

17-350 / 17-650

Lecture 13: Drivers, `embedded-hal`, and DMA

Jonathan Aldrich

Oct 20, 2026

Embedded Systems in Rust

Date	Lecture	HW5
Oct 8 (remote)	Bare-metal Rust, hardware abstraction, typestate	Part 1 foundations
Oct 20 (today)	Drivers, <code>embedded-hal</code> , DMA, live on the board	HW5 released; Parts 0–1
Oct 22	Interrupts and concurrency on bare metal	Part 2

- **You have a micro:bit V2.** Sign it out with a TA, and keep it until your checkoff
- HW4 is due today. HW5 is due **Nov 3**, with a live checkoff Nov 3–6

Lecture Outline

- HW5: a tour
- Hardware day: blinky three ways
- How nRF peripherals talk
- Registers only the owner can reach
- Portable drivers: `embedded-hal`
- DMA: a second agent writing your memory
- From the toy to HW5 Part 1

HW5: A Safety Controller for a 3D-Printer Hotend

- The **heater** is the LED matrix (and pin P0.02); the chip's **temperature sensor** stands in for the thermistor
- It **trips** on an E-stop button, on over-temperature (with hysteresis), and when the sensor stops answering. A **watchdog** catches a hung main loop
- **The parts:**
 - Part 0: set up the board (by Thursday)
 - Part 1: the HAL, from raw registers (today's lecture)
 - Part 2a: the safety state machine (pure Rust, no hardware)
 - Part 2b: the device, with interrupts (Thursday's lecture)
 - Part 3: measurements and a write-up
- **Checkoff, Nov 3–6:** 8 minutes with a TA, on your board. Return the board then

Your Code Runs Twice

On the **board**, `raw_read` / `raw_write` are volatile accesses. On your **laptop**, the same driver code talks to a simulated nRF52833:

```
#[test]
fn output_configures_only_its_own_pin() {
    let p = Peripherals::take_for_test();
    let parts = Parts::new(p.p0, p.p1);
    let _heater = parts.p0_02.into_output();
    assert_eq!(sim::pin_cnf(0, 2), gpio::DIR_OUTPUT | gpio::INPUT_DISCONNECT);
    assert_eq!(pin_cnf_writes().len(), 1, "touched another pin's PIN_CNF");
    assert!(sim::output_high(0, 2), "into_output starts low");
}
```

- `cargo test` is most of your grade: provided tests now, hidden ones at grading
- `unsafe` lives only in `mmio.rs` and `dma.rs`, enforced. Clippy requires a `// SAFETY:` comment on every `unsafe` block, and denies `unwrap` and indexing

Two Ways to See Output

	RTT	The UART log
What	Debug prints (<code>rprintln!</code>) through the debug probe	The device's log, and the commands you type
Where	The terminal running <code>cargo run</code> (probe-rs)	A serial terminal: PuTTY, <code>screen</code> , <code>picocom</code> , at 115200
HW5 uses it for	Your own debugging	The checkoff

- UART stands for Universal Asynchronous Receiver-Transmitter
- Windows: the port is "USB Serial Device (COM n)" in Device Manager
- **Open the serial terminal first, then flash.** Lines sent while it's closed are lost
- When the board resets under `cargo run`, probe-rs exits with an error. The board is fine

Lecture Outline

- HW5: a tour
- **Hardware day: blinky three ways**
 - raw registers, a PAC, a BSP
 - what each costs
 - probe-rs, RTT, and a HardFault
- How nRF peripherals talk
- Registers only the owner can reach
- Portable drivers: `embedded-hal`
- DMA: a second agent writing your memory
- From the toy to HW5 Part 1

