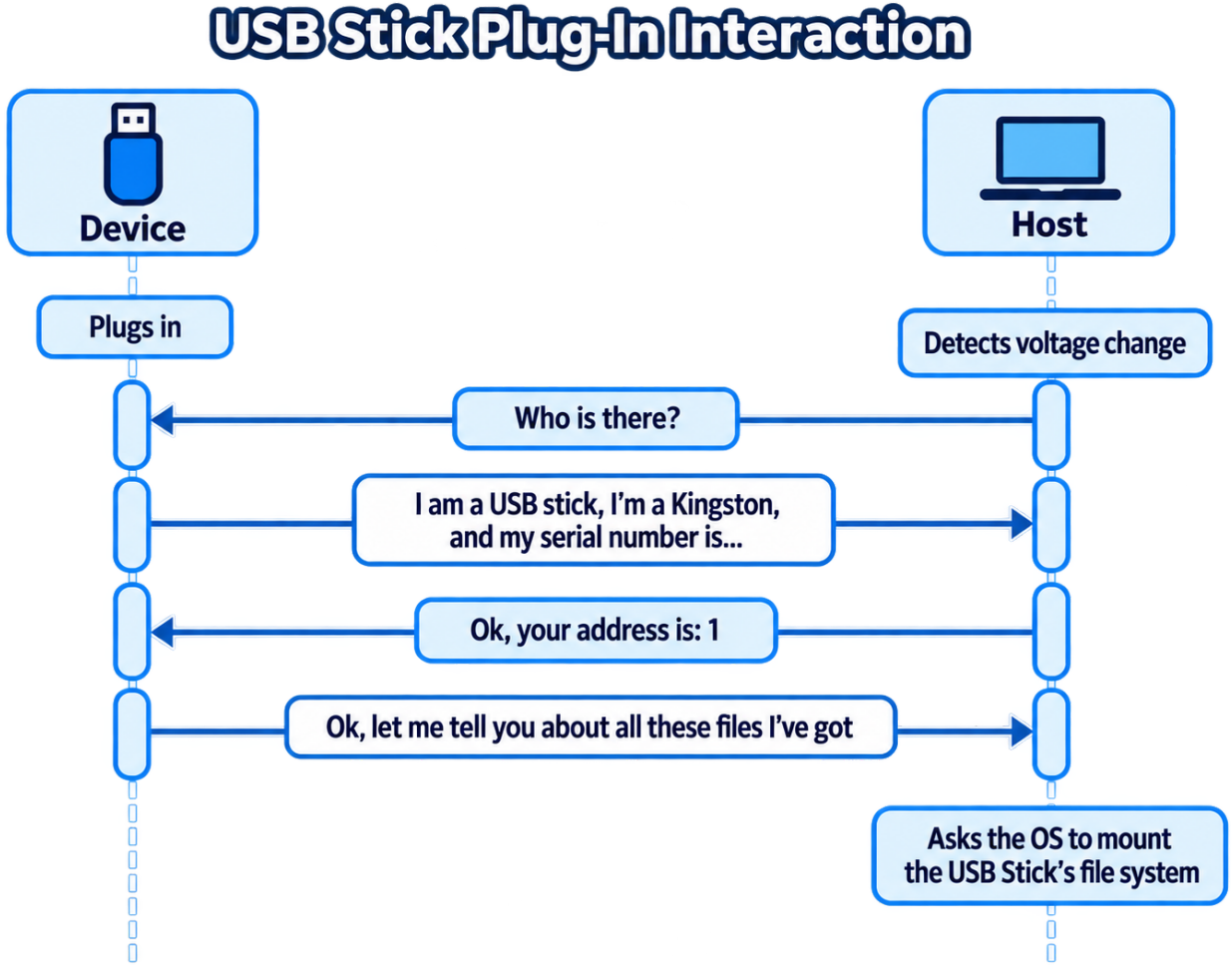
USB



USB

- Multiple devices on one host port using hubs
- Devices 'self describe'
- An extensive set of default device classes

SIMPLIFIED USB CONNECTION SEQUENCE



USB HID DEVICES

HID Usage Tables
FOR
Universal Serial Bus (USB)

Version 1.7

1996-2026 USB Implementers' Forum - All rights reserved.

