

FreeRTOS

FreeRTOS

- I. Learning Objectives
- II. FreeRTOS Introduction**
- III. Hardware Setup
- IV. Experiment Steps
- V. Code Analysis
- VI. Experiment Phenomenon

I. Learning Objectives

1. Understand basic FreeRTOS knowledge.
2. Use FreeRTOS to schedule RGB, serial port, LED, buzzer, and OLED peripherals.

II. FreeRTOS Introduction

FreeRTOS is a real-time operating system (RTOS) widely used in embedded systems, characterized by open source, lightweight, and scalable features, particularly suitable for resource-constrained microcontrollers (MCUs) and embedded devices. Its design core is to achieve efficient multi-task management and real-time response on resource-constrained hardware. It provides core functions such as task scheduling, memory management, synchronization and communication, helping developers build responsive and highly reliable real-time applications, widely used in smart home, industrial control, IoT devices and other fields.

Project address: <https://github.com/FreeRTOS/FreeRTOS>

Official website: <https://www.freertos.org>

I. FreeRTOS Principles

1. Task Management and Scheduling: Core Mechanism of Multi-tasking

The core of FreeRTOS is the **task scheduler**, which manages multiple independent tasks and decides when to run which task. A task is the smallest execution unit in the operating system, with independent stack space and execution logic, can be viewed as a "micro-program".

- **Essence of Tasks:** Each task is an infinite loop (while(1)), containing independent local variables and stack (for saving context, such as register status). Task creation requires specifying **priority** (0~configMAX_PRIORITIES-1, higher value means higher priority).
- **Task Control Block (TCB):** Each task corresponds to a TCB_t structure (task control block), used to store key information of the task, including: stack pointer (save task context), priority, status, delay time, belonging queue, etc. All task TCBs form a linked list for scheduler management.
- **Task States and Transitions:** Tasks have 4 core states, scheduler achieves task switching through state transitions:
 - **Ready:** Task is ready to run, waiting for CPU resources (only limited by priority).
 - **Running:** Task is currently occupying CPU execution.

- **Blocked:** Task temporarily cannot run due to waiting for events (such as delay, semaphore, queue data), does not participate in scheduling.
- **Suspended:** Task is forcibly paused, needs explicit recovery through `vTaskResume()`, does not participate in scheduling.
- **Scheduler Working Mechanism:**

FreeRTOS uses **preemptive scheduling** as core (default mode), meaning when high-priority tasks are ready, they immediately preempt CPU resources of low-priority tasks, ensuring real-time response of high-priority tasks. Specific rules:

- Only ready-state tasks participate in scheduling, scheduler always selects **highest-priority ready task** to run.
- Same-priority tasks can enable **time-slice round-robin** (configured through `configUSE_TIME_SLICING`), each task takes turns occupying CPU (time slice determined by system tick period).
- Cooperative scheduling (optional): Actively yield CPU through `taskYIELD()`, suitable for scenarios with low real-time requirements.

2. Inter-task Communication and Synchronization: Solving Multi-task Collaboration Problems

When multiple tasks run in parallel, specific mechanisms are needed to achieve data interaction and resource competition control. FreeRTOS provides rich synchronization and communication tools:

- **Queue:**

Used for asynchronous data transfer between tasks (such as sensor data, commands), based on FIFO (First In First Out) principle, supports multi-task read/write. Queues can store fixed-size elements (such as integers, pointers), block when queue is full during write, block when queue is empty during read (blocking time configurable).

- **Semaphore:**

Used for resource synchronization or counting, divided into two types:

- Binary semaphore: Only two states (0/1), similar to "switch", often used for mutual exclusion (such as controlling single thread access to hardware resources) or synchronization (such as task A notifying task B after completion).
- Counting semaphore: Supports counting (0~n), used to manage limited number of resources (such as 3 UART ports, count initial is 3, decrease by 1 when occupied, increase by 1 when released).

- **Mutex:**

Special binary semaphore, used to solve **priority inversion** problem (when low-priority task occupies resources, high-priority task is blocked due to waiting for resources, causing medium-priority tasks to "jump the queue"). Mutex temporarily boosts low-priority task's priority through **priority inheritance protocol**, making it release resources faster.

- **Event Group:**

Used for multiple tasks waiting for multiple events (such as "task waits for both sensor A and sensor B to be ready"). Each event is represented by a bit, tasks can wait for "any event satisfied" or "all events satisfied", suitable for complex condition trigger scenarios.