Blinky, Raw

```
const P0: usize = 0x5000_0000;
const OUTSET: *mut u32 = (P0 + 0x508) as *mut u32; // write 1s: drive those pins high
const OUTCLR: *mut u32 = (P0 + 0x50C) as *mut u32; // write 1s: drive those pins low
const ROW1: u32 = 21;    const COL1: u32 = 28;

// SAFETY: P0's OUTSET, OUTCLR, and PIN_CNF registers; nothing else uses port 0.
unsafe { // levels first, then directions: each pin comes up at the right level
    write_volatile(OUTCLR, (1 << ROW1) | (1 << COL1)); // LED off; column active-low
    write_volatile(pin_cnf(ROW1), DIR_OUTPUT);
    write_volatile(pin_cnf(COL1), DIR_OUTPUT);
}
loop {
    unsafe { write_volatile(OUTSET, 1 << ROW1) }; // row 1 high: LED on
    cortex_m::asm::delay(32_000_000); // at least 32M cycles (0.5 s)
    unsafe { write_volatile(OUTCLR, 1 << ROW1) }; // LED off
    cortex_m::asm::delay(32_000_000);
}
```

- Lecture 12's QEMU UART, on real silicon: addresses from the datasheet, every access `unsafe`

[Live demo] demos/board: cargo run --release --bin blinky_raw

Blinky, PAC (Peripheral Access Crate)

```
let Some(p) = pac::Peripherals::take() else { ... };
p.P0.outclr.write(|w| w.pin21().clear().pin28().clear()); // both low first
p.P0.pin_cnf[21].write(|w| w.dir().output());
p.P0.pin_cnf[28].write(|w| w.dir().output());
loop {
    p.P0.outset.write(|w| w.pin21().set()); // LED on
    cortex_m::asm::delay(32_000_000);
    p.P0.outclr.write(|w| w.pin21().clear()); // LED off
    cortex_m::asm::delay(32_000_000);
}
```

- `nrf52833-pac` is **generated** by `svd2rust` from Nordic's register description: names, bit fields, and write-only registers are checked by types. No `unsafe`
- `Peripherals::take()`: Lecture 12's ownership tokens, generated for every peripheral
- Design choices differ: Embassy's PACs skip owned singletons and leave ownership to the HAL

[Live demo] demos/board: cargo run --release --bin blinky_pac

Blinky, BSP (Board Support Package)

```
let Some(board) = Board::take() else { ... };
let mut row1 = board.display_pins.row1; // the board knows which pin is which
let mut col1 = board.display_pins.col1;
let _ = col1.set_low();
loop {
    let _ = row1.set_high();
    cortex_m::asm::delay(32_000_000);
    let _ = row1.set_low();
    cortex_m::asm::delay(32_000_000);
}
```

- `microbit-v2` names the pins, and its pins are typestate GPIO from `nrf52833-hal`
- `set_high` is `embedded-hal`'s `OutputPin` (later today); it returns a `Result`
- **HW5 Part 0 is this program.** Run it before the next class

[Live demo] demos/board: cargo run --release --bin blinky_bsp

What Does Each Layer Cost?

	<code>.text</code> (bytes)	The blink loop
Raw	1,816	<code>str r1,[r0]</code> · delay · <code>str r1,[r0,#4]</code> · delay · <code>b</code>
PAC	1,848	the same instructions
BSP	2,024	the same instructions

```
loop:  mov    r3, r2                @ r0 = &P0.OUTSET, r1 = 1 << 21, r2 = 32,000,001
      str    r1, [r0]            @ OUTSET: LED on
wait1: subs   r3, r3, #1          @ at 0x330: 3 cycles per pass -> LED on ~1.5 s
      bne    wait1
      str    r1, [r0, #4]        @ OUTCLR: LED off
      mov    r3, r2
wait2: subs   r3, r3, #1          @ at 0x338: 7 cycles per pass -> LED off ~3.5 s
      bne    wait2
      b      loop
```

- The abstractions compile away. The extra bytes are one-time setup, mostly RTT
- One visible difference: the PAC's `write` starts from the reset value, so it writes `PIN_CNF = 3`, not `1`
- `delay(n)` promises **at least** `n` cycles (0.5 s here). The **same two instructions** take 3 or 7 cycles per pass, depending on **where they sit in flash**. Busy-waiting is not a clock: HW5's Part 0 blinky uses a hardware timer

Tools: probe-rs, RTT, and a HardFault

- `cargo run` builds, then `probe-rs run` flashes the chip and stays attached
- RTT (Real-Time Transfer): `rprintln!` writes into a RAM buffer; the probe reads it out while the program runs
- `unsafe` means **you** own the consequences:

```
const NOWHERE: *const u32 = 0x6000_0000 as *const u32; // nothing is mapped here
// SAFETY: none! This breaks read_volatile's contract on purpose.
let value = unsafe { read_volatile(NOWHERE) };
```

