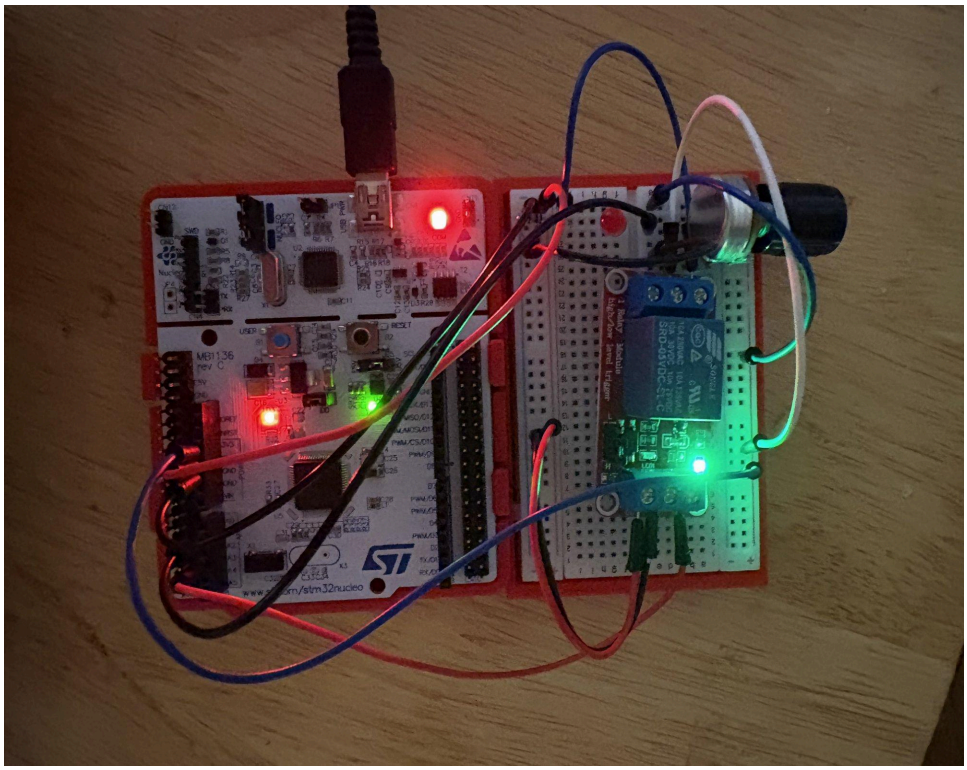
Deterministic Multi-Task Embedded Control Framework

STM32F401RE + FreeRTOS

Portfolio Project — Embedded Systems Architecture

Palmer Hutt | huttlabs.com | github.com/phutt7



STM32F401RE Nucleo with relay module, potentiometer, and status LEDs — normal operating state

1. Goal

Build a real-time embedded firmware system that demonstrates deterministic timing, static memory discipline, and structured RTOS task orchestration — the core competencies expected in modern embedded systems roles.

2. Project Objective

Design and implement a real-time embedded firmware framework that:

- Uses hardware timers for clock-driven ADC sampling
- Executes prioritized RTOS tasks with deterministic behavior
- Enforces static memory allocation with no dynamic heap usage
- Implements modular hardware abstraction and clear task boundaries
- Operates continuously under watchdog supervision

3. Target Hardware

- STM32F401RE Nucleo board (ARM Cortex-M4 @ 84 MHz, 512 KB Flash, 96 KB SRAM)
- Analog input — 10kΩ potentiometer on PA1 (Arduino header A1), simulating a sensor
- Digital output — AEDIKO 5V optocoupler relay module (high-trigger) on PB0 (Arduino header A3)
- Status indicators — onboard green LED LD2 (PA5, normal state), red LED via relay NO terminal (fault)
- UART telemetry — USART2 via ST-Link virtual COM port at 115200 baud

Sensor Selection Note — ACS712 Incompatibility

The ACS712 Hall-effect current sensor was evaluated but rejected. It requires a 5V supply and its output swings up to 5V, exceeding the STM32 ADC's 3.3V maximum. The potentiometer provides a clean 0–3.3V range across the full ADC scale (0–4095 counts). Recommended upgrade path: INA219 (I2C, 3.3V native).

Relay Module — Polarity & Pin Mapping

The AEDIKO relay module is high-trigger: GPIO HIGH energizes the coil, GPIO LOW releases it. The relay IN pin connects to PB0 (Arduino header A3). PB0 is initialized HIGH in MX_GPIO_Init to keep the relay de-energized during boot before the RTOS scheduler starts.

