Microprocessors and Embedded Systems

University Electrical Engineering

What this course is, and how to use it

This is the eighth course in University Electrical Engineering: **microprocessors and embedded systems** . EE03 built the logic; this course builds the machine that runs programs, and the discipline of making hardware and software meet.

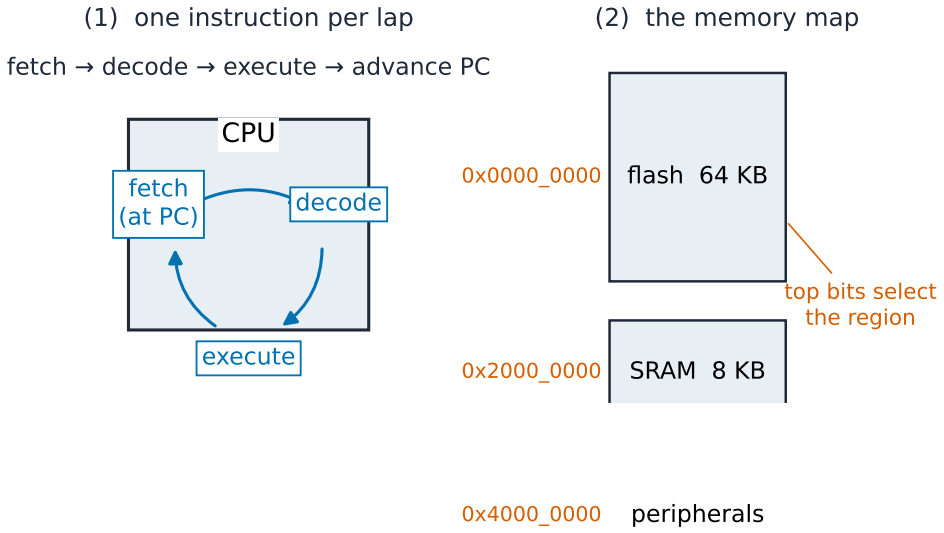
- Unit 8.1 maps the architecture: CPU, registers, memory, and the fetch-decode-execute cycle.
- Units 8.2–8.3 program it: embedded C and assembly, then memory-mapped I/O.
- Units 8.4–8.5 add time: interrupts/timers and the serial protocols.
- Units 8.6–8.7 close with analog interfaces and debugging method.

The running story is one small product: a battery-powered sensor node with a button, an LED, a serial link and a 12-bit sensor. Every unit below moves that node forward, and every number is checked the way a firmware engineer checks it – by bit widths and limits (does the value fit?), by the clock it assumes (every timing closes against 84 MHz), and by tracing the exact state changes of every code fragment.

Microcontroller architecture

Why it matters. The thermostat on the wall, the anti-lock brake controller and the earbud on the desk each run one of these: a complete computer on a chip the size of a grain of rice, costing less than the postage that ships it.

The idea, with a picture. A **microcontroller** integrates three things on one chip: flash memory holding the program, SRAM holding the data, and a CPU that runs the **fetch-decode-execute cycle** : the **program counter** (PC) holds the address of the next instruction, the control unit decodes it, the datapath executes it, and the PC advances (or jumps). Every transfer rides the **bus** – address lines saying where, data lines carrying what, control lines saying read or write – sequenced by the **clock** .



One instruction per lap: fetch at the PC, decode, execute, advance

The derivation. Addresses are unsigned integers, so an N-bit address field addresses 2^N locations exactly. The whole memory map follows:

$$\text{size} = 2^N \text{ bytes} \quad \Longleftrightarrow \quad N = \log_2(\text{size}) \text{ address lines.}$$

Flash at `0x0000_0000` and SRAM at `0x2000_0000` never collide because their top bits differ – the address decoder just looks at bits [31:16].