USB HID DEVICES

- Write description
- Respond with reports that match the description

USB HID DESCRIPTOR

```
0x05, 0x01,         // Usage Page (Generic Desktop Ctrls)
0x09, 0x04,         // Usage (Joystick)
0xA1, 0x01,         // Collection (Application)
0xA1, 0x00,         //   Collection (Physical)
0x09, 0x30,         //     Usage (X)
0x09, 0x31,         //     Usage (Y)
0x15, 0x00,         //     Logical Minimum (0)
0x26, 0xFF, 0x07,   //     Logical Maximum (2047)
0x35, 0x00,         //     Physical Minimum (0)
0x46, 0xFF, 0x00,   //     Physical Maximum (255)
0x75, 0x10,         //     Report Size (16)
0x95, 0x02,         //     Report Count (2)
0x81, 0x02,         //     Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
0xC0,               //   End Collection
0x75, 0x08,         //   Report Size (8)
0x95, 0x03,         //   Report Count (3)
0x81, 0x03,         //   Input (Cnst,Var,Abs)
0xC0,               // End Collection
```

USB CDC CLASS

- Closest to original serial port
- Interact using normal terminal tools
- COM port on windows
- /dev/cu.* on other platforms

USB HID DESCRIPTOR IN RUST

```
#[gen_hid_descriptor(
    (collection = APPLICATION, usage_page = GENERIC_DESKTOP, usage = JOYSTICK) = {
        (collection = APPLICATION, usage = POINTER) = {
            (usage = X,) = {
                #[item_settings(data,variable,absolute)] x=input;
            };
        };
        (usage_page = BUTTON, usage_min = BUTTON_1, usage_max = BUTTON_8) = {
            #[packed_bits = 8] #[item_settings(data,variable,absolute)] buttons=input;
        }
    }
)]
#[allow(dead_code)]
pub struct JoystickReport {
    pub x: i16,
    pub buttons: u8,
}
```

CONFIGURING USB IN ESP-HAL

```
let usb = Usb::new(peripherals.USB0, peripherals.GPIO20, peripherals.GPIO19);
// Create the driver, from the HAL.
let ep_out_buffer = EP_OUT_BUFFER.init([0u8; 1024]);
let config = esp_hal::otg_fs::asynch::Config::default();

let driver = Driver::new(usb, ep_out_buffer, config);

let mut usb_config = embassy_usb::Config::new(0xa0de, 0xdafe);
usb_config.manufacturer = Some("Frank's USB shop");
usb_config.product = Some("Frank's Joystick");
usb_config.serial_number = Some("12345678");
usb_config.max_power = 100;
usb_config.max_packet_size_0 = 64;
```

CONFIGURING USB IN ESP-HAL (2)

```
// Bunch of buffers
let config_descriptor = CONFIG_DESCRIPTOR.init([0u8; 256]);
let bos_descriptor = BOS_DESCRIPTOR.init([0u8; 256]);
let msos_descriptor = MSOS_DESCRIPTOR.init([0u8; 256]);
let control_buf = CONTROL_BUF.init([0u8; 64]);

let mut usb_builder = embassy_usb::Builder::new(
    driver,
    usb_config,
    config_descriptor,
    bos_descriptor,
    msos_descriptor,
    control_buf,
);
```

CONFIGURING USB + HID IN ESP-HAL

```rust
// Create classes on the builder.
let hid_config = embassy_usb::class::hid::Config {
    report_descriptor: JoystickReport::desc(),
    request_handler: None,
    poll_ms: 60,
    max_packet_size: 64,
    hid_subclass: HidSubclass::Boot,
    hid_boot_protocol: HidBootProtocol::Mouse,
};

let hid_state = HID_STATE.init(State::new());
let writer: HidWriter<'_, Driver<'_>, 8> =
    HidWriter::new(&mut usb_builder, hid_state, hid_config);
```

CONFIGURING USB + HID IN ESP-HAL (2)

```rust
#[task]
pub async fn usb_writer(
    mut writer: HidWriter<'static, Driver<'static>, 8>,
    mut receiver: Receiver<'static, CriticalSectionRawMutex, JoystickReport, 3>,
) -> ! {
    let mut ticker = Ticker::every(Duration::from_millis(5));

    loop {
        if let Some(value) = receiver.try_get() {
            match writer.write_serialize(&value).await {
                Ok(_) => {}
                Err(_e) => {
                    warn!("Write failed: {:?}",e);
                }
            }
        }
        ticker.next().await;
    }
}
```

CONFIGURING USB + CDC IN ESP-HAL

```
let cdc_state = CDC_STATE.init(CdcState::new());
let cdc = CdcAcmClass::new(&mut usb_builder, cdc_state, 64);
let (cdc_sender, _cdc_receiver) = cdc.split();

// Done building, initialize usb object
let usb = usb_builder.build();
```