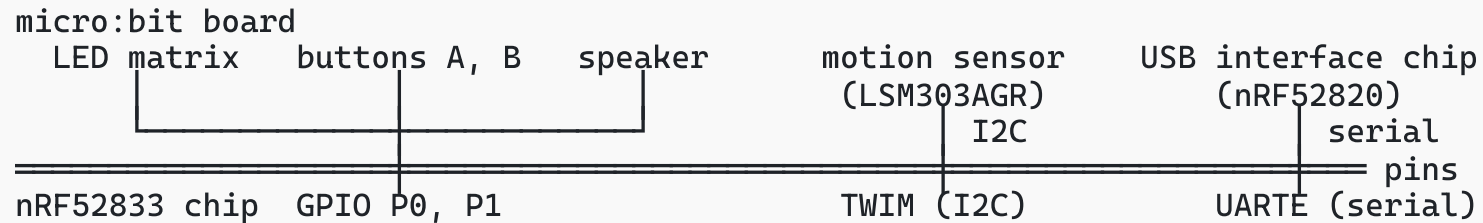
```
reading 0x60000000, where nothing is mapped ...
Firmware exited unexpectedly: Multiple
  Frame 0: HardFault_ @ 0xc6a
  Frame 1: HardFault <Cause: Escalated BusFault <Cause: Precise data access error
           at location: 0x60000000>> @ 0x288
  Frame 2: read_volatile<u32> @ 0x288 inline
  Frame 3: __cortex_m_rt_main @ 0x288
           ...\\demos\\board\\src\\bin\\hardfault.rs:24:26
```

[Live demo] demos/board: cargo run --release --bin hardfault

Lecture Outline

- HW5: a tour
- Hardware day: blinky three ways
- **How nRF peripherals talk**
 - the big picture: board, chip, peripherals
 - tasks, events, and interrupt enables
 - EasyDMA
 - a worked example: the random number generator
- Registers only the owner can reach
- Portable drivers: `embedded-hal`
- DMA: a second agent writing your memory
- From the toy to HW5 Part 1

The Big Picture: Board, Chip, Peripherals



On the **chip**, everything is memory-mapped:

- **Nordic's peripherals** at `0x4000_0000`: UARTE, TWIM, GPIOTE, TIMER, RNG, TEMP, WDT, the radio, ...; and **GPIO** at `0x5000_0000`, on a faster bus
- **ARM's Cortex-M4 CPU**, with its own **core peripherals** at `0xE000_0000`: NVIC (interrupts), SysTick, DWT (the cycle counter)
- Flash at `0x0`, RAM at `0x2000_0000` (Lecture 12)

Board parts aren't peripherals. You reach them *through* peripherals and pins.

HW5's drivers are all Nordic peripherals; Part 3's cycle counter is ARM's

- "Peripheral" is overloaded: in everyday computing it's a device *outside* the computer (a mouse, a printer); in a chip's datasheet it's a block *on the chip*, outside the CPU core. Same idea, different boundary: a mouse is to your laptop's USB controller as the motion sensor is to the nRF's TWIM

How nRF Peripherals Talk

nRF is Nordic Semiconductor's family of chips with built-in radios (the "RF" is *radio frequency*). The micro:bit's nRF52833 is one. Most nRF peripherals use the same three kinds of register:

Kind	Example	You...	The hardware...
Task	TASKS_START	write 1 to make it act	starts, stops, sends
Event	EVENTS_VALRDY	read it; write 0 to clear it	sets it to 1 on events
Interrupt enable	INTENSET / INTENCLR	write 1s to choose which events interrupt	raises the interrupt (Thursday)

- Waiting = polling an event until it's 1
- **Forgetting to clear an event** is the classic bug. Today: a "wait" that returns at once. Thursday: an interrupt that fires forever
- INTENSET / INTENCLR are write-1-to-set and write-1-to-clear, like GPIO's OUTSET / OUTCLR : other bits are untouched
- **The exception: GPIO** has no tasks or events of its own. A separate peripheral, **GPOTE** ("GPIO tasks and events"), adds them: in HW5, it turns button presses into events

