Fault-Tolerant Embedded LED Controller on KL25Z + RTXv5

FRDM-KL25Z (ARM Cortex-M0+) · RTXv5 RTOS · buck-converter HBLED with PID controller · LCD + touchscreen UI · 15-fault injection catalog · NCSU ECE 560 (Embedded System Architectures, Prof. Dean) · Fall 2024 · Final Project

Topics: *ARM Cortex-M0+ embedded systems · CMSIS-RTOS2 / RTXv5 · mutex synchronization · event flags · state-machine synchronization · ADC server pattern · PID control loop · fault tolerance · watchdog timer · HardFault handler · register scrubbing · data validation · compile-time configuration via macros · Keil MDK-ARM toolchain*

> **The one-liner:** A two-part hardening of a real-time embedded LED controller running on an NXP KL25Z (ARM Cortex-M0+) under the RTXv5 RTOS. Part 1: synchronize multiple LCD-drawing threads + the ADC ISR + a low-priority waveform-display thread so the scope-style current trace renders correctly without corrupting LCD SPI transactions. Part 2: detect and recover from a 15-fault catalog injected by a built-in Thread_Fault_Injector - covering shared-data corruption, stack overflow, peripheral-clock disablement, RTOS-deadlock attacks, and IRQ disablement.
>
> **Why it's interesting:** The closest of my projects to production embedded firmware - a real microcontroller, a real RTOS (RTXv5 is the CMSIS-RTOS2 reference implementation that Keil ships), a real control loop driving real hardware, and a defense-in-depth approach to faults that matches what safety-aware firmware actually does. Worth noting: this was built with a project partner under the spec's two-person option. The work split is described below.

Scope - given starter code vs. what we built

The course supplies a working but unhardened system. Our deliverable was to add synchronization + fault tolerance on top.

Given (starter code)	Built (our modifications)
KL25Z + expansion shield hardware. Working buck-converter LED controller with PID compensator. LCD + touchscreen driver stack (ST7789). MMA8451 accelerometer driver. ADC server pattern multiplexing the ADC between the time-critical buck-converter sensing path and the touchscreen.	No hardware changes. Stayed inside the application/RTOS layer.
Five RTXv5 threads: `Thread_Read_Touchscreen`, `Thread_Draw_Waveforms`, `Thread_Draw_UI_Controls`, `Thread_Update_Setpoint`, `Thread_Read_Accelerometer`. Two ISRs: TPM0 (PWM + ADC trigger) and ADC0 (calls `Control_HBLED`). LCD mutex (`LCD_mutex`) acquired at high-level in both drawing threads.	Added a sixth thread (`Thread_WDT`, low priority) that periodically services the COP watchdog. Added `Mutex_Acq` wrapper with 200 ms timeout + delete-and-recreate on timeout. Added scope-sync state machine variable (`Initial_done`) and parallel RTOS event-flag path (`adcEventFlags`), selectable at compile time via `SYNC_W_RTOS` macro.
`Thread_Fault_Injector` (above-normal priority) that periodically injects faults from a 15-entry catalog: shared-data corruption (TR_Setpoint_High, TR_Flash_Period, TR_PID_FX_Gains), stack overflow, IRQ disablement, peripheral-clock disablement, MCU clock change, TPM overflow change, mutex hold/delete, message-queue fill, kernel lock, and priority-boost-then-infinite-loop.	Implemented handlers for every catalog entry: `HardFault_Handler` → `NVIC_SystemReset`; periodic scrubbing of `TPM0->MOD` / `NVIC` ADC0 enable / `SIM->SCGC6`; in-line data validation on `g_set_current_mA` in `Control_HBLED`; mutex timeout wrapper with delete + LCD_Create_OS_Objects recreate.
LCD critical-section requirement: three rectangle-transaction-sequence critical sections per LCD update, only one allowed at a time across threads.	Mutex usage parameterized via `USE_LCD_MUTEX_LEVEL` macro in `config.h`. After measurement, kept level 1 (high-level mutex around whole loop body) - the lower-level refactor's OS overhead didn't justify the marginal latency gain at the measured critical-section durations.