ELSEWHERE, RUN THIS USB STACK AS A TASK

```
#[task]
async fn run_usb(mut usb: UsbDevice<'static, Driver<'static>>) -> ! {
    usb.run().await
}
```

SENDING TELEMETRY OVER CDC

```
#[derive(Clone, Copy, Serialize, Deserialize, Debug, PartialEq)]
pub struct TelemetryMessage {
    shaft_position: ShaftPosition,
    torque: I16F16,
}
```

SENDING TELEMETRY OVER CDC (2)

```rust
#[task]
pub async fn cdc_logger(
    mut sender: cdc_acm::Sender<'static, Driver<'static>>,
    mut telemetry_receiver: Receiver<'static, CriticalSectionRawMutex, TelemetryMessage, 3>,
) -> ! {
    let mut frame = [0u8; 128];

    loop {
        sender.wait_connection().await;

        loop {
            let msg = telemetry_receiver.changed().await;
            let encoded = match postcard::to_slice_cobs(&msg, &mut frame) {
                Ok(encoded) => encoded,
                Err(_) => continue,
            };

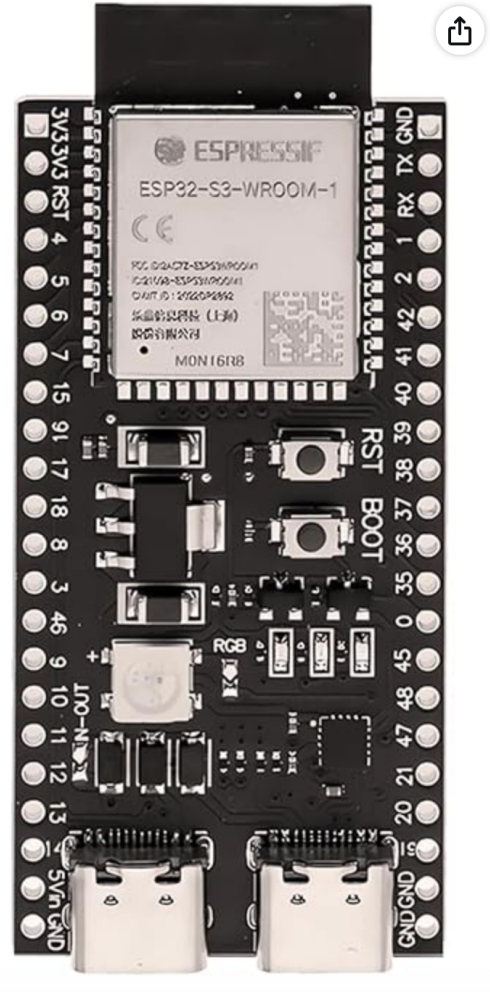
            if sender.write_packet(encoded).await.is_err() {
                break;
            }
        }
    }
}
```

BUILDING OUR DEVICE

SHOPPING LIST

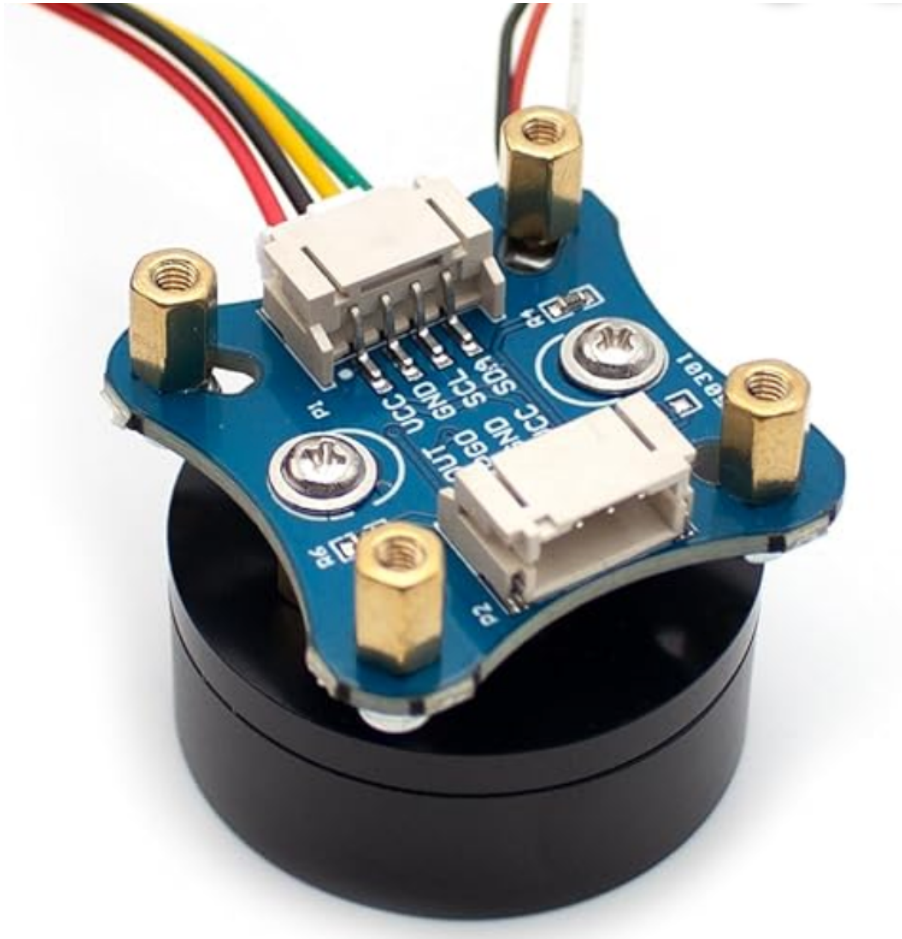
- Microcontroller
- Brushless motor
- Shaft hall sensor
- Motor driver
- Power supply