> **Worked example 1 – sizing the memory map.** The node's chip has 64 KB flash and 8 KB SRAM. Flash: $64\,\text{KB} = 2^{16}$ bytes $\rightarrow$ 16 address lines, so it occupies `0x0000_0000`–`0x0000_FFFF`. SRAM: $8\,\text{KB} = 2^{13}$ bytes $\rightarrow$ 13 lines, at `0x2000_0000`–`0x2000_1FFF`. Check: any address with bits [31:16] = `0x0000` and [15:0] $\leq$ `0xFFFF` lands in flash; `0x2000_…` lands in SRAM; the regions cannot overlap because their top bits differ ✓.

> **Worked example 2 – how fast is the loop itself?** The node's CPU runs at 84 MHz with a nominal one instruction per cycle (a Cortex-M class core): 84×10^6 instructions per second, so one instruction takes $\dfrac{1}{84 \times 10^6} = 11.9$ ns. A 100-instruction control pass therefore costs 1.19 µs – and a 1 ms control loop has room for about 840 passes if nothing else intervenes. The budget arithmetic that starts here is reused in every later unit.

Common mistakes. Mixing address bits with data bits (the address lines count locations, the data lines their width). Forgetting that the map is fixed in the linker script: the same chip boots at whatever address the hardware vendor chose. And

quoting program size in bits when flash sizes are in bytes – a factor of eight that sinks the part choice.

Summary. One chip: flash + SRAM + CPU cycling fetch, decode, execute. An N-bit address field addresses 2^N bytes; regions with distinct top bits cannot overlap; the clock turns instruction count into seconds.

Embedded C and assembly

Why it matters. The node's firmware is a few thousand lines of C touching hardware registers directly. Nothing between your line of code and a transistor – which is exactly why the mapping from C to machine must be precise.

The idea, with a picture. A **memory-mapped register** is just an address; C reaches it through a `volatile` pointer (volatile tells the compiler: this memory changes outside your view, touch it every time). The compiler turns C into loads, stores, arithmetic and branches; **assembly** is the same program written in those instructions by hand, one line each.

C

```
*GPIO_OUT |= (1u << 5);          set bit 5 only
```

assembly: load, mask, store

```
LDR  r0, =0x20000100     address of GPIO_OUT
LDR  r1, [r0]            bus transfer 1: read
ORR  r1, r1, #0x20       inside the CPU, no bus
STR  r1, [r0]            bus transfer 2: write
```

register transfer

```
GPIO_OUT ← GPIO_OUT | 0x20      other bits unchanged
```

One C line, one instruction sequence, one register transfer – three views of one act

The derivation. Read-modify-write is the fundamental act: to change one pin without disturbing its neighbours, the CPU must load the register, mask the bit locally, and store it back. That is three operations but only two bus transfers: one read and one write. This example uses a plain read/write word register; some real peripherals also provide atomic set/clear registers:

```
volatile uint32_t *const GPIO_OUT = (uint32_t *)0x40000000u;
*GPIO_OUT |= (1u << 5);   // load, OR, store - set bit 5 only
```

Worked example 1 – trace the counter. The C loop

```
uint32_t count = 0;
while (1) { count = count + 1; }
```

An illustrative load-increment-store plus branch implementation traces $0, 1, 2, 3, \ldots$. A compiler may instead keep this non-volatile counter in a CPU register. Check the limit: the largest value is $2^{32} - 1 = 4\,294\,967\,295$; the next increment wraps to zero after 2^{32} increments from zero. Assuming four clock cycles per iteration at 84 MHz, the wrap takes $\dfrac{4\,294\,967\,296 \times 4}{84 \times 10^6} \approx 204$ s – arithmetic, not waiting three minutes with a stopwatch.