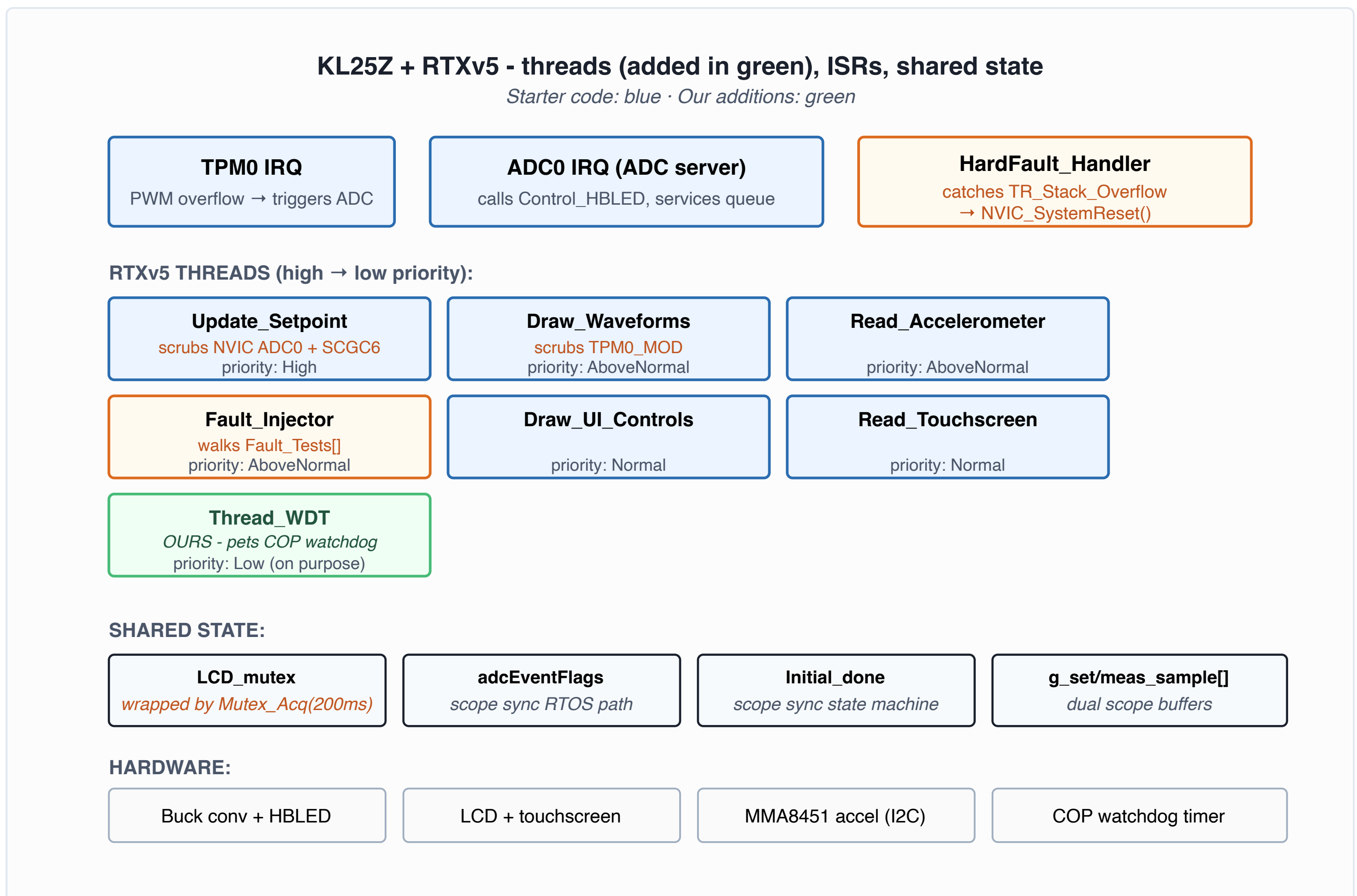
Architecture - KL25Z + RTXv5 + the buck-converter LED control loop

The system is a strobe-style LED driver with a touchscreen UI. The buck converter drives a high-brightness LED at an adjustable current setpoint. The LCD shows an oscilloscope-style trace of the requested setpoint

(blue) and measured current (yellow), plus numerical control fields. Tilting the board changes the strobe period via the accelerometer.

