UNIVERSITY ELECTRICAL ENGINEERING • EXERCISE SHEET

8.7 Debugging and integration

Name: _____ Class: _____ Date: _____ **Total: 22 marks**

KEY WORDS watchdog • debounce

Objective

Debug by bisection with instrumentation at the trusted/suspected boundary, kick watchdogs correctly, and structure a firmware project – every fix verified by re-running the failing case.

1 Key knowledge and worked examples

The bisection method

1. Reproduce with the smallest failing input. 2. Inspect at the boundary between trusted and suspected code (LED, breakpoint, UART print). 3. Fix ONE thing. 4. Re-verify. A well-placed observation halves the search each run – three observations locate a fault among eight modules.

Worked example: the intermittent hang

A device hangs after minutes. Hypotheses: stack overflow, an unhandled interrupt, an infinite loop, or hardware. The bisection: an LED toggled in the main loop stops at the hang → the main loop died but the interrupt LED keeps ticking → the fault is IN the main loop (not the ISR); a UART print of the loop's state variable shows it stuck at a "wait for sensor" value → the sensor I2C read never returns. Root cause: a missing timeout on the I2C wait. The fix is one line (a retry counter), verified by the hours-long soak that previously failed. Each observation halved the search – that is the method working.

Watchdogs and project structure

A **watchdog** is a countdown that resets the system unless kicked: kick it from the MAIN loop, never from an ISR (an ISR-kicked watchdog hides a dead main loop). The project separates drivers (register access), middleware (protocol stacks) and application; the linker script and startup code place flash/RAM/stack – a stack overflow is the classic "works for minutes" bug, and the fill-pattern check (0xDEADBEEF markers below the stack) makes it visible in one inspection.

2 Practice

2.1 A fault appears only when a UART message arrives during an ADC conversion. Name the likely interaction and the smallest reproducing test. [3]

2.2 Why is kicking the watchdog from an ISR wrong? [2]

2.3 A stack-overflow check places markers below the stack; after the hang, the markers are overwritten. What does that prove, and what is the next check? [3]

3 Exam-style questions

3.1 A data logger writes to flash every 10 s and sends a status line by UART every second; after a random 5–20 minutes the logger stops responding but the power LED stays on.

Set out a debugging plan: the three instrumentation points you would add (what each confirms or excludes), the order you would check them, and the hypotheses each observation would kill. The flash driver disables ALL interrupts during a page erase (2 s!) – explain how this can produce exactly the observed behaviour, and state the fix. [7]

3.2 A firmware project must integrate: a 1 kHz control loop (timer ISR), a 115 200-baud command parser (UART ISR), and a 5 Hz display update (main loop). The control computation takes 120 µs.

Assign the work to ISRs versus the main loop with reasons (what must run in the ISR, what must not). Compute the interrupt load (1 kHz × 20 µs ISR overhead + computation placement – state your placement). Define the communication between contexts (flags/buffers) and state the rule that makes a shared 1-byte flag safe without a mutex. State why printf in the control ISR is forbidden and what the ISR should do instead. [7]

Solutions

2.1

- Likely: a shared-resource or priority interaction – the UART ISR preempts the ADC completion handler mid-state.
- Smallest test: a script that fires the exact UART message while forcing an ADC conversion (both events at once, looped until it fails).

2.2

- An ISR keeps kicking even when the main loop is dead – the watchdog never fires, hiding the very hang it exists to catch.

2.3

- It proves the stack grew past its boundary at some point (an overflow occurred).
- Next: find WHAT grew it – record the stack high-water mark (fill the stack with a pattern and measure the untouched region) and inspect the deepest call chain / largest local arrays.

3.1

- Point 1: LED toggle in the main loop (does the main loop live?).
- Point 2: UART status of the write scheduler (which task was active at the freeze?).
- Point 3: a timestamp on the last completed flash write (correlates the hang with the erase window).
- Order: 1 → 2 → 3, each halving: dead main loop vs dead everything; active task; correlation with the flash schedule.
- Mechanism: the 2 s global interrupt disable blocks the UART ISR → the UART's receive buffer overruns during the erase → the UART driver wedges in an error state (or the missed characters desynchronise the parser) → the logger stops responding while power (main loop still ticking between erases... if the parser death blocks the scheduler, everything stops) – the observed "random minutes" is just the first erase that coincides with UART traffic.
- Fix: erase with interrupts ENABLED (a flash driver that locks per-page and tolerates reads/writes from RAM, or erase in the background with a completion interrupt); never block interrupts for seconds.

3.2

- The control MUST run in the timer ISR (or be triggered by it): its 1 ms deadline cannot survive the 200 ms display loop's latency. The computation (120 µs) runs inside the ISR or in a highest-priority task woken by it – either way, timer-locked.
- The UART parser: ISR receives bytes into a ring buffer; PARSING runs in the main loop (no hard deadline). Display: main loop (5 Hz). Load: 14% + 2.3% ≈ 16%, leaving ~84%.
- Shared flags: one byte, one writer – no mutex; declare it volatile; ring buffers use head/tail. printf stays out of the ISR (milliseconds): set a flag, print from the main loop.