- **Task Notification:**

More efficient mechanism than queues (no additional memory allocation), achieves communication by directly writing notification value to task's TCB. A task can be notified by multiple tasks, supports operations like overwrite, increment, set bit, suitable for lightweight interaction.

3. Memory Management: Adapting to Embedded Resource-constrained Scenarios

Embedded systems have limited memory, FreeRTOS provides 5 memory allocation schemes (selected through `heap_x.c`), balancing efficiency, determinism and flexibility:

- **heap_1**: Only supports memory allocation (`pvPortMalloc()`), does not support free (`vPortFree()`), suitable for scenarios without dynamic memory release (such as tasks not deleted after creation), advantages are simple, no fragmentation.
- **heap_2**: Supports allocation and free, but does not merge adjacent free blocks, prone to memory fragmentation after long-term use, suitable for scenarios with fixed memory block sizes.
- **heap_3**: Wraps standard library `malloc()` and `free()`, depends on compiler implementation, advantage is universal, disadvantage is poor real-time performance (allocation time uncertain).
- **heap_4**: Supports allocation, free and free block merging (reduces fragmentation), manages memory through linked list, suitable for scenarios needing dynamic release and sensitive to fragmentation.
- **heap_5**: Based on `heap_4`, supports non-contiguous memory regions (such as external RAM + internal RAM), defines memory blocks through `vPortDefineHeapRegions()`, suitable for hardware with scattered memory distribution.

4. Time Management: Tick-based Timing Mechanism

FreeRTOS relies on **system tick** to achieve timing and scheduling, ticks are generated by hardware timer (such as `SysTick`), period (`configTICK_RATE_HZ`) usually 100~1000Hz (i.e., 1~10ms once).

- Scheduler's task switching, delay functions (such as `vTaskDelay()`) are all based on tick counting.
- `vTaskDelay(x)`: Task blocks for `x` ticks then enters ready state (relative delay, calculated from call moment).
- `vTaskDelayUntil()`: Absolute delay, ensures task executes at fixed period (such as running every 10ms, not affected by task execution time).

II. FreeRTOS Features

1. Open Source and Free, Friendly License

Uses MIT license, allows free commercial use, source code completely open, no patent or commercial authorization restrictions, suitable for enterprise and personal projects.

2. Lightweight, Extremely Low Resource Usage

Core code only a few thousand lines, compiled size can be as low as 6KB (depending on feature trimming), RAM usage mainly depends on task stack size (each task stack can be as small as dozens of bytes), suitable for 8-bit/16-bit MCUs (such as STM8, PIC) and resource-constrained 32-bit MCUs (such as ARM Cortex-M0).

3. Highly Scalable, Configure on Demand

Through `FreeRTOSConfig.h` header file, functions can be enabled/disabled (such as queues, mutexes, time-slice round-robin), adjust priority count, stack size and other parameters, only keep necessary components, maximize resource savings.

4. Strong Real-time Performance, Deterministic Scheduling

Preemptive scheduling ensures high-priority tasks can respond at microsecond level (delay determined by task switching time, usually $<10\mu s$), scheduling behavior is predictable (task switching time fixed under same conditions), meeting real-time scenario requirements such as industrial control, automotive electronics.

5. Broad Hardware and Architecture Support

Supports almost all mainstream embedded architectures, including ARM (Cortex-M/R/A), AVR, PIC, RISC-V, MIPS, etc., adapts to thousands of MCUs (such as STM32, ESP32, MSP430), official provides numerous porting examples.

6. Flexible Task Scheduling Strategies

Supports preemptive, cooperative scheduling, same-priority tasks can time-slice round-robin, priority range configurable (up to 256 levels), meeting task priority design needs of different scenarios.

7. Complete Synchronization and Communication Mechanisms

Provides queues, semaphores, mutexes, event groups, task notifications and other tools, covering single/multi-task data transfer, resource competition, condition triggering and other needs, with simple and easy-to-use interfaces.