The full thread roster, priorities, ISRs, and shared state appear in the architecture diagram below. Two threads worth highlighting in prose: `Thread_Update_Setpoint` (high priority) generates the strobe square wave AND hosts our NVIC ADC0 + SIM_SCGC6 scrubbing. `Thread_WDT` (ours, lowest priority on purpose) periodically services the COP watchdog - if a higher-priority faulty thread monopolizes the CPU, our WDT thread can't run, the COP isn't petted, the watchdog resets the chip.

Two ISRs run alongside the threads (see the diagram below). `TPM0_IRQHandler` (PWM overflow) triggers ADC conversions every `SW_CTL_FREQ_DIV_FACTOR` overflows so the buck converter switches faster than the control loop runs. `ADC0_IRQHandler` implements the ADC server, calling `Control_HBLED` on time-critical buck-converter conversions and servicing queued touchscreen requests when there's time before the next priority conversion. `Control_HBLED` itself runs the PID compensator in fixed-point arithmetic, updates the PWM duty cycle, and (when scope sync permits) fills the next sample-buffer slot.



System architecture: 7 RTXv5 threads (one of ours - Thread_WDT in green at lowest priority) + 2 ISRs + custom HardFault handler. Orange annotations mark fault-handling code we added to existing threads. Shared state row shows the synchronization primitives my project partner built (LCD_mutex wrapper, adcEventFlags, Initial_done state machine).

Synchronization - LCD mutex + scope waveform display

LCD mutex - preventing rectangle-transaction-sequence interleaving

The LCD controller draws via three-command rectangle transaction sequences (Column Address Set → Row Address Set → Memory Write with pixel data). If two threads interleave LCD commands mid-sequence, the wrong pixels go to the wrong addresses and the display corrupts. Per the spec, there are exactly three such critical sections in the LCD-drawing code.

Two mutex placement strategies were evaluated, selectable via `USE_LCD_MUTEX_LEVEL` in `config.h`. **Level 1 (active in submission)** acquires `LCD_mutex` once at the top of each drawing thread's loop iteration and releases at the bottom - conservative, one acquire/release pair per iteration, but increases blocking time for the other drawing thread. **Level 2/3 (the 560-only refactor option)** acquires/releases only around each of the three individual critical sections, shortening max hold time to microseconds at the cost of 3× the OS mutex call overhead. After timing the critical-section durations with debug pulses on DBG_LCD_COMM_POS and comparing against the OS acquisition overhead, the lower-level approach didn't materially improve overall latency - sections were short enough that OS overhead would have dominated - so we kept level 1 active behind the macro.

Scope waveform sync - state machine OR event flags, selectable at compile time

The scope-style trace needs two synchronization properties: (1) trigger on a rising-edge of the setpoint pulse so the waveform display is stable rather than scrolling, and (2) prevent `Control_HBLED` (which runs from the ADC ISR) from corrupting the sample buffers while `Thread_Draw_Waveforms` is reading them. The 560 spec required implementing both a non-RTOS state-machine approach and an RTOS-event-flag approach, with the choice made by a `config.h` macro.

State machine path (`SYNC_W_RTOS = 0`): a static `Initial_done` state variable inside `Control_HBLED` cycles through three states. State 0 ("armed"): waiting for the setpoint to cross the trigger threshold low-to-high. On detection, state advances to 1 and starts filling buffers. State 1 ("filling"): writes each new ADC sample to `g_set_sample[]` / `g_meas_sample[]` until the buffers are full, then sets `stop_fill = 1` and advances to state 2. State 2 ("displaying"): does nothing until `stop_fill` is cleared back to 0 by `Thread_Draw_Waveforms` after it finishes drawing, then returns to state 0 for the next trigger. The drawing thread polls `stop_fill == 1` in its loop and draws when set.