4. Software Architecture

4.1 Layered Structure

- HAL Layer: ADC, Timer, GPIO, UART
- Driver Layer: Circular buffer, ADC driver, relay driver
- Application Layer: Tasks, control logic, logging

4.2 Data Flow

1. TIM2 ISR fires at 1 kHz → triggers ADC conversion
2. ADC sample written to static ring buffer
3. Task A drains ring buffer → computes 16-sample rolling average
4. Task A pushes average to FreeRTOS message queue
5. Task B reads queue → compares to threshold → drives relay and LED
6. Task C reads shared volatile → transmits telemetry every 500 ms

5. System Architecture

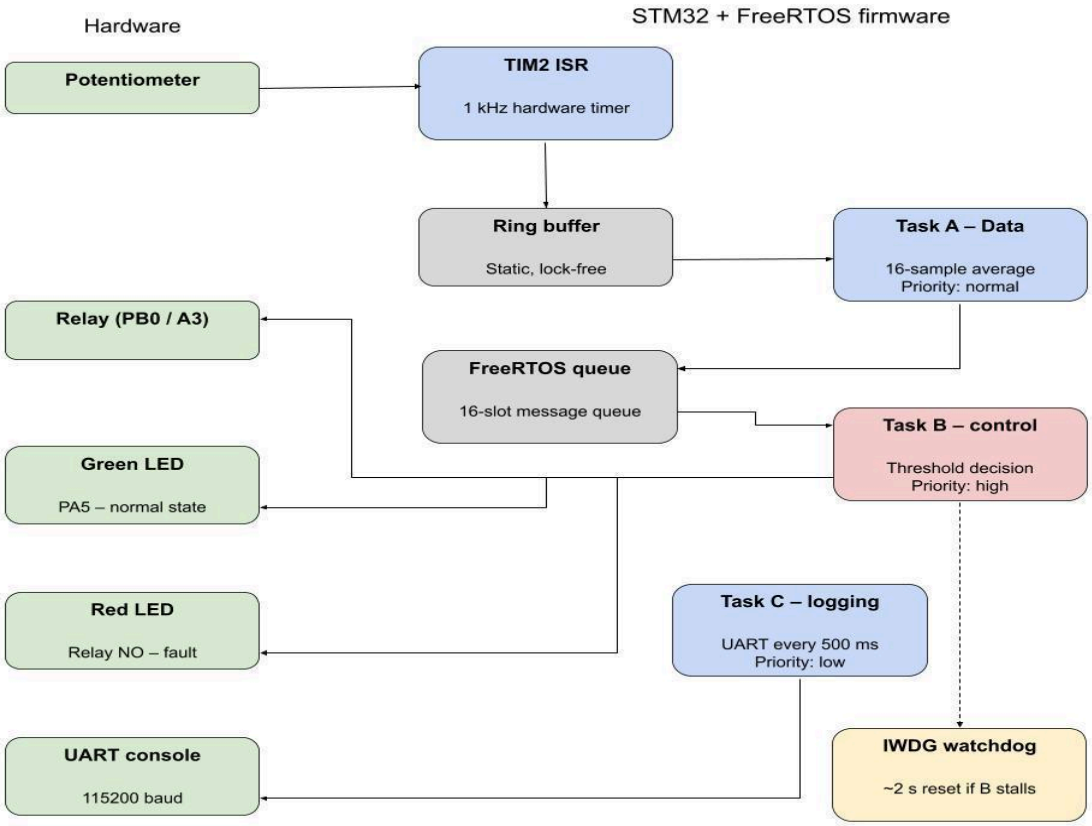


Fig. 1: System Block Diagram — STM32 + FreeRTOS

6. Key Features

6.1 Deterministic Timer-Driven ADC Sampling

Demonstrates hardware timer configuration, interrupt-driven design, deterministic 1 kHz sampling, and ISR minimalism discipline.

Specification:

- TIM2 configured at 1 kHz (Prescaler 83, Period 999 at 84 MHz)
- Timer interrupt triggers ADC conversion and reads ADC register directly
- Sample stored in statically allocated ring buffer (64 samples, power-of-2)
- ISR execution time < 10% of timer period
- No blocking or complex logic inside ISR

Constraint: ISR must remain bounded and non-blocking at all times.

6.2 RTOS Multi-Task Scheduling

Demonstrates FreeRTOS integration, task prioritization strategy, queue-based inter-task communication, and strict separation of concerns.