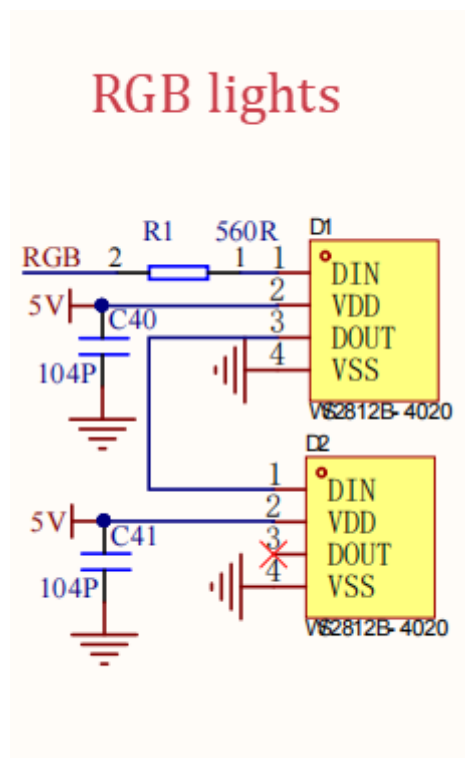
8. Priority Inversion Protection

Mutex's priority inheritance protocol solves common priority inversion problems in real-time systems, avoiding high-priority tasks being blocked for long time due to waiting for resources held by low-priority tasks.

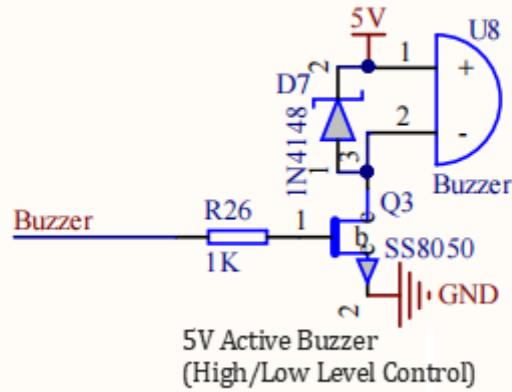
9. Strong Portability

Kernel is hardware-independent, adapts to different architectures through "port layer" (only need to implement hardware-related code such as context switching, interrupt handling), porting workload is small (usually a few hundred lines of code).

