

Building a Real-Time System at Your Dream Company

Scenario Overview

Imagine you've landed your dream job as a software engineer at a tech company in Central Florida. It could be a cutting-edge startup or an established firm – perhaps one of the innovative companies in the Orlando region. The company has tasked you with developing a **real-time system** for one of its key projects. This means you'll be dealing with software where **timing is as critical as correctness**: the system's functions must not only produce correct outputs, but must do so within strict time deadlines. Failure to meet those timing requirements could degrade the product's performance or even cause a system failure, depending on how critical the task is.

Company Context and Examples

Your choice of company and industry will shape the kind of real-time system you work on. In Central Florida's tech hub, there are many exciting possibilities. For example, you might envision working at:

- **Aerospace/Defense (Orlando):** Perhaps Lockheed Martin or a simulation startup, developing a flight control or pilot training simulator. These systems demand high responsiveness – a fighter jet's control software or an augmented reality (AR) training system for pilots must react instantly to inputs.
- **Autonomous Vehicles (Orlando/Melbourne):** A company like Luminar Technologies (an Orlando-born startup) builds LiDAR sensors and perception software for self-driving cars [Links to an external site.](#). Here you'd handle real-time sensor processing – the car must “see” pedestrians, other cars, and obstacles in real time to drive safely. Missing a deadline in processing could mean a collision, so many of these tasks are safety-critical.
- **Theme Park Systems (Orlando):** Imagine working for Disney or Universal on roller coaster **ride control systems**. These involve real-time control of motors, brakes, and safety sensors. For instance, a roller coaster's control system must synchronize sensors and actuators with precise timing to ensure a smooth and safe ride. An underlying real-time operating system (RTOS) helps guarantee **instantaneous reactions** to emergency conditions (e.g. an e-stop button or sensor trigger) so that the ride can be halted immediately if needed [Links to an external site.](#). This is absolutely vital for rider safety.
- **Healthcare Tech (Florida):** You might be at a medical device company developing a patient monitoring system or an insulin pump. Such devices often have tasks with strict timing – e.g. delivering a dose at the right moment or sounding an alarm within

seconds of detecting an anomaly. The software in a heart pacemaker is a classic example of a **hard real-time system** – missing its timing deadlines could be life-threatening.

- **Gaming/AR/VR (Orlando):** Perhaps you join a local AR/VR startup or a game studio. Orlando's tech scene includes companies like **Red 6**, which integrates AR into flight training, and game developers. In these domains, the system (be it a video game or an AR headset) also operates in real time: the graphics must render each frame quickly to keep up with user actions. A delay might only cause a drop in frame rate or a brief glitch rather than total failure, so these tend to be **soft real-time** tasks.

Your Engineering Scenario

You've just been hired by a Central-Florida tech firm – **again your choice**. The team's first directive - show us what you know about real time systems by developing a proof of concept RTS relevant to the company that includes a few elements shown below. **"Build a working proof-of-concept in Wokwi that proves we can meet our real-time deadlines."**

You'll design the embedded firmware, wire up the virtual hardware, and demonstrate predictable timing.

Think of this as an **internal engineering milestone**: a prototype the CTO will green-light only if it works *and* you can explain why it's safe on deadline-critical paths.

Prototype Requirements (all implemented in Wokwi)

Area	What you must include	Notes
Hardware	ESP32, ≥ 2 external LEDs, ≥ 1 sensor,	Wire & label everything in
Sketch	≥ 1 momentary input	Wokwi's diagram.json.
	≥ 4 FreeRTOS tasks <i>plus</i> ≥ 1 ISR ; at	Map each to a company use-case
Tasks & ISRs	least 1 task must take variable amount of time	(e.g., "Brake-sensor ISR", "LiDAR frame task").
Timing	Declare a period/deadline for every task; mark Hard vs Soft in code comments	Hard miss = failure for your product scenario.
Synchronization	Use ≥ 2 distinct mechanisms (mutex, queue, binary sem, critical section...)	Show why each is needed.
Inter-task & External Comms	One internal channel (queue, event group, etc.) and one outward link (UART, Wi-Fi, I ² C, ...)	External link can talk to Wokwi serial monitor or a simple web page.