Task A — Data Processing (Priority: Normal)

- Drains ring buffer on each scheduler tick
- Computes 16-sample rolling average
- Detects stuck sensor values (10 consecutive identical readings → safe state, 0xDEAD sentinel)
- Pushes average to FreeRTOS message queue

Task B — Control Logic (Priority: High)

- Blocks on queue with 100 ms timeout
- Compares average to ADC_THRESHOLD (2048, mid-scale of 12-bit ADC)
- Above threshold → green LED on, relay off. Below → green LED off, relay energizes (fault state)
- Kicks IWDG watchdog every cycle — system resets within ~2 s if Task B stalls

Task C — UART Logging (Priority: Low)

- Reads shared volatile lastADCAverage
- Transmits telemetry every 500 ms: ADC:[value] OVFL:[count] STK:[headroom]
- Stack high-water mark monitored in real time — stable value confirms no stack leak

6.3 Static Memory & Safety Discipline

- No malloc / new anywhere in the system
- No STL containers
- All buffers fixed-size at compile time
- Explicit ownership model per module
- Independent watchdog enabled

7. Memory Allocation Map

Region	Size	Contents	Notes
Flash	128 KB	Firmware code, const tables	No dynamic allocation
RAM (Static)	8 KB	Task stacks, ring buffer, queues	All compile-time allocated
Ring Buffer	256 bytes	64 samples × 2 bytes	Owned by ADC driver
Task Stacks	1 KB each	Fixed stacks	No overflow allowed
Watchdog	32 bytes	WDG registers	Kicked by Task B

Table 1: Static Memory Allocation Map

8. Project Rules

- No dynamic memory allocation
- No delay() calls
- No polling loops for timing
- ISR must remain bounded and minimal
- All modules separated: HAL → Driver → Application

9. Success Criteria

- ADC sampling jitter < 5%
- ISR execution < 10% of timer period
- No missed watchdog kicks
- No heap usage
- System runs 24 hours without reset

10. Design Rationale

10.1 Why 1 kHz Sampling?

1 kHz simulates industrial control loop frequencies common in power electronics and monitoring systems. It provides meaningful timing jitter analysis while remaining computationally lightweight for an STM32F401RE running FreeRTOS.

10.2 Why No Dynamic Allocation?

Heap fragmentation and non-deterministic allocation latency violate real-time constraints. All buffers, task stacks, and queues are statically allocated at compile time.

10.3 Why Is Control Logic Highest Priority?

Task B directly influences physical outputs and must preempt all lower-priority tasks to ensure output changes occur within bounded latency.

10.4 Why No Queue Operations in the ISR?

Queue operations increase ISR execution time and risk priority inversion. The ISR is intentionally minimal: read ADC register → store sample in ring buffer → exit.

10.5 Why Is the Watchdog Tied to Task B?

Task B is the final control decision layer. The watchdog is refreshed only after successful Task B completion, ensuring any stall triggers automatic recovery.

11. Fault Handling Strategy

11.1 ADC Stuck Value

If identical ADC readings persist for 10 consecutive windows, a sensor fault is declared. overflowCount is set to 0xDEAD and relay forced to safe state.

11.2 Queue Overflow

If Task B cannot consume items fast enough, overflowCount increments. Oldest sample discarded to preserve real-time behavior.

11.3 Missed Execution Window

Repeated Task B execution failures result in a watchdog-triggered system reset.

11.4 Task Starvation

FreeRTOS stack high-water mark monitoring active. Stable headroom (130 words confirmed) indicates no stack growth.

11.5 Watchdog Reset Behavior

- MCU performs hardware reset
- Relay defaults to safe OFF state via startup pin initialization
- System resumes normal RTOS operation

12. Timing Budget

STM32F401RE @ 84 MHz. 1 kHz timer period = 1 ms. ISR budget: < 10% → < 100 µs.

Component	Max Execution Time	Budget %
Timer ISR	< 50 µs	< 5%
Task A	< 150 µs	Bounded
Task B	< 100 µs	Highest priority
Task C	Non-critical	Best effort

Table 2: Timing Budget

13. Risks & Mitigations

- ISR too long → minimize logic, validate with timing measurements
- Queue overflow → bounded processing time, overwrite oldest sample strategy
- UART blocking → low-priority task with ring buffer decouples logging from control
- Task starvation → FreeRTOS stack high-water mark monitoring, safe-state fallback

14. Implementation Status

Fully implemented and verified on an STM32F401RE Nucleo board. A potentiometer on PA1 simulates a sensor input. An AEDIKO relay module on PB0/A3 serves as the digital output. The firmware is hardware-agnostic — swapping to a real current sensor requires only a pin and channel change.