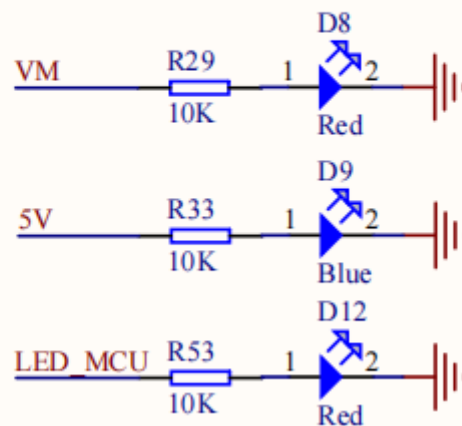
III. Hardware Setup



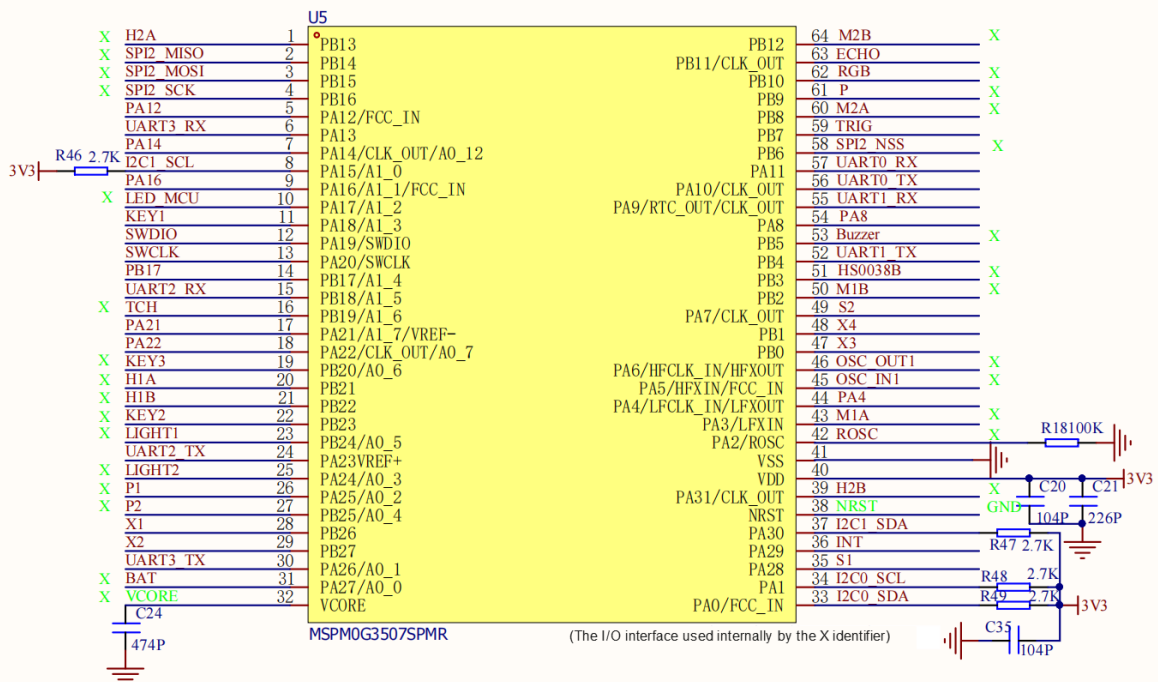
Buzzer



LED Light

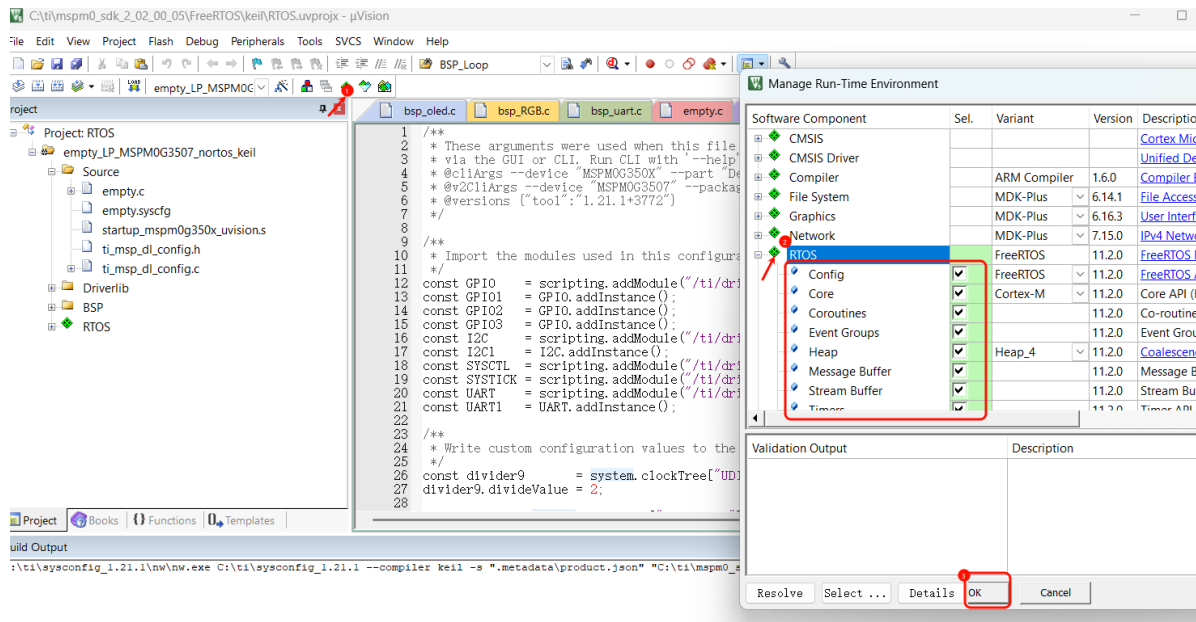


MCU

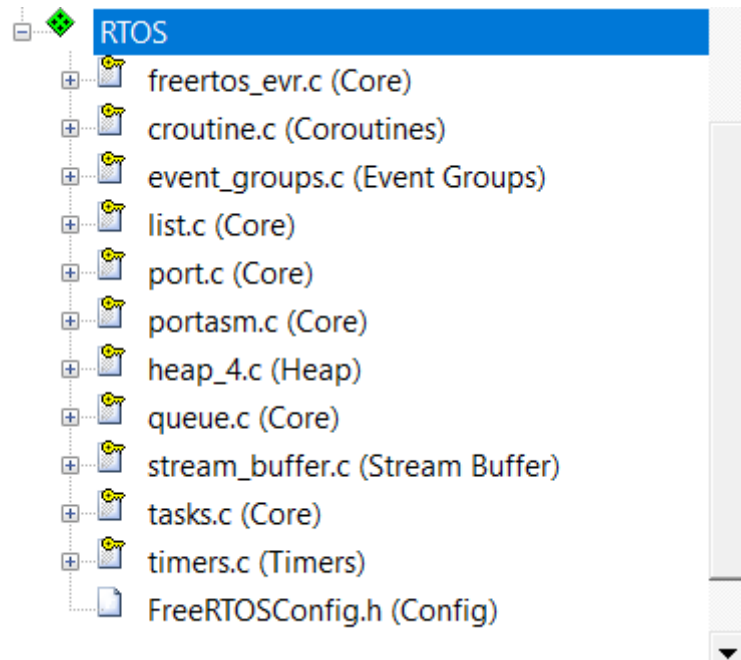


IV. Experiment Steps

This project is built based on the previously configured 80MHz crystal example. We click Manage RunTime Environment, check and add all components under RTOS. Here we select FreeRTOS as the RTOS version, and select Heap_4 as the memory management algorithm.



Add components as follows



Then we add peripheral configuration through sysconfig tool

RGB light configuration

PIO (3 Added) ?

+ ADD

REMOVE ALL

✓ LED



✓ BEEP



✓ RGB



Name	RGB
Port	Any
Port Segment	Any

Group Pins



1 added

+ ADD

REMOVE ALL

✓ WQ2812



Name	WQ2812
Direction	Output
Initial Value	Cleared
IO Structure	Any

Digital IOMUX Features



Assigned Port	PORTB
Assigned Port Segment	Any
Assigned Pin	10

Interrupts/Events



LaunchPad-Specific Pin	No Shortcut Used
------------------------	------------------

Digital IOMUX Peripheral and Pin Configuration



Buzzer configuration

GPIO (3 Added) ?

+ ADD

REMOVE ALL

✓ LED

✓ BEEP

✓ RGB

Name

BEEP

Port

Any

Port Segment

Any

Group Pins

1 added

+ ADD

REMOVE ALL

✓ OUT

Name

OUT

Direction

Output

Initial Value

Cleared

IO Structure

Any

Digital IOMUX Features

Assigned Port

PORTB

Assigned Port Segment

Any

Assigned Pin

5

Interrupts/Events

LaunchPad-Specific Pin

No Shortcut Used

MCU indicator light configuration

GPIO (3 Added) ?

+ ADD

REMOVE ALL

✓ LED



✓ BEEP



✓ RGB



Name	LED
Port	PORTA
Port Segment	Any

Group Pins



1 added