Determinism Proof Log timestamps or blink a “heartbeat” LED; show all hard deadlines are met
Use [Wokwi logic-analyzeLinks to an external site.r](#) [full credit] or serial time-stamps [partial credit]

Company Context In variables / comments / README:
tie every task to the chosen company’s product
Classify tasks as **Hard/Soft**

Engineering Analysis

(≈ 4-6 sentences each in README)

- Scheduler Fit:** How do your task priorities / RTOS settings guarantee every **H** task’s deadline in Wokwi? Cite one timestamp pair that proves it.
- Race-Proofing:** Where could a race occur? Show the exact line(s) you protected and which primitive solved it.
- Worst-Case Spike:** Describe the heaviest load you threw at the prototype (e.g., sensor spam, comm burst). What margin (of time) remained before an **H** deadline would slip?
- Design Trade-off:** Name one feature you *didn’t* add (or simplified) to keep timing predictable. Why was that the right call for your chosen company?

AI Tool Policy (use if you wish)

Your company has its own internal GenAI system - because they don't want their sensitive data ingested and possibly reused - also there's liability, if you use genAI code, you need to document it - the lawyers need to be prepared if ever called to court - so be careful how you use AI to assist you and document it when you do.

- Allowed for brainstorming or code *snippets*.
- **Must** list each chat/url in README AND inline with your code.
- You own all final timing proofs—don’t paste code you can’t explain.

Deliver a prototype the CTO can run, inspect, and trust. Good luck building real-time greatness!

Submit

Item	Upload / Provide
Wokwi URL	Public link to your full simulation.
Project ZIP	Save → Download ZIP and attach for grading backup.
Source Code	Clean, commented; compiles & runs inside Wokwi.

Concurrency Diagram	Boxes = tasks/ISR, arrows = queues/semaphores. Label periods (tasks/accesses) & Hard/Soft task.
README.md	<ul style="list-style-type: none"> • 75-word company synopsis & why RT • Task table (period, H/S, consequence) • Short answers to analysis prompts (below) • AI-usage disclosure + links if used
90-sec Demo Video	Screen-capture of Wokwi proving deadlines met (voiceover required).

Rubric

Category	Pts
Working Wokwi Prototype	40
Real-Time Design Quality	20
Concurrency Diagram & Code Clarity	10
Analysis Insight	15
Professional Context & Originality; honest AI disclosure	10
Demo Video Quality	5

Code:

```

/*
Application: 06 - Final
Theme: Theme Park Systems (Orlando)
Author: Blake Whitaker
UCFID: 543877
Google Gemini used for some logic and commenting
*/

// Header files
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "driver/gpio.h"
#include "esp_timer.h"
#include "rom/ets_sys.h"

// Mapping hardware components to GPIO pins on the ESP32
#define LED_SYSTEM_POWER GPIO_NUM_5 // Yellow LED for the power/heartbeat indicator.
#define LED_EMERGENCY_BRAKE GPIO_NUM_4 // Red LED for the fault/emergency brake indicator.