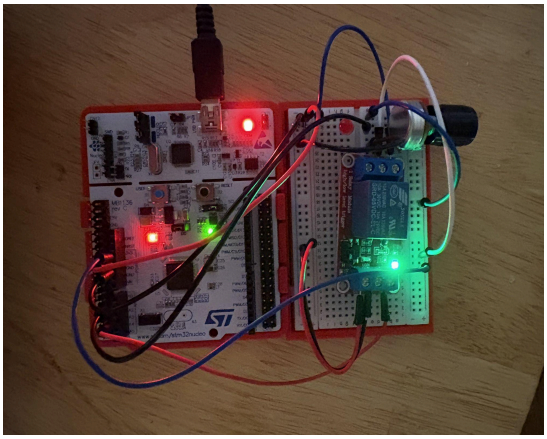


Fig. 7: Normal state — green LED on, relay off (ADC > 2048)

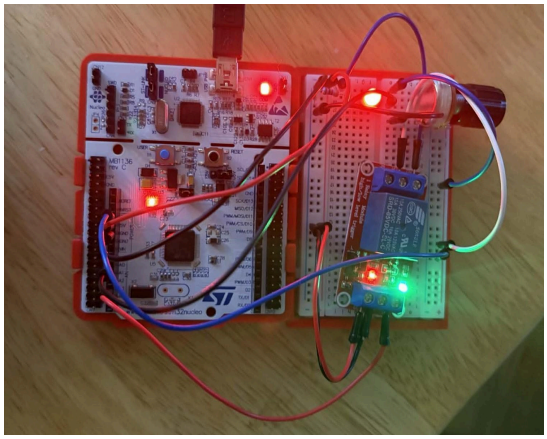


Fig. 8: Fault state — relay energized, red LED active (ADC ≤ 2048)

14.1 Completed Features

- TIM2 at 1 kHz — Prescaler 83, Period 999. ISR minimal and non-blocking.
- Static ring buffer (64 samples) — power-of-2, no dynamic allocation.
- Task A — drains buffer, 16-sample average, stuck-value detection, pushes to queue.
- Task B — reads queue, drives relay/LED at 2048 threshold, kicks IWDG.
- Task C — UART telemetry every 500 ms.
- IWDG — Prescaler 16, Reload 4095, ~2 s timeout, tied exclusively to Task B.

14.2 Verified Outputs

Live UART telemetry confirmed ADC values tracking potentiometer position across the full 0–4095 range. Green LED and relay transitions verified at the 2048 threshold. Stack headroom stable at 130 words. Overflow counter remained 0 under normal operation. Stuck-value sentinel (0xDEAD) triggers correctly on disconnected input.

14.3 UART Telemetry — Live Console Output

The following screenshots show the live serial output captured from the running system on COM5 at 115200 baud.

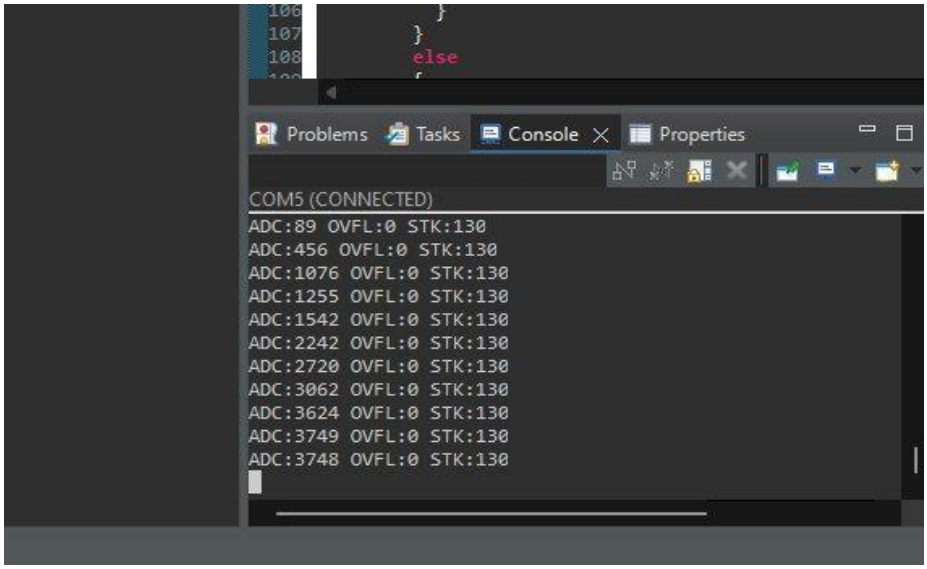


Fig. 9: ADC crossing the 2048 threshold — values climbing from 89 to 3749, OVFL:0, STK:130 stable throughout