+ ADD

REMOVE ALL

✓ MCU



Name	MCU
Direction	Output
Initial Value	Cleared
IO Structure	Any

Digital IOMUX Features



Assigned Port	PORTA
Assigned Port Segment	Any
Assigned Pin	17

Interrupts/Events



LaunchPad-Specific Pin	No Shortcut Used
------------------------	------------------

Serial port configuration

✓

UART_0

📄

🗑

Name ?	UART_0
Selected Peripheral	UART0

Quick Profiles

^

UART Profiles

Custom

▼

Basic Configuration

^

UART Initialization Configuration

^

Clock Source

MFCLK

▼

Clock Divider

Divide by 1

▼

Calculated Clock Source

4.00 MHz

Target Baud Rate

9600

Calculated Baud Rate

9598.08

▼

Calculated Error (%)

0.02

Word Length

8 bits

▼

Parity

None

▼

Stop Bits

One

▼

HW Flow Control

Disable HW flow control

▼

Serial port configuration

Interrupt Configuration

^

Enable Interrupts

Receive

▼

Interrupt Priority

Default

▼

OLED configuration

Name	OLED_IIC
Selected Peripheral	I2C1

Quick Profiles
^

I2C Profiles
Custom
▼

Basic Configuration
^

Enable Controller Mode
☒

Enable Target Mode
☐

Clock Configuration
▼

I2C Controller Basic Configuration
^

Standard Bus Speed
Standard (100kHz)
▼

Custom Bus Speed (Hz)
100000

Actual Bus Speed (Hz)
100000.00
▼

Enable 10-bit Addressing Mode
☐

Advanced Configuration
^

Enable I2C After Initialization
☒

Analog Glitch Filter
Filter pulses < 50ns
▼

Digital Glitch Filter
Disabled
▼

Calculated Digital Glitch Filter
0.00 s

Advanced Configuration ^

Enable I2C After Initialization	<input checked="" type="checkbox"/>
Analog Glitch Filter	Filter pulses < 50ns ▼
Digital Glitch Filter	Disabled ▼
Calculated Digital Glitch Filter	0.00 s