```

```

#define LED_ALL_CLEAR GPIO_NUM_19 // Green LED for the 'all clear' and ready to restart indicator.
#define BUTTON_EMERGENCY_STOP GPIO_NUM_18 // Emergency brake and system restart.
#define PROXIMITY_SENSOR_TRIG_PIN GPIO_NUM_17 // Trigger (output) pin for the ultrasonic sensor.
#define PROXIMITY_SENSOR_ECHO_PIN GPIO_NUM_16 // Echo (input) pin for the ultrasonic sensor.

// System Constants
#define PROXIMITY_THRESHOLD_CM 30 // The safety threshold; @10mph->need ~1.5ft; @25mph-> ~10.5 ft; @40->
~27ft.
#define BUTTON_DEBOUNCE_TIME_MS 200 // The time in milliseconds to ignore subsequent button presses to prevent
'bouncing'.

// Global variables to hold references to the RTOS objects we create
SemaphoreHandle_t sem_emergency_stop_button; // Handle for the binary semaphore signaled by the E-Stop ISR.
SemaphoreHandle_t sem_train_proximity_event; // Handle for the binary semaphore signaled by the sensor task.

// State Machine
// A custom data type (enum) to represent the possible states of the ride system.
typedef enum {
RIDE_ALL_CLEAR, // Normal operation.
HALTED_BY_PROXIMITY, // Halted due to a proximity obstruction.
HALTED_BY_ESTOP, // Halted due to a human operator pressing the restart button.
AWAITING_RESTART // Obstruction cleared, waiting for operator confirmation to restart.
} RideStatus;

// 'volatile' keyword tells the compiler that a variable's value can be changed by an external source
// (like another task or an ISR) and prevents optimizations that could lead to using a stale value.
volatile RideStatus ride_status = RIDE_ALL_CLEAR; // The main global variable that holds the current state of the ride.
volatile bool is_train_in_zone = false; // A flag indicating if an obstruction is currently within the threshold.
volatile int current_proximity_cm = 999; // Stores the latest proximity reading from the sensor task.
volatile int64_t last_isr_time_us = 0; // Stores the timestamp of the last valid E-Stop interrupt for debouncing.

// Forward Declarations for Tasks & an ISR
void system_power_monitor_task(void *pvParameters);
void train_sensor_monitor_task(void *pvParameters);
void status_output_task(void* pvParameters);
static void IRAM_ATTR gpio_isr_handler(void* arg);

// [Hard Real-Time] Processes safety events and manages the ride state machine logic.
void ride_control_handler_task(void *pvParameters) {
while (1) { // Loop forever to continuously manage the ride's state.
// State Transition Logic

// 1. A proximity event can halt a ride that is running OR awaiting restart.
if ((ride_status == RIDE_ALL_CLEAR || ride_status == AWAITING_RESTART) &&
xSemaphoreTake(sem_train_proximity_event, 0) == pdTRUE) {
ride_status = HALTED_BY_PROXIMITY;
gpio_set_level(LED_EMERGENCY BRAKE, 1);
gpio_set_level(LED_ALL_CLEAR, 0);
}
// 2. An E-Stop event is processed based on the current ride status.
if (xSemaphoreTake(sem_emergency_stop_button, 0) == pdTRUE) { // Check if the E-Stop ISR gave the semaphore.
}
}
}
```

```

switch (ride_status) { // Use a switch statement to handle the logic for each possible state.
// If ride is running or halted by proximity, an E-Stop press engages the E-Stop.
case RIDE_ALL_CLEAR:
case HALTED_BY_PROXIMITY:
ride_status = HALTED_BY_ESTOP;
gpio_set_level(LED_EMERGENCY_BRAKE, 1);
gpio_set_level(LED_ALL_CLEAR, 0);
break;
// If ride is E-Stopped or waiting for a proximity-halt reset, a button press performs the reset.
case HALTED_BY_ESTOP:
case AWAITING_RESTART: // <<-- This case is now correctly grouped with the reset logic.
if (!is_train_in_zone) { // Final safety check: track must be clear to restart.
ride_status = RIDE_ALL_CLEAR;
gpio_set_level(LED_EMERGENCY_BRAKE, 0);
gpio_set_level(LED_ALL_CLEAR, 1);
}
break;
}

// 3. Automatically transition from a PROXIMITY halt to AWAITING_RESTART if the obstruction clears.
if (ride_status == HALTED_BY_PROXIMITY && !is_train_in_zone) {
ride_status = AWAITING_RESTART;
}

vTaskDelay(pdMS_TO_TICKS(10)); // Pause task for 10ms to let other tasks run.
}

}

// [Soft Real-Time] Provides periodic status updates based on the state machine.
void status_output_task(void* pvParameters) {
while (1) { // Loop forever.
const char* status_text;
int proximity = current_proximity_cm;
RideStatus current_status = ride_status;

switch (current_status) {
case HALTED_BY_PROXIMITY:
status_text = "Obstruction - Ride Halted";
break;
case HALTED_BY_ESTOP:
status_text = "Emergency Stop Activated";
break;
case AWAITING_RESTART:
status_text = "Obstruction Cleared - Awaiting Restart";
break;
case RIDE_ALL_CLEAR:
default:
status_text = "All Clear";
break;
}
}