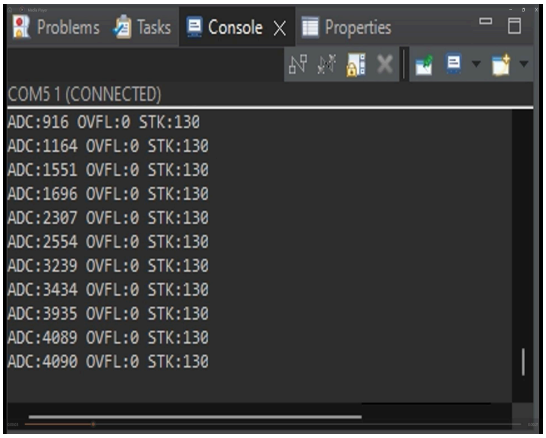


Fig. 10: Fault state telemetry — ADC 916–1696, climbing toward threshold

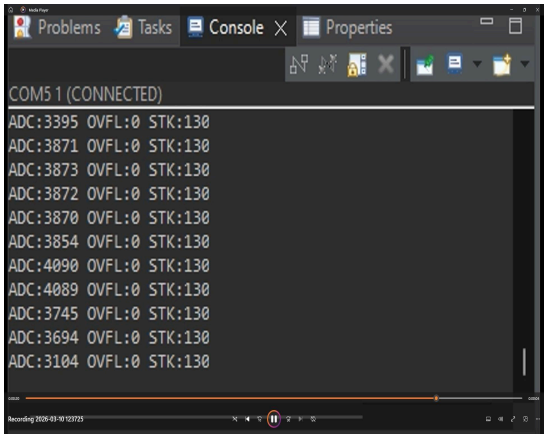


Fig. 11: Above-threshold telemetry — ADC 3395–4090, system in normal state

Key observations across all three captures:

- OVFL:0 across all readings — queue never backed up, Task B always keeping pace
- STK:130 locked constant — no stack growth, no memory leak over time
- Full ADC range 0–4095 cleanly represented across the potentiometer sweep
- Threshold crossing visible mid-scroll in Fig. 9 — relay trips at 2048

14.4 UART Telemetry Format

```
ADC:412  OVFL:0 STK:130
ADC:1843 OVFL:0 STK:130
ADC:2901 OVFL:0 STK:130
```

- ADC — 16-sample rolling average (0–4095)
- OVFL — queue overflow counter (0 = healthy, 0xDEAD = sensor fault)
- STK — Task C stack headroom in words (stable = no stack leak)

15. Hardware Upgrade Path

- Change ADC_CHANNEL_1 to match the current sensor input pin
- Update GPIO pin assignment for the relay driver
- Adjust ADC_THRESHOLD to the current trip point in ADC counts
- Recommended sensor upgrade: INA219 (I2C, 3.3V native) replaces ACS712

All task logic, timing, memory discipline, and watchdog behavior remain unchanged.

16. Business-Level Explanation

This system models a production-grade industrial control loop where deterministic timing and predictable memory behavior are critical. It demonstrates clean separation of data acquisition, control decision-making, and system monitoring — preventing functional interference between subsystems.

From an applications engineering perspective: the system reads a field sensor, makes a control decision, and drives an output — while logging its own health continuously. Every design decision optimizes for predictable behavior under real-world operating conditions.