ESP32-S3

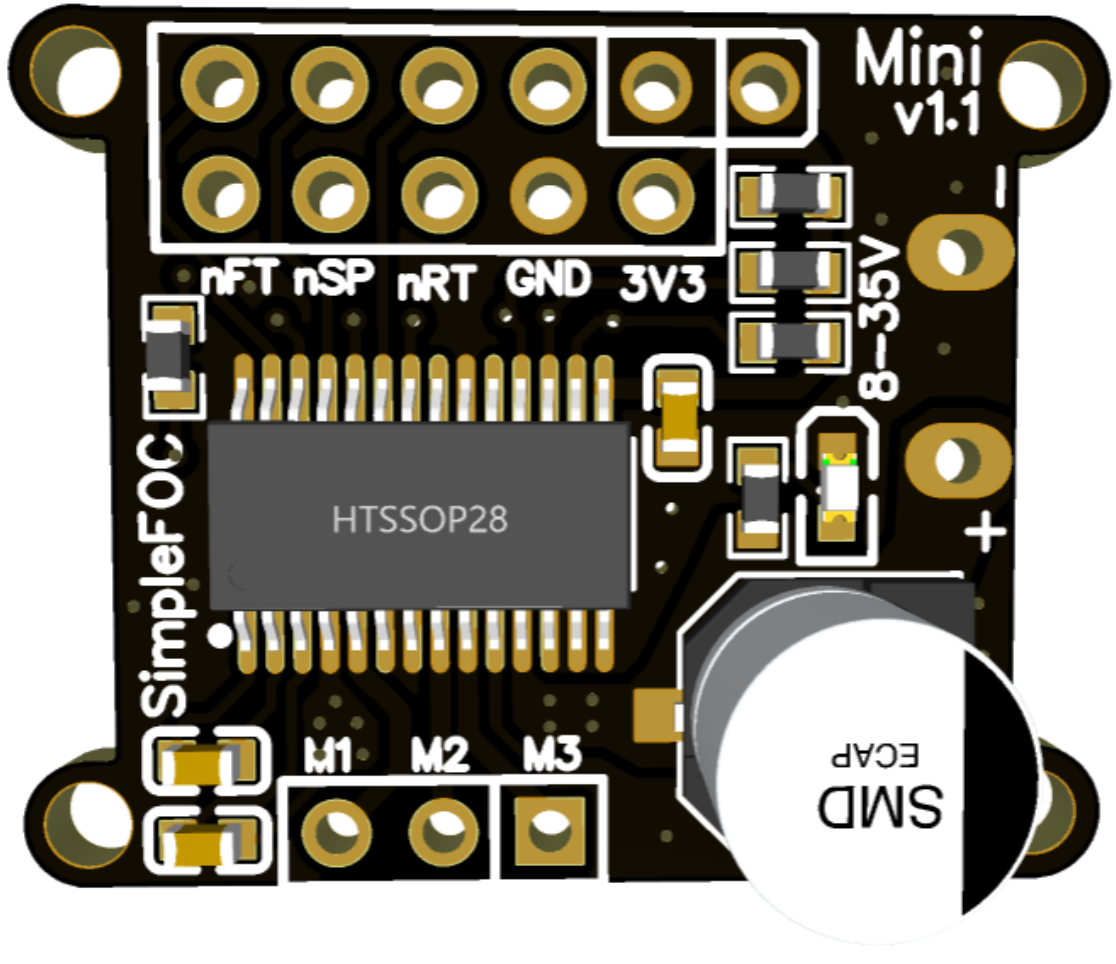

ESPRESSIF
ESP32-S3-WROOM-1
M0N16R8
RST
BOOT
RGB
3V3
3V3
RST
4
5
6
7
15
9
17
18
8
3
46
9
10
11
12
13
14
5V
GND
GND
GND
TX
GND
RX
1
2
42
41
40
39
38
37
36
35
0
45
48
47
21
20
19