> **Worked example 2 – the bit mask, escalated.** To toggle bits 5 and 6 and clear bit 3 in one expression: the mask is $(1 \ll 5) \,|\, (1 \ll 6) = 0x60$. Then `*GPIO_OUT = (*GPIO_OUT & ~0x08) ^ 0x60` clears bit 3 and flips 5 and 6 in one read-modify-write. Verify on paper with `GPIO_OUT = 0xCE`: `0xCE & ~0x08 = 0xC6`, then `0xC6 ^ 0x60 = 0xA6` – bit 3 now 0 (was 1), bits 6 and 5 flipped from 1, 0 to 0, 1 ✓. The hex check on paper is the fastest debugger in this unit.

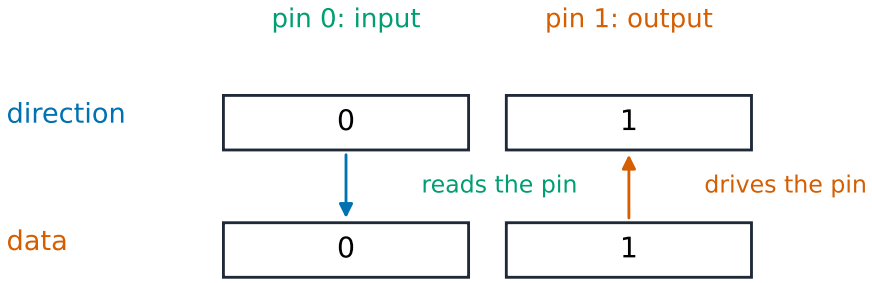
Common mistakes. Dropping `volatile` (the compiler caches the register in a CPU register and your loop reads it forever). Assuming C evaluation order between two register writes when the hardware needs them in sequence – insert a barrier or read-back. And writing bit 5 as `*GPIO_OUT = 0x20`, which clears every other bit on the port.

Summary. Registers are addresses; `volatile` keeps the compiler honest; this plain word-register example uses load-mask-store. Trace state on paper in hex before trusting the silicon.

Memory and I/O

Why it matters. The node's button, LED and sensor all live outside the CPU. How they connect decides whether the firmware is a clean table of addresses or a pile of magic numbers.

The idea, with a picture. **Memory-mapped I/O** makes a peripheral register look exactly like a memory location: reading a data register receives a byte; writing one sends it. A **GPIO** port has two such registers – direction (input or output, one bit per pin) and data (the pin's level) – so configure once, then read or write forever.



configure direction once, then read or write data

Direction says which way the pin speaks; data says what it says

The derivation. **Polling** is the honest baseline: the status register's done-bit is a memory read in a loop. Each lap costs the loop's instructions; waiting T seconds burns

$$T \times f_{\text{clock}}$$

CPU cycles, or approximately $T f_{\text{clock}}/C$ complete laps when each lap costs C clock cycles. The factor C changes the number of checks, not the CPU time occupied by a continuous busy wait. Detection and timeout tests are quantised to the loop's check times.

> **Worked example 1 – the cost of asking.** The node polls a sensor-ready bit at 84 MHz; each loop lap is 4 cycles (load, AND, branch, increment), so the CPU performs $84 \times 10^6/4 = 21$ million laps per second. Waiting 1 ms is ~21 000 laps = 84 000 cycles: 100% of the CPU for that millisecond, with zero useful work.

> **Worked example 2 – the whole node's map.** The node's peripherals sit at `0x4000_0000` (GPIO), `0x4001_0000` (UART), `0x4002_0000` (timer), `0x4003_0000` (ADC). A firmware constant table maps each driver to its base; a typo in one entry (`0x4001_0000` vs `0x4001_0004`) lands four bytes into the UART's register set – which is why every driver file opens with named offsets, never raw addresses. The check: after init, read each peripheral's ID register back; a wrong base reads garbage on the first try.