RTOS event-flag path (`SYNC_W_RTOS = 1`): uses an `adcEventFlags` object. `Control_HBLED` calls `osEventFlagsSet(adcEventFlags, 0x01)` when the buffer fills. The drawing thread blocks on `osEventFlagsWait(adcEventFlags, 0x01, osFlagsWaitAny, NULL)` instead of polling; the OS unblocks it when the flag is set. After drawing, the thread clears the flag and the cycle restarts.

The compile-time selection is a textbook microarchitecture tradeoff: the state machine approach has zero OS overhead and is simpler to reason about, but burns CPU on polling. The event flag approach is cleaner

code (the thread blocks rather than spins) but pays OS scheduling overhead on every signal/wait. For a 24 kHz control loop on a Cortex-M0+, either is acceptable; production firmware would benchmark both and pick.

Fault management - the 15-entry catalog and our handlers

The starter `Thread_Fault_Injector` injects faults from a catalog defined as an enum in `fault.c`. Each fault models a real-world failure mode - data corruption, RTOS deadlock, peripheral clock loss, IRQ disablement, runaway thread. Our response strategy uses five distinct defensive patterns, layered so most faults have multiple defenses behind them.

Defense pattern	Where implemented	Faults caught
Watchdog timer thread - `Thread_WDT` pets the KL25Z's Computer Operating Properly (COP) watchdog at `osPriorityLow`. If the scheduler stops running this thread, the COP times out and resets the chip.	`threads.c` - `Thread_WDT` (~10 lines), called from `Create_OS_Objects`. COP initialized in `main()` before kernel start.	TR_Disable_All_IRQs (sets PRIMASK, blocks SysTick → scheduler dies), TR_osKernelLock (RTOS lock keeps current thread on CPU), TR_High_Priority_Thread (fault injector boosts to RealTime then infinite loops, starving everything below it including the WDT thread).
HardFault handler - overrides the default weak HardFault_Handler with a function that calls `NVIC_SystemReset()` for a clean processor reset.	`control.c` - `HardFault_Handler` (4 lines).	TR_Stack_Overflow (walks the stack pointer downward writing 0xeeeeeeee until it crashes off the end of allocated stack memory, triggering a HardFault exception).
Periodic register scrubbing - every iteration of a periodic thread, compare a critical register against its expected value and rewrite it if it's been corrupted.	`Thread_Draw_Waveforms` scrubs `TPM0->MOD`. `Thread_Update_Setpoint` scrubs `NVIC_ISER` ADC0 enable bit and `SIM->SCGC6` peripheral clock gates.	TR_Slow_TPM (writes `TPM0->MOD = 23456` - would slow the switching frequency by ~10x). TR_Disable_ADC_IRQ (NVIC_DisableIRQ on ADC0 - would silence the control loop). TR_Disable_PeriphClocks (zeros `SIM->SCGC6` - kills ADC, DAC, and several other peripherals).

Defense pattern	Where implemented	Faults caught
Mutex timeout + delete-recreate wrapper - `Mutex_Acq()` wraps `osMutexAcquire` with a 200 ms timeout. On timeout, the mutex is deleted via `osMutexDelete` and the LCD OS objects are recreated via `LCD_Create_OS_Objects`.	`control.c` - `Mutex_Acq`. Called by both drawing threads everywhere they previously called `osMutexAcquire(LCD_mutex, osWaitForever)`.	TR_LCD_mutex_Hold (the fault injector acquires the mutex and never releases it - without the timeout, the drawing threads would block forever on `osWaitForever`). TR_LCD_mutex_Delete (the fault injector deletes the mutex - any subsequent acquire call would fail; recreating via `LCD_Create_OS_Objects` restores it).
In-line data validation - bounds check on `g_set_current_mA` before using it to compute duty cycle.	`Control_HBLED` - the `if (g_enable_control && (g_set_current_mA < MAX_G_SET_CURRENT_MA_VALUE))` gate around the compensator branch. Out-of-range values are clipped (the control branch is skipped, leaving the previous duty cycle in place).	TR_Setpoint_High (overwrites `g_set_current_mA = 1000`, which would saturate the duty cycle to 100% and potentially over-drive the LED).