BLDC + SENSOR



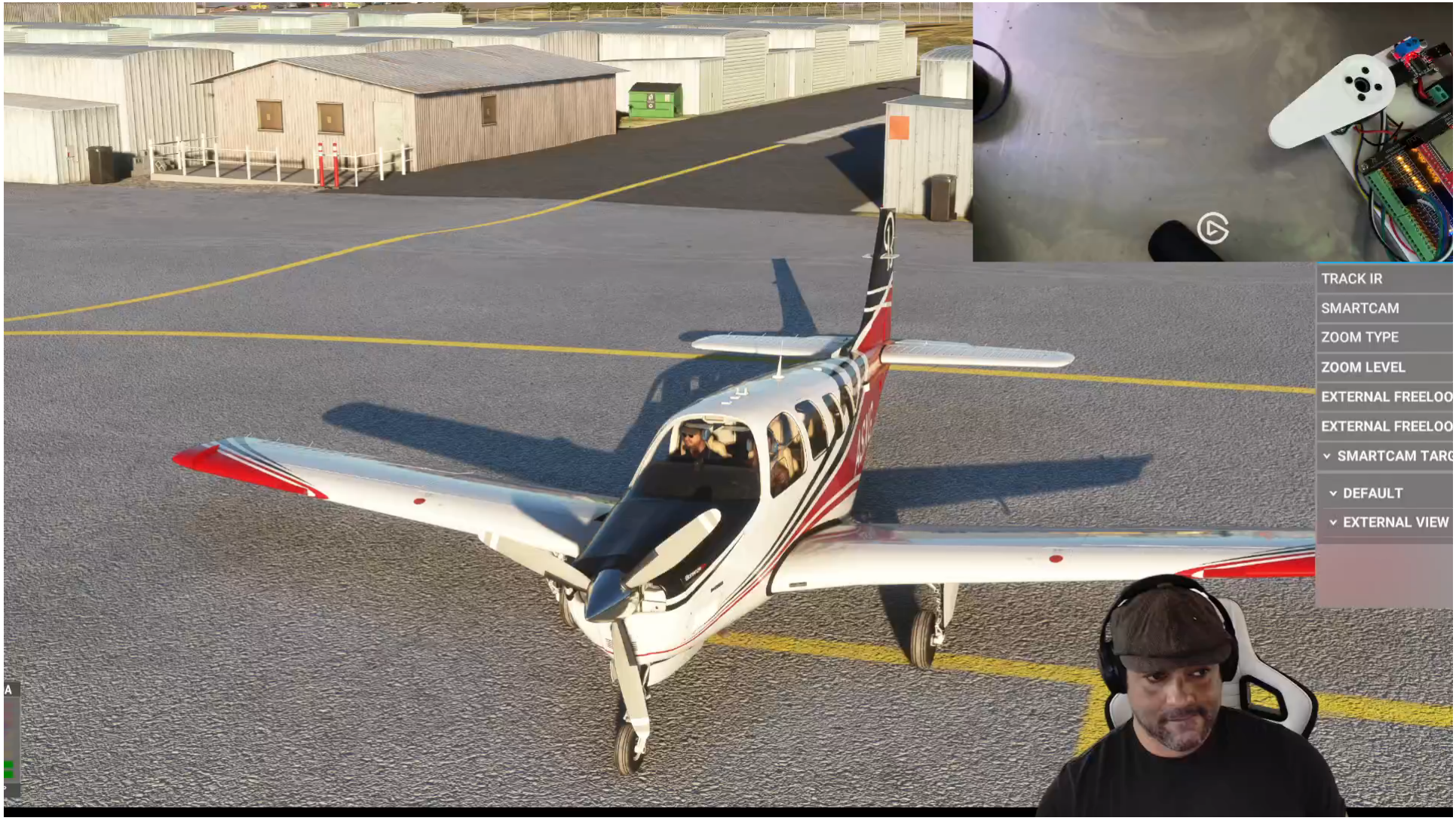


MOTOR CONTROLLER