Example: The Random Number Generator

```
const RNG: usize = 0x4000_D000;
const TASKS_START: *mut u32 = RNG as *mut u32;
const EVENTS_VALRDY: *mut u32 = (RNG + 0x100) as *mut u32;
const VALUE: *const u32 = (RNG + 0x508) as *const u32;

unsafe { write_volatile(TASKS_START, 1) };           // task: start generating
for _ in 0..8 {
    let value = unsafe {
        while read_volatile(EVENTS_VALRDY) == 0 {}    // wait for the event
        let value = read_volatile(VALUE);
        write_volatile(EVENTS_VALRDY, 0);             // clear it
        value
    };
    rprintln!("{:#04x}", value);                       // (the demo also times each wait)
}
```

```
0x55  after 13525 cycles (211 us)
0x23  after 3990 cycles (62 us)
0x58  after 3765 cycles (58 us)
0xa6  after 5046 cycles (78 us)
```

```
0x66  after 3958 cycles (61 us)
0x41  after 2837 cycles (44 us)
0x87  after 4581 cycles (71 us)
0x95  after 1174 cycles (18 us)
```

[Live demo] demos/board: cargo run --release --bin rng, with bias correction on (CONFIG = 1), which makes generation slower and variable. The RNG isn't used by HW5; your temperature driver follows the same pattern.

The Classic Bug: Never Clearing the Event

```
for v in values.iter_mut() {  
    unsafe {  
        while read_volatile(EVENTS_VALRDY) == 0 {}    // still 1 from last time: no wait  
        *v = read_volatile(VALUE);                    // ... and no new value yet  
    }  
}
```

```
[fc, fc, fc, fc, fc, fc, fc, fc] (8 reads in 4337 cycles)
```

- The first read waits for one fresh value; the other seven **don't wait at all**: eight reads in about the time it takes to generate one
- The event says "something happened **since you last cleared me**"
- Rust reports neither error nor fault: it just provides wrong data, fast

EasyDMA: Peripherals That Move Memory Themselves

Many nRF peripherals transfer data **directly** to and from RAM:

Register	Meaning
<code>PTR</code>	the buffer's address
<code>MAXCNT</code>	how many bytes
<code>TASKS_START</code>	go: the peripheral reads (or writes) the buffer by itself
<code>EVENTS_END</code>	done
<code>TASKS_STOP</code> → <code>EVENTS_STOPPED</code>	abort a transfer, and know when it has stopped
<code>AMOUNT</code>	how many bytes actually moved

- For the UART's transmitter: `TXD.PTR` , `TXD.MAXCNT` , `TASKS_STARTTX` , `EVENTS_ENDTX` , ...
- While a transfer runs, the CPU is free. **So is the buffer, as far as Rust knows.**
The second half of today is about that

Lecture Outline

- HW5: a tour
- Hardware day: blinky three ways
- How nRF peripherals talk
- **Registers only the owner can reach**
 - register handles with access kinds
 - pins as types, checked at compile time
 - splitting a peripheral into owned halves
- Portable drivers: `embedded-hal`
- DMA: a second agent writing your memory
- From the toy to HW5 Part 1

A Register Handle With an Access Kind

```
pub struct RO; pub struct WO; pub struct RW;           // access kinds
pub trait Readable {} impl Readable for RO {} impl Readable for RW {}
pub trait Writable {} impl Writable for WO {} impl Writable for RW {}

/// One 32-bit device register, usable while its owner is borrowed for 't.
pub struct Reg<'t, A> {
    addr: usize,
    _owner: PhantomData<(&'t (), A)>,
}

impl<A: Readable> Reg<'_, A> { pub fn read(&self) -> u32 { ... } }
impl<A: Writable> Reg<'_, A> { pub fn write(&self, value: u32) { ... } }
```

- Typestate (Lecture 12), with marker **traits** bounding `impl` blocks: a read-only register simply **has no** `write`
- `Reg` is zero-cost: it has one `usize`, and every check is at compile time

You'll see this in `mmio.rs` in the HW5 startup code. You'll implement the methods.