Appendix A — System Diagrams

A1. Interrupt Flow Diagram

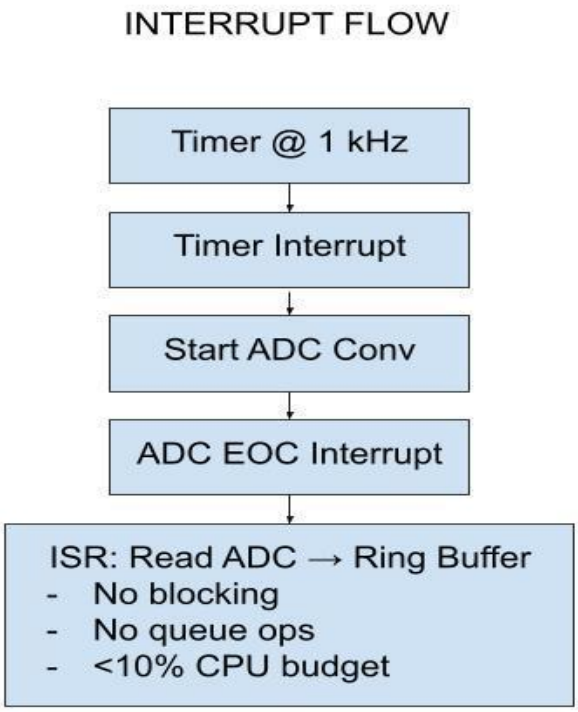


Fig. 2: Interrupt Flow

A2. Task Priority Diagram

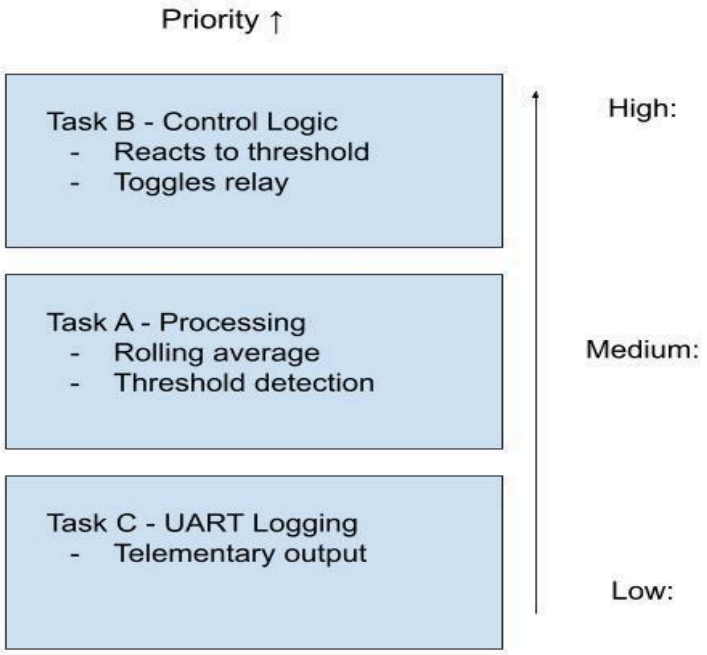


Fig. 3: Task Priority Levels

A3. Task A — Data Processing

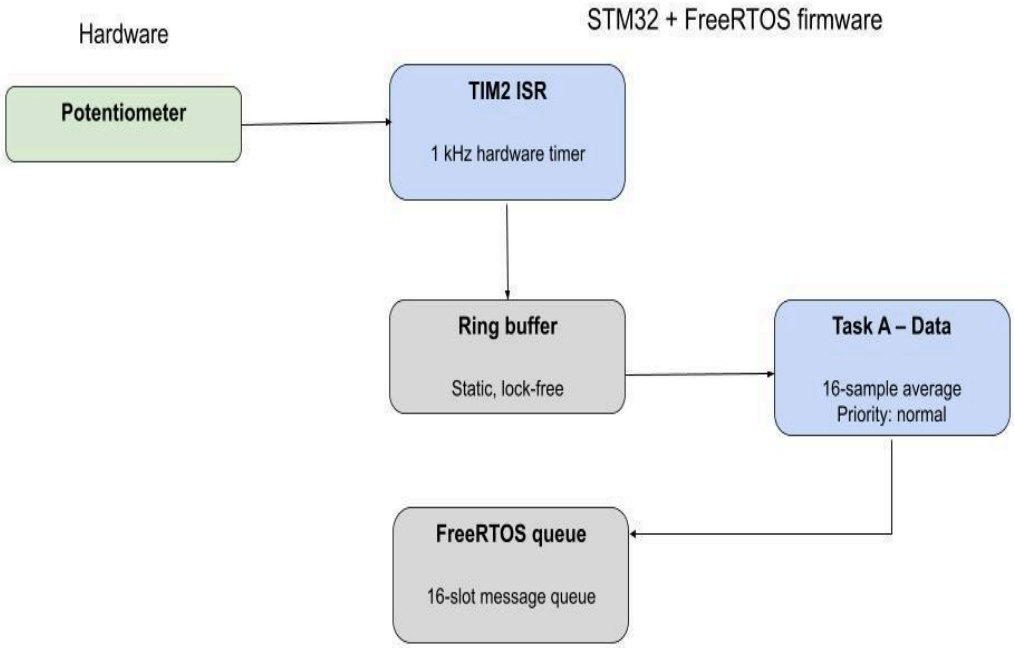


Fig. 4: Task A Block Diagram

A4. Task B — Control Logic

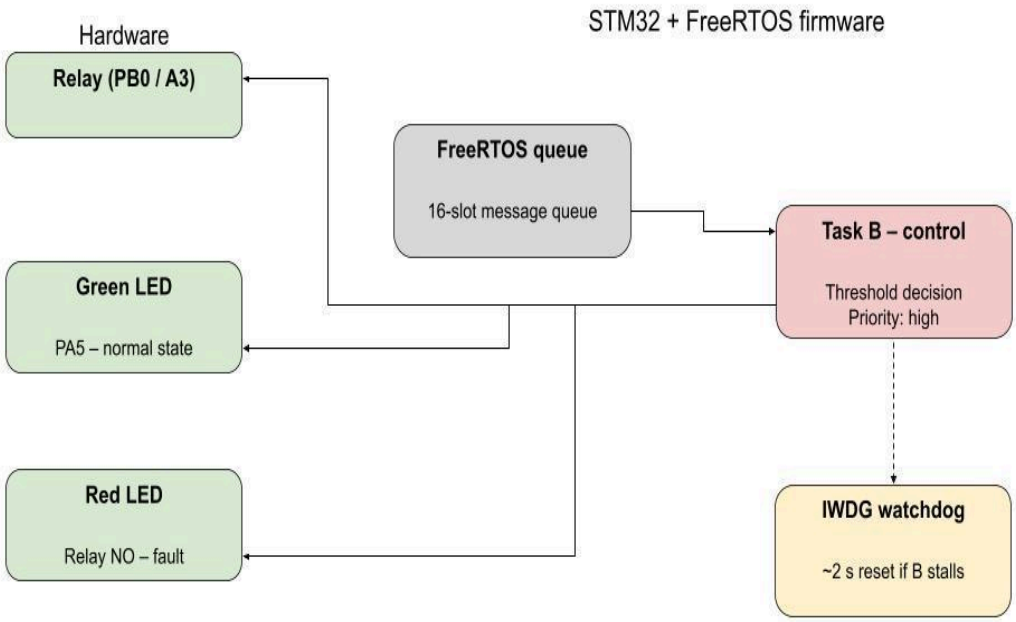


Fig. 5: Task B Block Diagram

A5. Task C — UART Logging

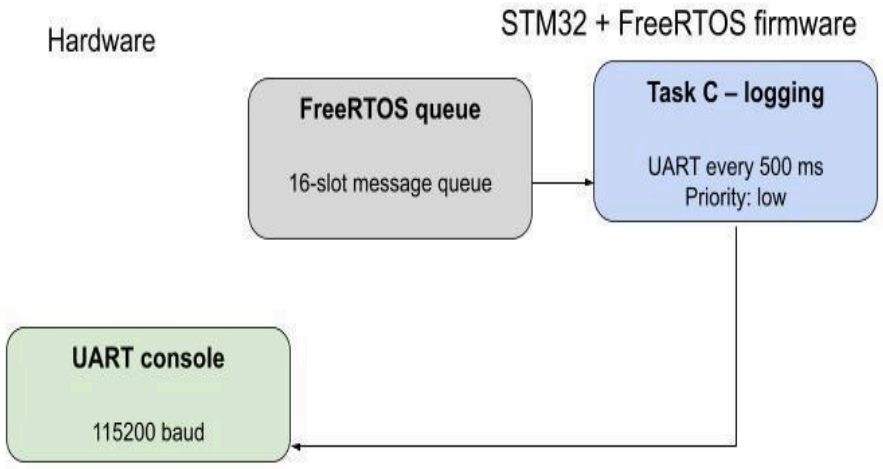


Fig. 6: Task C Block Diagram

Appendix B — CubeMX Configuration

Peripheral	Setting	Value	Reason
TIM2	Prescaler	83	84 MHz / 84 = 1 MHz tick
TIM2	Period	999	1 MHz / 1000 = 1 kHz ISR
ADC1	Conversion mode	Single	Triggered from ISR
ADC1	EOC selection	Single conv	Matches poll usage
ADC1	Sampling time	3 cycles	Fast, adequate for pot
IWDG	Prescaler	16	LSI ~32 kHz / 16 = 2 kHz
IWDG	Reload	4095	~2 s timeout
SYS	Timebase	TIM1	FreeRTOS owns SysTick
FreeRTOS	API	CMSIS-RTOS V2	
FreeRTOS	Heap	heap_4	Best-fit with coalescence

Table B1: CubeMX Peripheral Configuration

Appendix C — Source Code

Full source is available on GitHub at github.com/phutt7. The four core files are reproduced here for reference.