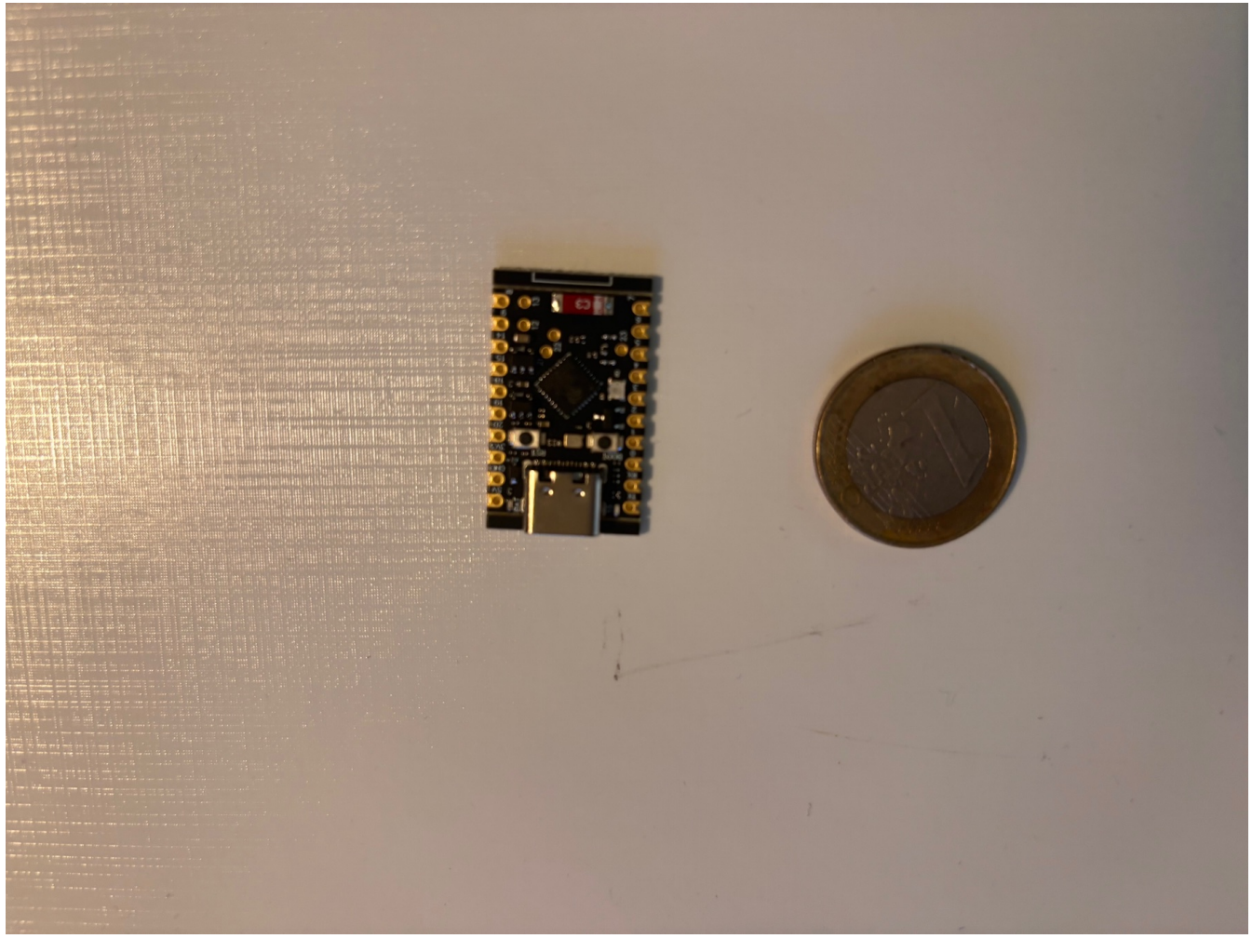
HARDWARE DEMO

FROM A SIMULATOR



THERE IS MORE!