Only the Owner Can Reach the Registers

```
impl<A> Reg<'_, A> { const unsafe fn at(addr: usize) -> Self { ... } } // PRIVATE

pub struct TempRegs<'t> { // access kinds, from the datasheet
    pub tasks_start: Reg<'t, WO>,
    pub events_datardy: Reg<'t, RW>,
    pub temp: Reg<'t, RO>,
}

impl TEMP { // TEMP: the sensor's token
    pub(crate) fn regs(&self) -> TempRegs<'_> {
        // SAFETY: addresses from regs.rs, access kinds as in the datasheet,
        // and '&self' proves we own the TEMP peripheral.
        unsafe { TempRegs { temp: Reg::at(regs::temp::TEMP), ... } } // A = RO, inferred
    }
}
```

- **Constructing** a handle is the `unsafe`, contract-carrying step; **using** one is safe
- The lifetime `'t` is the borrow of the token: a handle can't outlive its owner, or be kept after the token moves to someone else
- Every `unsafe` in your HAL is in one file, and each claim is written next to it

Pins as Types, Checked at Compile Time

```
struct Led<const ROW: u8, const COL: u8>;    // each position is its own type

impl<const ROW: u8, const COL: u8> Led<ROW, COL> {
    fn new() -> Self {
        const { assert!(ROW < 5 && COL < 5, "the LED matrix is 5x5") };
        Led
    }
}
let _off_the_board = Led::<7, 2>::new();
```

```
error[E0080]: evaluation panicked: the LED matrix is 5x5
  |
  |         const { assert!(ROW < 5 && COL < 5, "the LED matrix is 5x5") };
  |         ^^^^^^^ evaluation of `Led::<7, 2>::new::{constant#0}` failed here
note: the above error was encountered while instantiating `fn Led::<7, 2>::new`
```

- The `const` block is evaluated **per instantiation**, at compile time
- HW5: `Pin<PORT, PIN, MODE>`, and `Pin<1, 12, _>` must not compile

demos/host: cargo build -p const-assert --features misuse-off-the-board

Splitting a Peripheral Into Owned Halves

- The UART has a transmitter and a receiver. The main loop wants to send a log while the receiver waits for a keypress
- One token for the whole UART would make them take turns. So **split** it:

```
impl Uarte {  
    pub fn split(self) -> (UarteTx, UarteRx) { ... }    // consumes the whole UART  
}
```

- Each half owns **only its own registers**. Two owners, no overlap
- The hardware version of `split_at_mut : one &mut [T]` becomes two disjoint ones

Provided in HW5's `uarte.rs`.

Lecture Outline

- HW5: a tour
- Hardware day: blinky three ways
- How nRF peripherals talk
- Registers only the owner can reach
- **Portable drivers:** `embedded-hal`
- DMA: a second agent writing your memory
- From the toy to HW5 Part 1

embedded-hal 1.0: Drivers That Work on Any Chip

- Traits for what drivers need: `OutputPin`, `InputPin`, `I2c`, `SpiDevice`, `DelayNs` (and `embedded-io`'s `Write` for serial ports)
- A driver is generic over them. Here's one for the micro:bit's motion sensor:

```
pub struct Lsm303<I> { i2c: I }

impl<I: I2c> Lsm303<I> {
    pub fn accel_id(&mut self) -> Result<u8, I::Error> {
        let mut value = [0u8];
        self.i2c.write_read(ACCEL_ADDR, &[ACCEL_WHO_AM_I], &mut value)?;
        let [value] = value;
        Ok(value)
    }
}
```

- Monomorphized (Lecture 6): one copy per bus type, no vtable, zero cost

The Same Driver, on the Board and on the Laptop

On the board, `I` is `nrf52833-hal`'s I2C controller:

```
let i2c = Twim::new(board.TWIM0, board.i2c_internal.into(), twim::Frequency::K100);
let mut sensor = Lsm303::new(i2c);
```

```
accelerometer WHO_AM_I = 0x33 (expect 0x33)
magnetometer  WHO_AM_I = 0x40 (expect 0x40)
x      64      y      -56      z    -1032      tilt (milli-g), lying flat: gravity on -Z
x    1092      y      -28      z       -36      ... standing on its edge: gravity on +X
```

On the laptop, `I` is a mock that checks the exact bus traffic:

```
let expected = [Transaction::write_read(ACCEL_ADDR, vec![ACCEL_WHO_AM_I], vec![0x33]),
                Transaction::write_read(MAG_ADDR, vec![MAG_WHO_AM_I], vec![0x40])];
let mut sensor = Lsm303::new(Mock::new(&expected));
assert_eq!(sensor.accel_id().unwrap(), 0x33);
```

[Live demo] demos/board: cargo run --release --bin who_am_i, then --bin tilt. demos/host: cargo test -p accel-driver (embedded-hal-mock 0.11). HW5: the provided LED-matrix driver is generic over OutputPin, so it runs on your pins.