C1. ring_buffer.h — Ring Buffer Header

```
#ifndef RING_BUFFER_H
#define RING_BUFFER_H
#include <stdint.h>
#include <stddef.h>

#define RING_BUFFER_SIZE 64

typedef struct {
    uint16_t buffer[RING_BUFFER_SIZE];
    volatile uint32_t head;
    volatile uint32_t tail;
} RingBuffer_t;

void     RingBuffer_Init  (RingBuffer_t *rb);
void     RingBuffer_Write (RingBuffer_t *rb, uint16_t value);
uint8_t  RingBuffer_Read  (RingBuffer_t *rb, uint16_t *value);
uint32_t RingBuffer_Count (RingBuffer_t *rb);

#endif
```

C2. ring_buffer.c — Ring Buffer Implementation

```
#include "ring_buffer.h"

void RingBuffer_Init(RingBuffer_t *rb)
{
 rb->head = 0;
 rb->tail = 0;
}

void RingBuffer_Write(RingBuffer_t *rb, uint16_t value)
{
    uint32_t next = (rb->head + 1) % RING_BUFFER_SIZE;
    if (next != rb->tail) {
        rb->buffer[rb->head] = value;
        rb->head = next;
    }
    // If full, silently discard oldest sample
}

uint8_t RingBuffer_Read(RingBuffer_t *rb, uint16_t *value)
{
    if (rb->tail == rb->head) return 0; // empty
    *value = rb->buffer[rb->tail];
    rb->tail = (rb->tail + 1) % RING_BUFFER_SIZE;
    return 1;
}

uint32_t RingBuffer_Count(RingBuffer_t *rb)
{
    if (rb->head >= rb->tail)
        return rb->head - rb->tail;
```

```
    return RING_BUFFER_SIZE - rb->tail + rb->head;
}
```

C3. freertos.c — Task Implementations (A, B, C)

```
/* Task A — Data Processing (Normal Priority) */
void StartDefaultTask(void *argument)
{
    uint16_t sample=0; uint32_t sum=0,count=0;
    uint16_t average=0,lastAverage=0; uint8_t stuckCount=0;
    for(;;) {
        while (RingBuffer_Read(&adcRingBuffer, &sample)) {
            sum+=sample; count++;
            if (count>=16) {
                average=(uint16_t)(sum/count);
                lastADCAverage=average; sum=0; count=0;
                if (average==lastAverage) {
                    stuckCount++;
                    if (stuckCount>=10) {
                        HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_RESET);
                        overflowCount=0xDEAD; stuckCount=0;
                    }
                } else { stuckCount=0; }
                lastAverage=average;
                if (osMessageQueuePut(adcQueueHandle,&average,0,0)!=osOK)
                    overflowCount++;
            }
        }
        osDelay(1);
    }
}

/* Task B — Control Logic (High Priority) */
void StartControlTask(void *argument)
{
    uint16_t average=0;
    for(;;) {
        if (osMessageQueueGet(adcQueueHandle,&average,NULL,100)==osOK) {
            if (average>ADC_THRESHOLD) {
                HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_SET);
                HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET); // relay OFF
            } else {
                HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_RESET);
                HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_SET);   // relay ON
            }
        }
        HAL_IWDG_Refresh(&hiwdg); // kick watchdog
    }
}

/* Task C — UART Logging (Low Priority) */
void StartLoggingTask(void *argument)
{
    char msg[64];
    for(;;) {
        UBaseType_t stackA=uxTaskGetStackHighWaterMark(NULL);
        sprintf(msg,"ADC:%u OVFL:%lu STK:%u\r\n",
                lastADCAverage,overflowCount,(unsigned int)stackA);
        HAL_UART_Transmit(&huart2,(uint8_t*)msg,strlen(msg),100);
        osDelay(500);
    }
}
```

C4. stm32f4xx_it.c — TIM2 ISR

```
/* stm32f4xx_it.c — TIM2 ISR */
#include "ring_buffer.h"
extern RingBuffer_t adcRingBuffer;
extern ADC_HandleTypeDef hadc1;

void TIM2_IRQHandler(void)
{
    HAL_ADC_Start(&hadc1);
    if (HAL_ADC_PollForConversion(&hadc1,0)==HAL_OK) {
        uint16_t sample=(uint16_t)HAL_ADC_GetValue(&hadc1);
        RingBuffer_Write(&adcRingBuffer,sample);
    }
    HAL_TIM_IRQHandler(&htim2);
}
```

Complete main.c including peripheral initialization is available in the GitHub repository.