I2C Controller Advanced Configuration ^

RX FIFO Trigger Level	RX FIFO contains >= 1 bytes ▼
TX FIFO Trigger Level	TX FIFO contains <= 0 bytes ▼
Enable Loopback Mode	<input type="checkbox"/>
Enable Multi-Controller Mode	<input type="checkbox"/>
Enable Controller Clock Stretching...	<input checked="" type="checkbox"/>

Additional Timeout Configuration ^

We focus on introducing some FreeRTOS configurations

CPU frequency setting

```
#define configCPU_CLOCK_HZ    ( ( unsigned long ) 80000000 )
```

Systick interrupt frequency setting

```
#define configTICK_RATE_HZ    1000
```

//Software timer is turned off and not used. If enabled, need to implement
vApplicationGetTimerTaskMemory vApplicationGetIdleTaskMemory

```
#define configUSE_TIMERS      0
```

System call stack size

```
#define configSYSTEM_CALL_STACK_SIZE    128
```

Enable static memory allocation

```
#define configSUPPORT_STATIC_ALLOCATION    1
```

Enable dynamic memory allocation

```
#define configSUPPORT_DYNAMIC_ALLOCATION    1
```

Kernel automatically allocates memory, we allocate manually here

```
#define configKERNEL_PROVIDED_STATIC_MEMORY    0
```

Boot MPU M0 core does not support, turn off here

```
#define configENABLE_MPU    0
```

FreeRTOS heap size in bytes

V. Code Analysis

xTaskCreate

Function Prototype	BaseType_t xTaskCreate(TaskFunction_t pxTaskCode, const char * const pcName, const configSTACK_DEPTH_TYPE uxStackDepth, void * const pvParameters, UBaseType_t uxPriority, TaskHandle_t * const pxCreatedTask)
Function Description	Create a new task in FreeRTOS. After successful creation, the task enters ready state, waiting for scheduler to allocate CPU resources to run
Input Parameters	pxTaskCode: Pointer to task function, task function must be in void vTaskFunction(void *pvParameters) format pcName: Task name, used for debugging identification, length not exceeding configMAX_TASK_NAME_LEN uxStackDepth: Task stack depth, unit is "stack items" (not bytes), stack size = uxStackDepth × stack item bytes pvParameters: Parameter pointer passed to task function uxPriority: Task priority, higher value means higher priority, range is 0~configMAX_PRIORITIES-1 pxCreatedTask: Used to save handle of newly created task, can be set to NULL (means not saved)
Return Value	BaseType_t type, return pdPASS means task creation successful; return errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY means insufficient memory, task creation failed

vTaskDelayUntil

Function Prototype	void vTaskDelayUntil(TickType_t *pxPreviousWakeTime, TickType_t xTimeIncrement)
Function Description	Implement periodic task scheduling, ensure task executes at fixed time intervals, suitable for scenarios requiring strict periods
Input Parameters	pxPreviousWakeTime: Pointer to TickType_t type variable, used to store the timestamp of last task wake-up, needs to be initialized before first call (such as using xTaskGetTickCount() to get current time) xTimeIncrement: Time interval of task execution, unit is system tick, i.e., time interval between two task execution starts
Return Value	None

xQueueSend

Function Prototype	BaseType_t xQueueSend(QueueHandle_t xQueue, const void *pvItemToQueue, TickType_t xTicksToWait)
Function Description	Send data to the tail of queue (macro definition, essentially calls xQueueGenericSend and specifies queueSEND_TO_BACK), wait specified time if queue is full
Input Parameters	xQueue: Handle of target queue pvItemToQueue: Pointer to data to send, data will be copied to queue xTicksToWait: Maximum time to wait for queue to have space (unit: system tick), 0 means no wait, portMAX_DELAY means infinite wait
Return Value	BaseType_t type, pdPASS means send successful; errQUEUE_FULL means wait timeout and queue still full