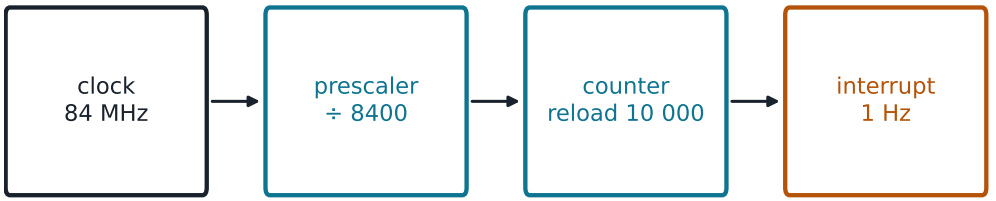
Common mistakes. Forgetting the direction register (the pin stays an input and the write does nothing). Polling with no timeout – a dead sensor hangs the whole product. And mixing up a port's bit number with the pin number on the package: the datasheet's pin table, not the header's convenience macro, is the truth.

Summary. Peripherals are memory: base plus named offsets. Configure direction once, poll with a timeout, and let a read-back ID check prove the map before any real work runs.

Interrupts and timers

Why it matters. The node runs on a battery for a year. Polling burns the battery asking questions; interrupts let the CPU sleep until the world needs it – the difference between a week and a year.

The idea, with a picture. An **interrupt** suspends the main program where it stands: the hardware pushes the return state, **vectors** to an **interrupt service routine** (ISR) chosen by the **NVIC** (the nested vectored interrupt controller, which also ranks priorities), the ISR runs and returns. A **timer** is just a counter clocked through a divider – the **prescaler** . In the STM32-style upcounter used here, an update event occurs when CNT wraps from ARR back to zero; an enabled interrupt can then call the ISR.



84 000 000 ÷ 8400 = 10 000 Hz; 10 000 ticks at 10 kHz = 1 s

Two divisions turn 84 MHz into a 1 Hz interrupt

The derivation. Timer frequency divides twice:

$$f_{\text{interrupt}} = \frac{f_{\text{clock}}}{PR}, \qquad P = PSC + 1, \quad R = ARR + 1,$$

where PSC and ARR are the stored 16-bit register values, at most 65,535. Each divider factor can reach 65,536 because zero is counted too. A design is valid when the stored values fit AND $PR = f_{\text{clock}}/f_{\text{interrupt}}$. Here ARR is nonzero and the counter runs continuously.

> **Worked example 1 – a 1 Hz heartbeat.** From 84 MHz: divider factor 8400 gives $84 \times 10^6/8400 = 10$ kHz; another factor of 10,000 gives exactly 1 Hz. Write PSC = 8399 and ARR = 9999. Both stored values fit 16 bits, and $8400 \times 10\,000 = 84 \times 10^6$ ✓ closes the design. A divider factor of 84,000 would need PSC = 83,999, which does not fit.

> **Worked example 2 – interrupt latency vs polling, priced.** A GPIO interrupt with 12 cycles of entry, a 30-instruction ISR and 10 cycles of return costs $12 + 30 + 10 = 52$ cycles $= 0.62$ µs at 84 MHz – against the 21 000 laps the polled wait burned. But the ISR's cost is paid on every event: at 1 kHz of events that is 52 000 cycles/s $= 0.06\%$ of the CPU; at 1 MHz of events it is 62% – past some rate, polling a fast peripheral is cheaper than interrupting. The crossover arithmetic is the design decision.

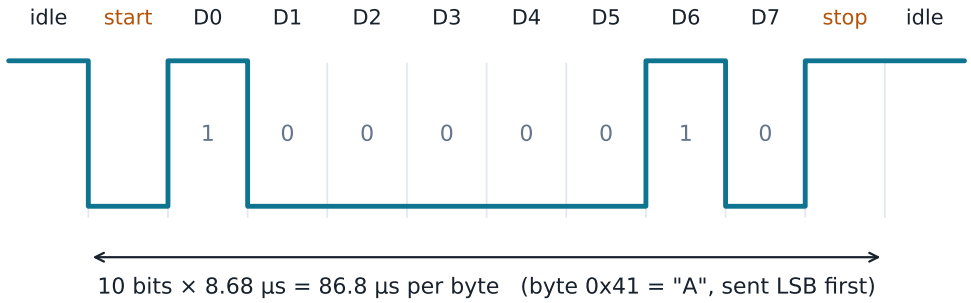
Common mistakes. An ISR that forgets to clear its interrupt flag – the CPU returns and immediately re-enters, forever. Long ISRs (do the minimum, set a flag, let the main loop work). And racing the timer: reading a 16-bit timer on an 8-bit bus needs the high byte read twice or latched, or a carry between reads corrupts the value.