Lecture Outline

- HW5: a tour
- Hardware day: blinky three ways
- How nRF peripherals talk
- Registers only the owner can reach
- Portable drivers: `embedded-hal`
- **DMA: a second agent writing your memory**
 - why DMA registers are special
 - a naive API, and how `mem::forget` breaks it
 - owning a `'static` buffer; fences; RAM only
 - the safety argument
- From the toy to HW5 Part 1

DMA: A Second Agent Writing Your Memory

- Once started, the DMA engine reads or writes the buffer **concurrently** with your code, **outside the compiler's view**
- To Rust, it's an extra thread that `Send` and `Sync` know nothing about
- The hazards:
 - the CPU writes the buffer while the engine reads it: **torn data**
 - the buffer is freed or reused while the engine writes it: **use-after-free**
 - the compiler reorders buffer accesses across the start or the end
- Our job, as in HW2–3 and Lecture 8: one small module, a sound safe API, and a written argument

DMA Registers Are Special

With today's register handles, this would be **safe** code:

```
regs.txd_ptr.write(addr);           // any address at all
regs.tasks_starttx.write(1);        // and now the hardware reads it
```

- For a receiver, the hardware would **write** wherever `addr` points
- So a safe `write` to a pointer register is an **unsound safe API**
- **Soundness is about what the hardware does** when you write a register, not just which addresses your Rust code touches
- HW5's fix: two more access kinds, `DmaPtr` and `DmaStart`, whose methods are `unsafe fn`s with a memory contract. **Which registers get them** is your design decision, defended in your safety argument

Pass 1: A Transfer That Borrows the Buffer

A fake DMA engine on the laptop: a background thread that writes the buffer, one byte per millisecond

```
impl Dma {
    pub fn start<'a>(&'a mut self, buf: &'a mut [u8], value: u8) -> Transfer<'a> {
        // SAFETY (?): assumes the Transfer is waited on before 'a ends
        unsafe { start_engine(buf.as_mut_ptr(), buf.len(), value) };
        Transfer { _buf: PhantomData }
    }
}

impl Drop for Transfer<'_> { fn drop(&mut self) { wait_for_engine() } }
```

```
error[E0506]: cannot assign to `buf[_]` because it is borrowed
  8 |         let transfer = dma.start(&mut buf, 0xAB);
    |         ----- `buf[_]` is borrowed here
 12 |         buf[0] = 1; // the engine is writing `buf` right now
    |         ^^^^^^^^^^ `buf[_]` is assigned to here but it was already borrowed
```

Airtight? **Question:** what safe code could break it?

demos/host: cargo build -p fake-dma --bin naive_ok --features misuse-touch-naive

Discussion: Can You Break This DMA API?

```
// You are a different module. You hold these, and nothing else:
let mut dma = Dma::new();
let mut buf = [0u8; 8];
// fn start<'a>(&'a mut self, buf: &'a mut [u8], value: u8) -> Transfer<'a>
// fn wait(self)                                     (and Transfer's Drop waits for the engine)
// One more tool, from the standard library. It is NOT unsafe:
// fn std::mem::forget<T>(t: T)                      (takes t and drops it WITHOUT running its Drop)
```

- Use `forget` to make the engine write memory that **someone else** is using, with **no unsafe**
- Which assumption in `start`'s `SAFETY` comment is false?
- Design question: what is the smallest change to `start`'s **signature** that makes your attack **harmless**?