```

```

// Print the formatted status line to the serial monitor.
printf("[%lu] Proximity = %4dcm Status: %s\n", xTaskGetTickCount(), proximity, status_text);

vTaskDelay(pdMS_TO_TICKS(250)); // Pause for 250ms to print status messages 4 times per second.
}

}

// Main entry point of the application
void app_main(void) {
    gpio_config_t io_conf = {};
    io_conf.pin_bit_mask = (1ULL << LED_SYSTEM_POWER) | (1ULL << LED_EMERGENCY_BRAKE) | (1ULL << LED_ALL_CLEAR);
    io_conf.mode = GPIO_MODE_OUTPUT;
    gpio_config(&io_conf);

    gpio_config_t btn_conf = {};
    btn_conf.pin_bit_mask = (1ULL << BUTTON_EMERGENCY_STOP);
    btn_conf.mode = GPIO_MODE_INPUT;
    btn_conf.pull_up_en = GPIO_PULLUP_ENABLE;
    btn_conf.intr_type = GPIO_INTR_NEGEDGE;
    gpio_config(&btn_conf);

    gpio_set_direction(PROXIMITY_SENSOR_TRIG_PIN, GPIO_MODE_OUTPUT);
    gpio_set_direction(PROXIMITY_SENSOR_ECHO_PIN, GPIO_MODE_INPUT);
    gpio_set_level(LED_EMERGENCY_BRAKE, 0);
    gpio_set_level(LED_ALL_CLEAR, 1);

    sem_emergency_stop_button = xSemaphoreCreateBinary();
    sem_train_proximity_event = xSemaphoreCreateBinary();

    xTaskCreate(system_power_monitor_task, "SystemPower", 2048, NULL, 1, NULL);
    xTaskCreate(train_sensor_monitor_task, "TrainSensor", 2048, NULL, 2, NULL);
    xTaskCreate(ride_control_handler_task, "RideControl", 2048, NULL, 3, NULL);
    xTaskCreate(status_output_task, "StatusOutput", 2048, NULL, 1, NULL);

    gpio_install_isr_service(0);
    gpio_isr_handler_add(BUTTON_EMERGENCY_STOP, gpio_isr_handler, (void*) BUTTON_EMERGENCY_STOP);
}

// The ISR handler function for the E-Stop button.
static void IRAM_ATTR gpio_isr_handler(void* arg) {
    int64_t current_time_us = esp_timer_get_time();
    if ((current_time_us - last_isr_time_us) > (BUTTON_DEBOUNCE_TIME_MS * 1000)) {
        last_isr_time_us = current_time_us;
        BaseType_t xHigherPriorityTaskWoken = pdFALSE;
        xSemaphoreGiveFromISR(sem_emergency_stop_button, &xHigherPriorityTaskWoken);
        if (xHigherPriorityTaskWoken) {
            portYIELD_FROM_ISR();
        }
    }
}

```

```

// [Hard Real-Time] Monitors the track proximity sensor.
void train_sensor_monitor_task(void *pvParameters) {
    bool train_in_zone_prev = false;
    while (1) {
        gpio_set_level(PROXIMITY_SENSOR_TRIG_PIN, 0);
        ets_delay_us(2);
        gpio_set_level(PROXIMITY_SENSOR_TRIG_PIN, 1);
        ets_delay_us(10);
        gpio_set_level(PROXIMITY_SENSOR_TRIG_PIN, 0);
        while (gpio_get_level(PROXIMITY_SENSOR_ECHO_PIN) == 0);
        int64_t start_time = esp_timer_get_time();
        while (gpio_get_level(PROXIMITY_SENSOR_ECHO_PIN) == 1);
        int64_t end_time = esp_timer_get_time();

        long duration_us = end_time - start_time;
        int proximity_cm = (int)(duration_us * 0.0343 / 2.0);
        current_proximity_cm = proximity_cm;
        bool train_in_zone_now = (proximity_cm > 0 && proximity_cm < PROXIMITY_THRESHOLD_CM);
        is_train_in_zone = train_in_zone_now;
        if (train_in_zone_now && !train_in_zone_prev) {
            xSemaphoreGive(sem_train_proximity_event);
        }
        train_in_zone_prev = train_in_zone_now;
        vTaskDelay(pdMS_TO_TICKS(150));
    }
}

// [Soft Real-Time] Blinks the control panel's power LED.
void system_power_monitor_task(void *pvParameters) {
    while (1) {
        gpio_set_level(LED_SYSTEM_POWER, 1);
        vTaskDelay(pdMS_TO_TICKS(1000));
        gpio_set_level(LED_SYSTEM_POWER, 0);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