Summary. Interrupt = vector + short ISR + flag clear; timer = $(PSC+1)(ARR+1)$ against the clock, with both stored values inside their registers. Interrupts win at low event rates; polling wins past ~1 MHz.

Serial protocols

Why it matters. The node's readings leave through three wires or fewer. Choosing UART, SPI or I2C – and pricing the wire against the speed – is a weekly decision in embedded work.

The idea, with a picture. **UART** sends asynchronous bytes: no shared clock, just a start bit, 8 data bits, a stop bit, at a **baud rate** both ends agreed on. **SPI** shares a clock: the master drives SCLK and chip-select, data flows on MOSI out and MISO in – full duplex. **I2C** needs two wires (SDA data, SCL clock), addresses devices with 7 bits, and wires them open-drain so any device can hold the line low.



An 8N1 frame: 10 bit times per byte, 86.8 µs at 115 200 baud

The derivation. UART has no clock line, so both ends reconstruct time from the agreed baud: a byte in 8N1 framing occupies exactly 10 bit times,

$$T_{\text{byte}} = \frac{10}{\text{baud}}, \qquad \text{throughput} = \frac{\text{baud}}{10} \text{ bytes/s}.$$

SPI instead sends one bit per complete clock cycle, so a byte takes 8 clocks plus setup: $T_{\text{byte}} \approx 8/f_{\text{SCLK}}$ – the clock is the speed.

> **Worked example 1 – sizing UART.** At 115 200 baud, 8N1: throughput is $115\,200/10 = 11\,520$ bytes/s; one byte takes $10/115\,200 = 86.8$ µs. Against the node's 1 ms control loop: 11 bytes just fit (0.95 ms) ✓; a blocking 100-byte burst (8.68 ms) blows the budget – split it or raise the baud. The sizing decides, not preference.

> **Worked example 2 – the three-way price.** SPI at 10 MHz clocks a byte in $8/10\,\text{MHz} = 0.8$ µs – about 108× faster than UART here – but costs a chip-select wire per device. I2C at 400 kHz needs 9 clocks for 8 data bits plus ACK: $9/400\,\text{kHz} = 22.5$ µs, before address, start and stop overhead. At these chosen rates it lies between SPI and UART, but two wires serve the whole board (7-bit address = up to 112 devices at 1 address per device). Sensor on I2C, flash on SPI, console on UART – the standard split, each priced by this arithmetic.

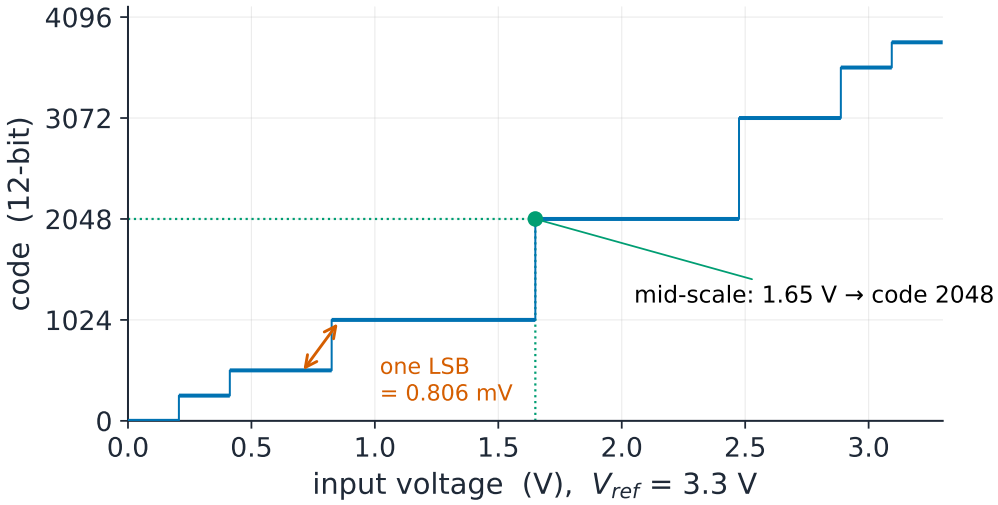
Common mistakes. Mismatched baud (the frame still 'works' and the bytes are garbage – check the divisor math against the peripheral clock, not the core clock). Reading an I2C address as 8 bits: the 7-bit address plus the read/write bit packs into one byte, and datasheets quote both forms. And forgetting SPI mode (clock polarity and phase): the four modes exist because devices differ; the wrong one clocks every byte one bit off.