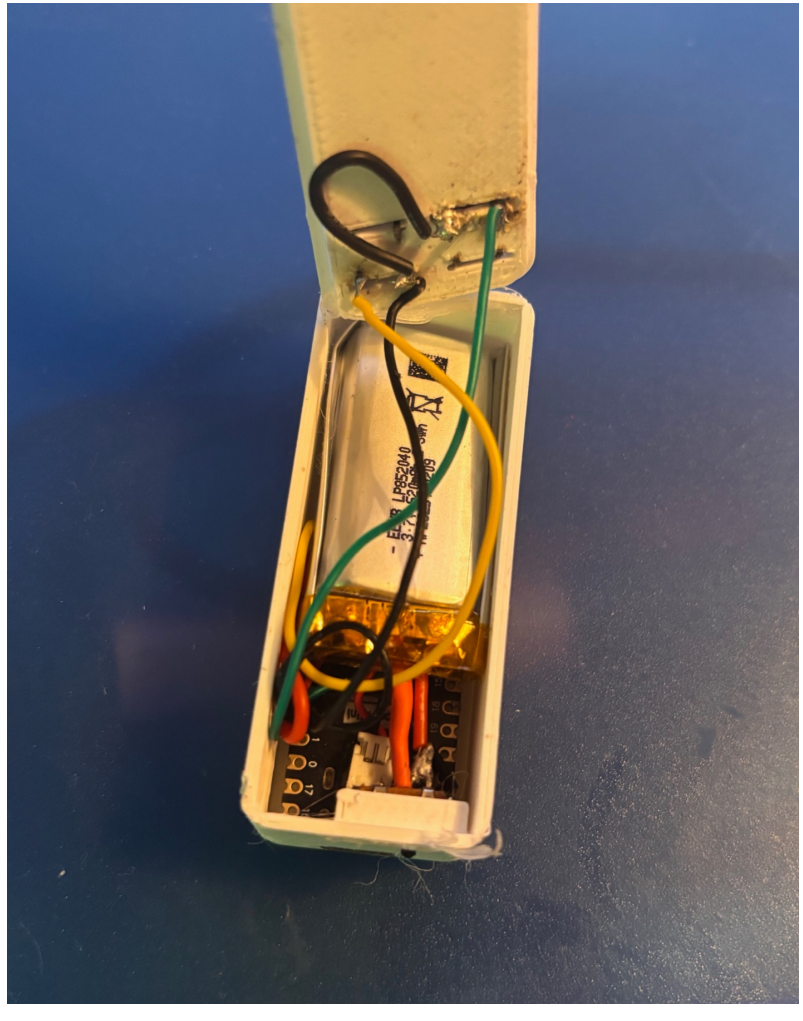
ESP32 C6 SUPERMINI



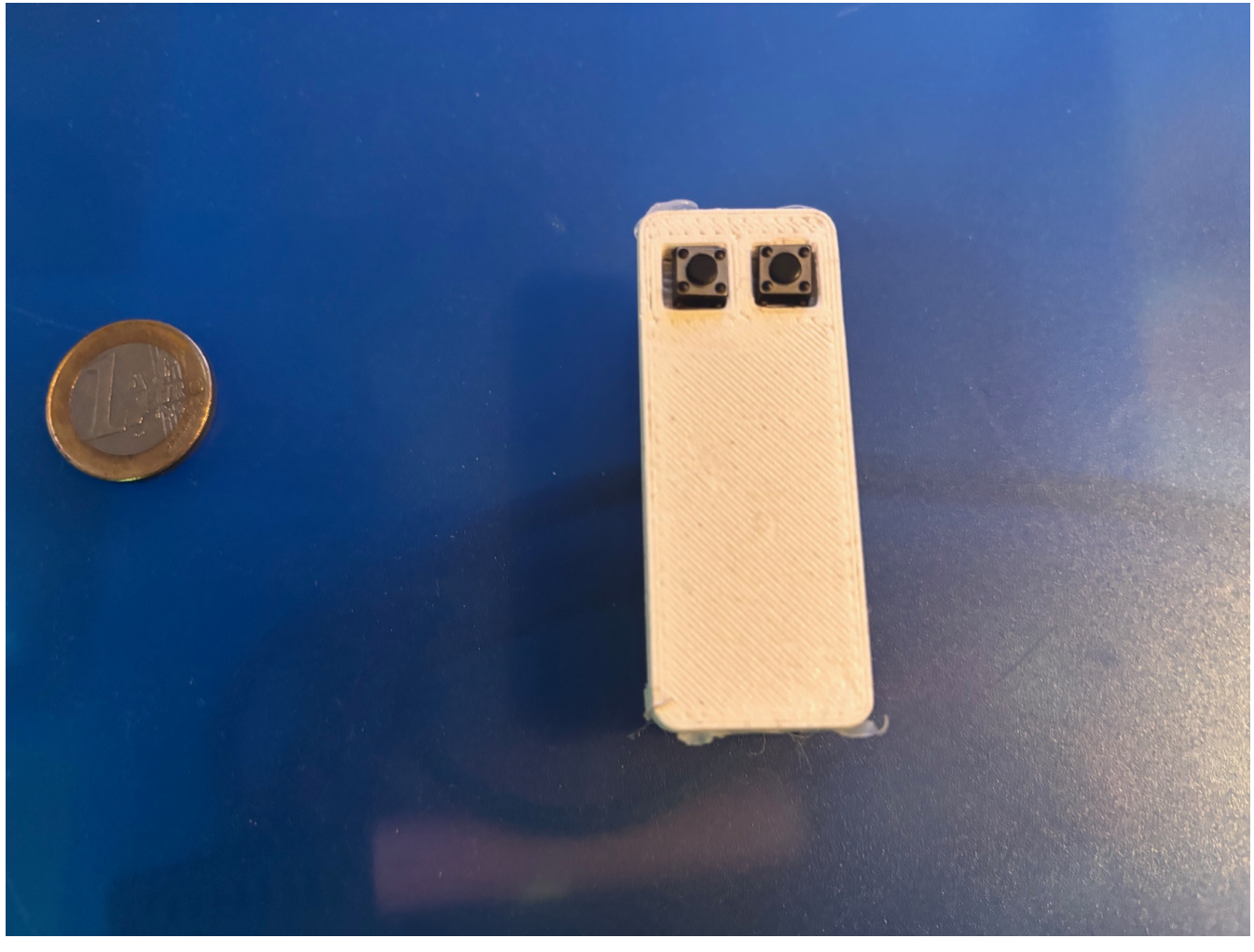
CLICKER BUILD



CLICKER BUILD (2)



CLICKER BUILD (3)



BLUETOOTH CLICKER EXAMPLE

```rust
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
pub enum ClickerEvents {
    Forward,
    Back,
}

pub const CLICKER_CHANNEL_SIZE: usize = 4;

pub type ClickerSender = Sender<'static, CriticalSectionRawMutex, ClickerEvents, CLICKER_CHANNEL_SIZE>;
pub type ClickerReceiver = embassy_sync::channel::Receiver<
    'static,
    CriticalSectionRawMutex,
    ClickerEvents,
    CLICKER_CHANNEL_SIZE,
>;
```

BLUETOOTH CLICKER EXAMPLE (2)

```
#[task]
pub async fn button_task(
    mut forward: Input<'static>,
    mut back: Input<'static>,
    events: ClickerSender,
) {
    loop {
        match select(
            back.wait_for_falling_edge(),
            forward.wait_for_falling_edge(),
        )
        .await
        {
            Either::First(()) => {
                events.send(ClickerEvents::Forward).await;
                ClickerEvents::Forward
            },
            Either::Second(()) => {
                events.send(ClickerEvents::Back).await;
                ClickerEvents::Back
            },
        };
        Timer::after_millis(300).await;
    }
}
```

BLUETOOTH CLICKER EXAMPLE (3)

```rust
async fn custom_task<p: packetpool="">(
    server: &Server<'_>,
    conn: &GattConnection<'_, '_, P>,
    link_encrypted: &AtomicBool,
    clicker_events: ClickerReceiver,
) {
    // Do not try to generate reports until the encrypted GATT link is ready.