```

Diagram.json:

```
{
  "version": 1,
  "author": "Gemini Rework",
  "editor": "wokwi",
  "parts": [
    {
      "type": "board-esp32-devkit-c-v4",
      "id": "esp",
      "top": -19.2,
      "left": -4.76,
      "width": 10.0,
      "height": 1.0
    }
  ]
}
```

```
"attrs": { "builder": "esp-idf" }
},
{
  "type": "wokwi-led",
  "id": "ledGreen",
  "top": 111.6,
  "left": 282.2,
  "attrs": { "color": "red" }
},
{
  "type": "wokwi-led",
  "id": "ledRed",
  "top": 44.4,
  "left": 282.2,
  "attrs": { "color": "yellow" }
},
{
  "type": "wokwi-pushbutton",
  "id": "button",
  "top": -115.8,
  "left": 233.4,
  "rotate": 90,
  "attrs": { "color": "red" }
},
{
  "type": "wokwi-resistor",
  "id": "r1",
  "top": 157.85,
  "left": 190.6,
  "rotate": 180,
  "attrs": { "value": "220" }
},
{
  "type": "wokwi-resistor",
  "id": "r2",
  "top": 90.65,
  "left": 190.6,
  "rotate": 180,
  "attrs": { "value": "220" }
},
{
  "type": "wokwi-hc-sr04",
  "id": "ultrasonic1",
  "top": -171.3,
  "left": -42.5,
  "attrs": { "distance": "198" }
},
{
  "type": "wokwi-led",
  "id": "ledYellow",
  "top": -22.8,
  "left": 282.2,
  "attrs": { "color": "green" }
},
```

```
{
  "type": "wokwi-resistor",
  "id": "r3",
  "top": 23.45,
  "left": 190.6,
  "rotate": 180,
  "attrs": { "value": "220" },
},
{
  "type": "wokwi-text",
  "id": "label_estop",
  "top": -153.6,
  "left": 182.4,
  "attrs": { "text": "Emergency Brake / Restart" },
},
{
  "type": "wokwi-text",
  "id": "label_sensor",
  "top": -201.6,
  "left": -19.2,
  "attrs": { "text": "Proximity Sensor" },
},
{
  "type": "wokwi-text",
  "id": "label_clear",
  "top": -9.6,
  "left": 326.4,
  "attrs": { "text": "Ride Operational" },
},
{
  "type": "wokwi-text",
  "id": "label_power",
  "top": 57.6,
  "left": 326.4,
  "attrs": { "text": "System Power" },
},
{
  "type": "wokwi-text",
  "id": "label_brake",
  "top": 124.8,
  "left": 326.4,
  "attrs": { "text": "Ride Halted" }
},
],
"connections": [
  [ "esp:TX", "$serialMonitor:RX", "", [] ],
  [ "esp:RX", "$serialMonitor:TX", "", [] ],
  [ "ultrasonic1:VCC", "esp:VIN", "red", [ "v0" ] ],
  [ "r3:2", "esp:GND.2", "black", [ "h-46.8", "v-19.2" ] ],
  [ "r2:2", "esp:GND.2", "black", [ "h-46.8", "v-86.4" ] ],
  [ "r1:2", "esp:GND.2", "black", [ "h-46.8", "v-153.6" ] ],
  [ "ledYellow:C", "r3:1", "green", [ "v0" ] ],
  [ "ledRed:C", "r2:1", "gold", [ "v0" ] ],
  [ "r1:1", "ledGreen:C", "red", [ "v0", "h76.8" ] ]
]
```

```
[ "ledRed:A", "esp:5", "gold", [ "v19.2", "h-172.8", "v-9.6" ] ],
[ "ledGreen:A", "esp:4", "red", [ "v19.2", "h-172.8", "v-48" ] ],
[ "ledYellow:A", "esp:19", "green", [ "v19.2", "h-172.8", "v38.4" ] ],
[ "button:2.r", "esp:GND.2", "black", [ "v9.8", "h-115.4", "v57.6" ] ],
[ "button:1.r", "esp:18", "red", [ "v19.4", "h-124.8", "v124.8" ] ],
[ "ultrasonic1:VCC", "esp:5V", "white", [ "v48", "h-48", "v211.2" ] ],
[ "ultrasonic1:TRIG", "esp:17", "violet", [ "v48", "h76.4", "v134.4", "h9.6" ] ],
[ "ultrasonic1:ECHO", "esp:16", "blue", [ "v38.4", "h76", "v153.6" ] ],
[ "ultrasonic1:GND", "esp:GND.2", "black", [ "v28.8", "h85.2", "v57.6" ] ]
],
{
  "dependencies": {}
}
```

Readme.md:

[## Engineering Analysis \(Google Gemini used for formatting and sources\)](#)

[## Scheduler Fit](#)

My task priorities guarantee every Hard Real-Time (H) task's deadline by assigning the highest priority (3) to the ride_control_handler_task. Because FreeRTOS uses a preemptive scheduler, this critical task can interrupt any lower-priority task the moment it's signaled by the E-Stop ISR or a sensor event. This ensures that safety logic for the Universal Creative ride system is always processed immediately, well within its deadline. The timestamped output proves this, as an event occurring just after tick [\[5000\]](#) results in an updated STATUS message by the next output cycle at [\[5250\]](#), showing the high-priority handler ran and changed the state in between.