xQueueReceive

Function Prototype	BaseType_t xQueueReceive(QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait)
Function Description	Receive data from queue head and remove it from queue, wait specified time if queue is empty
Input Parameters	xQueue: Handle of target queue pvBuffer: Pointer to receive data buffer, data in queue will be copied to this buffer xTicksToWait: Maximum time to wait for queue to have data (unit: system tick), 0 means no wait, portMAX_DELAY means infinite wait
Return Value	BaseType_t type, pdPASS means receive successful; errQUEUE_EMPTY means wait timeout and queue still empty

prvQueueSendTask

Function Prototype	static void prvQueueSendTask(void *pvParameters)
Function Description	Queue send task: Periodically send fixed value (100UL) to queue, send frequency defined by mainQUEUE_SEND_FREQUENCY_MS, uses vTaskDelayUntil to ensure strict period
Input Parameters	pvParameters: Task parameter, must equal mainQUEUE_SEND_PARAMETER (validated through configASSERT)
Return Value	None

prvQueueReceiveTask

Function Prototype	static void prvQueueReceiveTask(void *pvParameters)
Function Description	Queue receive task: Wait indefinitely for queue data, after receiving verify if value is 100UL, if matches then control LED to blink (restore state after short delay)
Input Parameters	pvParameters: Task parameter, must equal mainQUEUE_RECEIVE_PARAMETER (validated through configASSERT)
Return Value	None

prvBuzzerTask

Function Prototype	static void prvBuzzerTask(void *pvParameters)
Function Description	Buzzer control task: With 2-second period, control buzzer to turn on for 200ms then turn off, uses vTaskDelayUntil to ensure period stability
Input Parameters	pvParameters: Task parameter, must equal mainBUZZER_TASK_PARAMETER (validated through configASSERT)
Return Value	None

prvUartTask

Function Prototype	static void prvUartTask(void *pvParameters)
Function Description	Serial port send task: With 1-second period, cyclically send counter information through serial port (format: "FreeRTOS running, counter: %u\r\n"), counter increments, uses vTaskDelayUntil to ensure period
Input Parameters	pvParameters: Task parameter, must equal mainUART_TASK_PARAMETER (validated through configASSERT)
Return Value	None

prvRgbTask

Function Prototype	static void prvRgbTask(void *pvParameters)
Function Description	RGB light control task: With 2-second period, update all RGB light colors (cycle through preset color array rgbColors), and print current color name through serial port, uses vTaskDelayUntil to ensure period
Input Parameters	pvParameters: Task parameter, must equal mainRGB_TASK_PARAMETER (validated through configASSERT)
Return Value	None

prvOledTask

Function Prototype	static void prvOledTask(void *pvParameters)
Function Description	OLED display task: After initializing OLED, update screen display with 2-second period, first line fixed displays "Current Color:", second line displays current RGB light color name (taken from colorNames array), uses vTaskDelayUntil to ensure period
Input Parameters	pvParameters: Task parameter, must equal mainOLED_TASK_PARAMETER (validated through configASSERT)
Return Value	None

rgb_SetColor

Function Prototype	void rgb_SetColor(unsigned char LedId, unsigned long color)
Function Description	Set color of specified LED, decompose color value into three primary color components and store in LED data array
Input Parameters	LedId: LED number to control (must be less than ledsCount, otherwise no operation performed) color: 32-bit color value, format is red (high 8 bits), green (middle 8 bits), blue (low 8 bits)
Return Value	None

rgb_SetRGB

Function Prototype	void rgb_SetRGB(unsigned char LedId, unsigned long red, unsigned long green, unsigned long blue)
Function Description	Set color of specified LED through three primary color components, combine red, green, blue components into 32-bit color value, then call rgb_SetColor to implement setting
Input Parameters	LedId: LED number to control red: Red component (0~255) green: Green component (0~255) blue: Blue component (0~255)
Return Value	None

VI. Experiment Phenomenon

After the program is flashed, connect Type-C to the car, open serial port assistant and configure as follows. LED executes one toggle every 1s, serial port counting task prints once every 1s, RGB, buzzer, OLED execute once every 2s.

```
FreeRTOS running, counter: 5
FreeRTOS running, counter: 6
RGB color changed to Yellow
FreeRTOS running, counter: 7
FreeRTOS running, counter: 8
RGB color changed to Purple
FreeRTOS running, counter: 9
FreeRTOS running, counter: 10
RGB color changed to Cyan
```

波特率

9600

▼

停止位

1

▼

数据位

8

▼

校验位

None

▼

串口操作

☒

 打开串口