    // The delay also gives the host time to discover the HID service and enable
    // notifications.
    if !link_encrypted.load(Ordering::Relaxed) {
        info!("[custom_task] waiting for encrypted link");
        while !link_encrypted.load(Ordering::Relaxed) {
            Timer::after_millis(250).await;
        }
    }
    loop {
        let keycode = match clicker_events.receive().await {
            ClickerEvents::Forward => 79,
            ClickerEvents::Back => 80,
        };

        if send_keyboard_key(server, conn, keycode).await.is_err() {
            info!("[custom_task] error sending keyboard key report");
            break;
        }
    }
```

DLC: DCS BIOS

```
fn main() {
    let mut latest_value = None;

    let mut builder = Av8bnaBuilder::<2, 256>::new();
    builder
        .register_jett_mode(|value: JettModeValue| {
            println!("Jettison switch position changed: {:?}", value);
            latest_value = Some(value);
            // TODO: Send command to move the motor to the right detent
        })
        .unwrap();

    let mut av8bna = builder.build();
    let socket = UdpSocket::bind(`"0.0.0.0:5010").expect("couldn't bind to address");
    let mut buf = [0; 256];
    loop {
        let (number_of_bytes, _src_addr) = socket.recv_from(&mut buf).expect("Didn't receive data");
        let filled_buf = &buf[..number_of_bytes];
        av8bna.feed(filled_buf).expect("invalid DCS-BIOS data");
    }
}
`
```

DLC: WAR STORIES

- Slightly broken hall sensor 🤢
- Bluetooth keyboard Mac 🤯
- FOC algorithm 🤓
- PID control loop 😬

ANY QUESTIONS?