Discuss in small groups for 3 minutes, share 2 minutes. Lecture 8 asked the same question of `SharedResults`.

mem::forget Is Safe

```
let transfer = dma.start(&mut buf, 0xAB);
mem::forget(transfer);    // safe! No Drop runs, and the borrow of `buf` ends
buf[0] = 1;               // so this compiles, while the engine writes `buf` too
```

```
$ cargo run --bin forget_race
buf: [ab, ab, 00, 00, 00, 00, 00, 00]      <- our write of 1 was lost

$ cargo +nightly miri run --bin forget_race
error: Undefined Behavior: Data race detected between (1) non-atomic write on thread
    `main` and (2) non-atomic write on thread `unnamed-1`
48 |         unsafe { addr.byte(i).write(value) };
   |                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ (2) just happened here
help: and (1) occurred earlier here
17 |         buf[0] = 1; // so this compiles, while the engine writes `buf` too
```

Zero `unsafe` in the caller

demos/host/fake-dma: forget_race.rs. Miri: see the Oct 6 talk.

Worse: A Dangling Buffer

```
fn start_and_return(dma: &mut Dma) {  
    let mut buf = [0u8; 8];           // on this function's stack  
    mem::forget(dma.start(&mut buf, 0xAB));  
}  
// back in main:  
let other_data = [7u8; 64];          // reuses that stack space
```

```
$ cargo run --release --bin forget_dangling  
other_data[..8]: [ab, ab, ab, ab, ab, ab, 07, 07] (we wrote 07s)  
  
$ cargo +nightly miri run --bin forget_dangling  
error: Undefined Behavior: memory access failed: alloc122 has been freed, so this  
    pointer is dangling  
help: alloc122 was allocated here:  
13 |     let mut buf = [0u8; 8];  
help: alloc122 was deallocated here:  
16 | } // `buf`'s stack slot is gone; the engine keeps writing into it
```

- The engine is writing into a **different variable** now. Run it again: a different count, or a crash (one run in eight: access violation), or nothing visible

Native runs on Windows 11, rustc 1.93, --release; Miri on nightly 2026-09-21.

Why: Rust Doesn't Promise That Destructors Run

- `mem::forget`, a reference cycle of `Rc`s, a leaked `Box`: all safe ways to skip `Drop`
- This was settled in 2015 ("the Leakpocalypse"), when an early `thread::scoped` API turned out unsound for exactly this reason
- So "`Drop` waits for the hardware" is not a safety argument for memory someone else can reuse
- The fix has to make forgetting harmless

Lecture 7's `thread::scope` avoids the problem differently: you never get a guard to forget, because the scope itself waits.

Pass 2: The Transfer Owns a `'static` Buffer

```
impl Dma {
    pub fn start(self, buf: &'static mut [u8], value: u8) -> Transfer { ... }
}
impl Transfer {
    pub fn wait(self) -> (&'static mut [u8], Dma) { ... }    // both come back
}
```

- `'static`: the buffer is never freed, so the engine can't write freed memory
- **Borrowed forever**: passing `buf` reborrows it for `'static`, so the caller can never touch it again, whether or not the `Transfer` is forgotten
- `wait` is the only way back: it returns a fresh `&'static mut`, after the engine is done
- `forget(transfer)` just leaks the buffer: unreachable forever. Miri: clean

```
error[E0503]: cannot use `*buf` because it was mutably borrowed
14 |         let transfer = Dma::new().start(buf, 0xAB);
    |         ----- argument requires that `*buf` is borrowed for `'static`
18 |         buf[0] = 1; // `buf` is borrowed forever by the transfer
```

demos/host/fake-dma: owned.rs, and --features misuse-touch-owned for the error. Lecture 8's Claim again: a linear token meaning "the hardware has this memory".

Where Does a `&'static mut` Come From?

```
static BUF: StaticCell<[u8; 512]> = StaticCell::new();  
let buf: &'static mut [u8; 512] = BUF.init([0; 512]); // once; a second init panics
```

```
let buf: &'static mut [u8; 512] = cortex_m::singleton!([u8; 512] = [0; 512]).unwrap();
```

- Both: a `static` in RAM, handed out **once** as a unique reference
- **Not** `static mut`: taking `&mut` to one is unchecked, and in edition 2024 it's an error by default. Thursday shows why

HW5's firmware uses `cortex_m::singleton!` for the log's DMA buffer.