Design decisions and trade-offs

Three limitations of the project, and the reasoning behind each.

1. **Built with a project partner.** The 560 spec explicitly allowed two-person teams and we used that option. I led the fault-tolerance side (WDT thread, HardFault handler, scrubbing in `Thread_Update_Setpoint` and `Thread_Draw_Waveforms`, in-line data validation in `Control_HBLED`) and the report. My partner led the synchronization side (the `Mutex_Acq` timeout wrapper, the scope-sync state machine in `Control_HBLED`, the `adcEventFlags` event-flag path). We paired closely on architecture decisions, and both contributed to the final report.
2. **The submitted `Fault_Tests[]` array is empty on purpose.** Per the spec's submission requirements, the array should contain only the tests the team handles, with the rest commented out. During development we tested each fault individually by uncommenting one at a time, verifying with the scope

on DBG_FAULT_POS, then re-commenting it. The final-submission state has all 15 commented out with the array terminated by `TR_End`. The handler code for all 15 is in source - verifiable by reading `threads.c`, `control.c`, and the `HardFault_Handler`.

3. **LCD mutex stayed at high-level (level 1) after measurement.** The 560 spec required evaluating whether a lower-level mutex refactor - acquiring/releasing only around the three individual critical sections rather than the whole loop iteration - was worth the additional OS overhead. After timing the critical-section durations with DBG_LCD_COMM_POS pulses, the per-critical-section OS mutex overhead would have dominated the marginal latency improvement at the LCD's SPI speed. The code is parameterized via `USE_LCD_MUTEX_LEVEL` in `config.h` - the lower-level paths exist in the design but level 1 is what shipped.

Common questions about this project

Question	Short answer
Who did what on this project?	Built with a project partner. I owned the fault-tolerance pieces: WDT thread, `HardFault_Handler`, the three scrubbing patterns in `Thread_Update_Setpoint` (NVIC ADC0 + SIM_SCGC6) and `Thread_Draw_Waveforms` (TPM0_MOD), and the in-line data validation in `Control_HBLED`. Plus the report writeup. My partner owned the synchronization side: the `Mutex_Acq` timeout wrapper, the `Initial_done` state machine in `Control_HBLED`, the `adcEventFlags` event-flag path. We paired on the integration and the `USE_LCD_MUTEX_LEVEL` / `SYNC_W_RTOS` macro design.
Why does the WDT thread sit at the lowest priority?	So the watchdog only gets petted when forward progress is happening. A faulty high-priority thread monopolizing the CPU starves the WDT thread, the COP isn't serviced, the watchdog resets the chip. Putting the WDT high would defeat the purpose - it would pet the dog even when nothing useful was running.
How does the scope sync state machine work?	Static `Initial_done` variable inside `Control_HBLED` cycles 0 → 1 → 2 → 0. State 0 waits for the setpoint to cross a trigger threshold low-to-high (oscilloscope-style triggering). State 1 fills the sample buffers on every ADC interrupt. State 2 stalls while `Thread_Draw_Waveforms` renders, then resets to state 0 for the next trigger. `stop_fill` is the shared flag between the ISR and the thread.
What happens on a TR_Stack_Overflow?	The fault reads the current stack pointer (PSP) and walks downward writing `0xeeeeeeee` in an infinite loop. Eventually it writes past the allocated stack region into protected memory, the processor takes a HardFault exception, and our `HardFault_Handler` calls `NVIC_SystemReset()` for a clean reset. A more sophisticated handler could log context and try to recover the offending thread, but full reset is the safe default for stack corruption.

Question	Short answer
What would change for a production system?	Three things. (1) Write-protect critical configuration registers after init, so peripheral-clock and TPM-MOD faults can't even land. (2) Add a second independent hardware watchdog (the KL25Z has only one COP; a real product uses an external chip as backup). (3) Replace in-line data validation with accessor functions + complementary-variable storage for safety-critical setpoints - the spec's report template walked through this pattern for `g_enable_control` and it's the production-grade approach. Plus MISRA C static analysis, unit tests on the handlers, and stack-watermark monitoring on top of the HardFault catch.