[\[5000\]](#) Proximity = 95cm STATUS: All Clear

[\[5250\]](#) Proximity = 15cm STATUS: Obstruction - Ride Halted

[## Race-Proofing](#)

A race condition could occur with the shared global state variables (ride_status, is_train_in_zone, current_proximity_cm), which are written by one task or ISR and read by others. These are protected by making them volatile, which is a C language primitive that prevents the compiler from making incorrect optimizations and ensures every read gets the absolute latest value from memory. For the ESP32's 32-bit architecture, reads and writes to these simple integer types are also atomic, meaning they cannot be interrupted halfway through. This combination prevents data tearing and ensures state consistency without the overhead of a mutex.

C // This volatile keyword is the primitive that protects against the race condition.

volatile RideStatus ride_status = RIDE_ALL_CLEAR;

[## Worst-Case Spike](#)

The heaviest load thrown at the prototype is a simultaneous E-Stop interrupt and a proximity sensor fault. The RTOS scheduler handles this by immediately running the E-Stop ISR, which signals the highest-priority ride_control_handler_task. This hard real-time task has an effective deadline of 10ms, set by its polling delay. The actual processing time for the state change logic is trivial, estimated at under 20 microseconds. This leaves a massive timing margin of over 9.98ms before the deadline would slip, proving the system is extremely robust and capable of handling worst-case event spikes safely.

Design Trade-off

One feature I chose not to add to keep the system's timing predictable was a dynamic audio system that would play sound effects from an SD card based on the ride's location. For a company like Universal Creative, rider safety depends on deterministic, guaranteed response times. File I/O from an SD card is a notoriously non-deterministic operation with highly variable delays, which would make it impossible to certify the controller as safety-critical. The right call was to simplify the system to its core safety functions, ensuring the hard real-time deadlines are always met and offloading non-essential features like audio to a separate, non-safety-critical processor.

Company Synopsis (Google Gemini used to generate company synopsis)

Universal Creative is the master planning, design, and engineering division for Universal Destinations & Experiences. They are responsible for creating the world's most immersive theme park attractions, from The Wizarding World of Harry Potter™ to the Jurassic World VelociCoaster. A real-time operating system (RTOS) is non-negotiable for their rides, as it provides the determinism needed to guarantee rider safety. An RTOS ensures that critical events, like an E-Stop, are processed within a fixed, predictable deadline, something a standard OS cannot promise.

Task & Deadline Analysis

Task ISR Name Period Deadline Type Consequence of Failure

gpio_isr_handler Aperiodic / ~10µs Hard Catastrophic.

-Failure to execute would render the physical E-Stop button useless.

ride_control_handler_task 10ms Hard Catastrophic.

-Failure to process a halt signal could lead to vehicle collision or injury.

train_sensor_monitor_task 150ms Hard Catastrophic.

-Missing a vehicle detection could defeat the anti-collision system.

status_output_task 250ms Soft Minor.

-A status message on the operator's console would be delayed or skipped. No impact on rider safety.

system_power_monitor_task 2000ms Soft Trivial.

-The power indicator light on the control panel might stutter, with no effect on ride safety.

=====

Concurrency Diagram (Generated by Google Gemini)

=====

KEY:

[Hardware] --> Signal (Semaphore) -->

+-----+ ... Data (Shared Var) ...>

| Task/ISR |

+-----+

```

+-----+
| system_power_monitor |
[ E-Stop Button ] | (Soft, 2000ms) |----> [ Yellow LED ]
+-----+
|
| Hardware Interrupt
V
+-----+
| gpio_isr_handler |
| (ISR / Hard) |--- sem_emergency_stop_button -->+-----+
| (Aperiodic / ~10us) || ride_control_handler |
+-----+ | (Hard, 10ms) |
| |----> [ Red/Green LEDs ]
+-----+-----+
|
| ... ride_status ... >+-----+
| | status_output_task |
| | (Soft, 250ms) |
| +-----+
|| 
|| 
[ Proximity Sensor ] |<... is_train_in_zone ... |
|| 
|| V
| Sensor Reading | [ Serial Monitor ]
V |
+-----+ |
| train_sensor_monitor_task|--- sem_train_proximity_event -->+ |
| (Hard, 150ms) ||
| | ... current_proximity_cm .....>+
+-----+

```