Summary. UART: 10 bit times per byte, no clock, both ends agree the baud. SPI: one bit per clock, a wire per device. I2C: two wires, 7-bit addresses, open-drain. Price bytes-per-millisecond against wires before choosing.

ADC, DAC and sensors

Why it matters. The physical world speaks volts; the CPU speaks numbers. The node's temperature reading is only as good as the converter and its wiring – and the errors here are quiet, not loud.

The idea, with a picture. An **ADC** quantises: an N-bit converter spanning a reference voltage V_{ref} has a **resolution** of one **LSB** $= V_{\text{ref}}/2^N$. The choice of N is the choice of how finely the world is measured; the wiring decides whether that fineness survives.



Each code owns a one-LSB band of input voltage; mid-scale is code 2048

The derivation. Quantisation maps a voltage to the code whose band contains it: code $= \lfloor V_{\text{in}}/V_{\text{ref}} \times 2^N \rfloor$ (with variants at the edges). Reconstructing each input band at its midpoint gives an ideal quantisation-error bound of half an LSB within the input range; an individual input can have zero error.

> **Worked example 1 – the 12-bit sensor.** With $V_{\text{ref}} = 3.3$ V, one LSB is $3.3/4096 = 0.806$ mV. A 1.65 V input reads $1.65/3.3 \times 4096 = 2048$ – mid-scale exactly ✓. If the node must resolve 0.1 °C and the sensor gives 10 mV/°C, it needs 1 mV of resolution – a 12-bit converter (0.806 mV) passes, a 10-bit one (3.22 mV) fails: the requirement, not the catalogue, picks N.

> **Worked example 2 – the source-impedance trap.** The ADC's sample-and-hold capacitor (≈ 7 pF on this chip) charges through the source resistance, and settling to ½ LSB needs a sampling time of at least $R_{\text{source}} C \ln(2^{N+1})$. For a 50 kΩ source that is $50\,000 \times 7\,\text{pF} \times 9.01 \approx 3.2$ µs for a full-scale input jump in this simple RC model. Shrink the window to 1 µs and the same bound allows only 16 kΩ. A 100 kΩ thermistor divider (50 kΩ Thevenin) therefore needs at least a 3.2 µs window – or halve the divider values. A **DAC** walks the same staircase in reverse: writing 2048 to a 12-bit DAC with $V_{\text{ref}} = 3.3$ V produces 1.650 V.