`volatile` Isn't Enough: `compiler_fence`

```
buf[..n].copy_from_slice(msg);           // ordinary writes to the buffer
compiler_fence(Ordering::SeqCst);         // ... must stay above this line
start_the_engine(buf);                   // volatile writes: PTR, MAXCNT, START
wait_for_the_end_event();                // volatile reads
compiler_fence(Ordering::SeqCst);         // the engine's writes come before
let reply = buf[0];                      // ... any read below this line
```

- Volatile orders volatile accesses **among themselves** (Lecture 12). The buffer writes are ordinary, so the compiler may move them past the start
- `compiler_fence` stops the **compiler** reordering across it. One core, no cache: that's all we need
- Atomics would also work; a fence is the minimal tool

EasyDMA Reads Only RAM. The Hardware Won't Tell You

```
fn send(&mut self, data: &'static [u8]) { ... } // 'static: never freed. Good?  
send(b"hello\r\n"); // a string literal is 'static ... and lives in flash (Lecture 12)
```

Measured on a micro:bit: EasyDMA pointed at 27 bytes of a string literal at `0x2824`

- **Sent:** 27 bytes, **all zero**. The terminal shows nothing
- **Reported:** "end of transfer" on time, `AMOUNT` = 27, **no error**

The type system can't see this, so it's a **runtime check inside the trusted module**: return an error if the buffer isn't in RAM. Then copy literals through a RAM buffer (HW5's `write!` support does this for you)

Captured by listening on the TX pin with the UART's own receiver; RAM lines before and after came back exactly. nrf-hal's version of the check returns `BufferNotInRAM`.

Other Designs You'll Meet

- Stop the DMA in `Drop` (Embassy's `OnDrop` guard). Sound for `async`:
a `Pin`ned future's memory can't be reused without running `drop` (Lecture 9).
For a plain struct, it's only defense in depth
- `embedded-dma`'s `unsafe trait ReadBuffer / WriteBuffer`: a type promises, in writing, that its buffer stays put. The contract becomes part of the API
- Copy through a stack buffer (Embassy, for data in flash): correct, but costs a copy

The Safety Argument for the DMA Module

Hazard	The engine accesses memory concurrently with the CPU: a concurrent writer, a dangling buffer, a non-RAM buffer, reordered accesses
Invariant	While a transfer is active, its buffer is RAM that is reachable only through the <code>Transfer</code> , and it is never freed
Establish	<code>start</code> checks the buffer is in RAM, then takes the <code>&'static mut</code> and the engine by value; a fence precedes the start task
Maintain	Only <code>wait</code> , after the end event and a fence, gives them back. <code>forget</code> only leaks. Uniqueness of the engine token means one transfer at a time

- `Transfer` is Lecture 8's `claim`: an unforgeable, linear token, now meaning *"the hardware has this memory"*

From the Toy to HW5 Part 1

What carries over, and what's yours:

- `mmio.rs`: `Reg`'s methods, and the volatile accessors. Pick access kinds for the UART's registers, including which are `DmaPtr` / `DmaStart`
- `gpio.rs`: Lecture 12's typestate across two ports, with the `const` check
- **Temperature sensor, timer**: today's task/event pattern
- **Watchdog**: one-way typestate (Lecture 12)
- `dma.rs`: the real UART transmit sequence, `Drop` that stops a running transfer, and one-byte receive into a provided `static RX_BYTE: AtomicU8` (an atomic, because hardware changes it under Rust's feet)
- Your write-up: H-I-E-M arguments for `mmio` and `dma`

Summary

- **Abstraction layers compile away:** raw, PAC, and BSP blinky run the same loop
- nRF peripherals speak tasks, events, and EasyDMA. Clear your events
- **Register handles** make the owner the only way in, at zero cost
- `embedded-hal` lets one driver run on any chip, and in host tests
- **DMA is a second agent.** A borrow isn't enough, because `forget` is safe: own a `'static` buffer, fence, and check it's in RAM

Tonight: HW5 Part 0. Install probe-rs, flash the blinky, and open the serial terminal. Setup problems found tonight are cheap; found on Nov 2, they're expensive

Thursday: interrupts, the other way hardware runs your code when you weren't expecting it