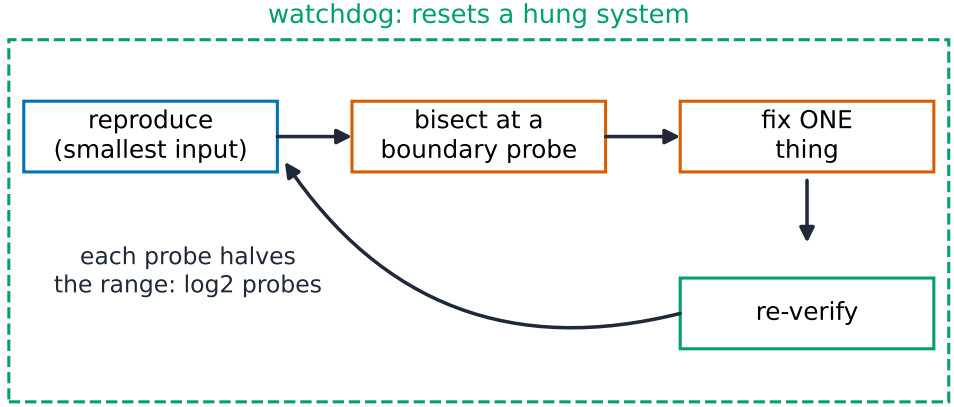
Common mistakes. Quoting resolution in bits and forgetting the reference (0.806 mV is only true at 3.3 V). Treating the ADC input as a voltmeter: its sampling current loads high-impedance sensors, as the RC bound above shows. And averaging to 'buy' bits – averaging 16 samples of white noise gains about 2 bits of effective resolution, but it cannot repair a systematic offset or a drooped sample.

Summary. One LSB $= V_{\text{ref}}/2^N$; the requirement picks N. Midpoint reconstruction gives a half-LSB quantisation-error bound; the RC bound $t_s \geq RC \ln(2^{N+1})$ trades source impedance against sampling time – the allowed source impedance depends on the model and acquisition window.

Debugging and integration

Why it matters. Firmware is tested by running it on the exact hardware it will ship on, and every bug arrives as a rumour ("it froze twice last night"). Method is the difference between fixing it and renaming it.

The idea, with a picture. The loop is: reproduce with the smallest input; bisect the boundary between trusted and suspected code (an LED, a breakpoint, or one UART print); fix ONE thing; re-verify. A **watchdog** is the safety net that resets a hung system – kick the counter before it expires, or the product reboots by design instead of freezing by accident.



Reproduce, bisect, fix one thing, re-verify – and a watchdog outside the loop

The derivation. Bisection is the algorithm: if the fault reproduces with the boundary probe placed at the midpoint of the suspected range, the bug lives in the first half; else in the second. Each probe halves the range, so $\log_2$(suspicious instructions) probes suffice – ten probes for a thousand instructions. The same halving that runs the ADC ladder runs the debug loop.

> **Worked example 1 – debounce as a state trace.** The node's button reads 1, 0, 0, 1 across four 10 ms samples (a bouncing contact). Rule: require two consecutive equal readings. Trace: t = 0 ms reads 1 (no verdict); t = 10 ms reads 0 (one, no verdict); t = 20 ms reads 0 again → stable 0 declared. The 1 at t = 30 ms is one sample, not yet stable, and is correctly ignored. **Debounce** = N consecutive samples = a timer tick + a GPIO read + a state decision: the whole course in one trace.

Worked example 2 – bisecting the freeze. The node freezes within an hour of a specific log sequence; the suspected region is 1 024 instructions of the comms driver. Make instruction 512 light the LED: if the node freezes with the LED still dark, the bug is in the first half; if the LED lights before the freeze, it is in the second. Each run halves the region: $512 \to 256 \to 128 \to \ldots$ – ten runs bound it to one instruction. The arithmetic says why bisection beats rereading the code hoping to see it: $2^{10} = 1024$.

Common mistakes. Fixing two things at once (one of them worked, one broke something else – now there are three bugs). Debugging on a different board revision than the failing one. And disabling the watchdog 'for now': the field failure the watchdog would have survived is exactly the one that ships.

Summary. Reproduce, bisect, fix one, re-verify; $\log_2$ probes bound the hunt. Debounce turns noise into events by N consecutive samples. The watchdog is the last line: kick it in the main loop only, never in an ISR.

exercises.lol exercises.com.cn