B31DG-Embedded Software 2023-2024

Assignment 2 Report FreeRTOs task

Student Name:Jie Liu
Student ID:H00364968
Email:jl2034@hw.ac.uk
Code repository:[yaolin759/B31DG-H00364968-Assignment2 (github.com)]

1. Hardware circuit

This section describes the specific circuit implementation of the FreeRTOS system, which involves components such as the ESP32 board, two LEDs, a button, a 330Ω resistor, and a potentiometer. Below are the detailed circuit structures.

To implement task1, an oscilloscope is connected to Pin12 to display the waveform. The oscilloscope needs to be connected to the signal on one end and to ground on the other end. To implement task7, the button is connected to the 3.3V power supply and Pin25. When connecting, it is important to ensure that the button is securely connected (the button's internals form an "H" circuit). For tasks 4 and 7, two LEDs are connected to pin19 and pin23 respectively. When connecting the potentiometer, the middle sliding terminal of the potentiometer is connected to pin27, and the other two terminals are connected to power and GND respectively.To implement tasks 2, 3, and 5, the 3.3V square wave signals 1 and 2 are connected to pin15 and pin16 respectively.

Figure 1 depict the specific circuit diagrams of the FreeRTOS system. Figure 1 illustrates the specific connections of the oscilloscope and the voltage source.

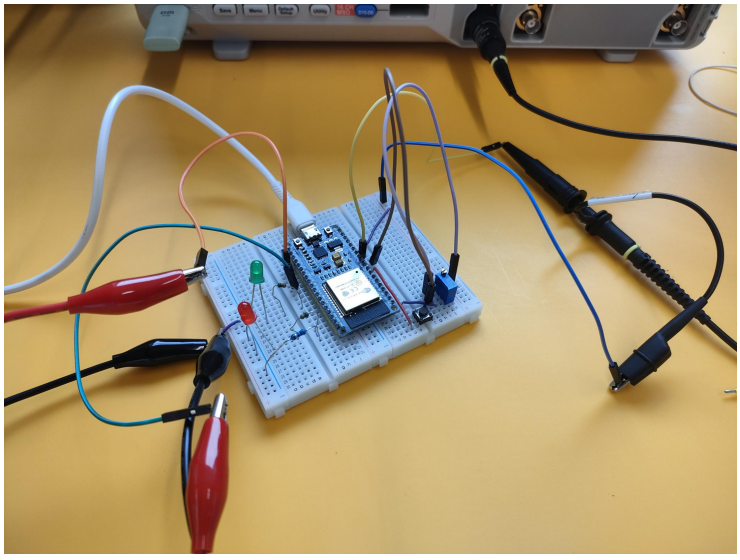

Figure 1 Hardware circuit 1

2. UML

This section is the UML diagram of the FreeRTOS system. The UML diagram includes: creating an event queue, creating a mutex semaphore, creating and starting tasks, and the parameters and functions involved in each task.

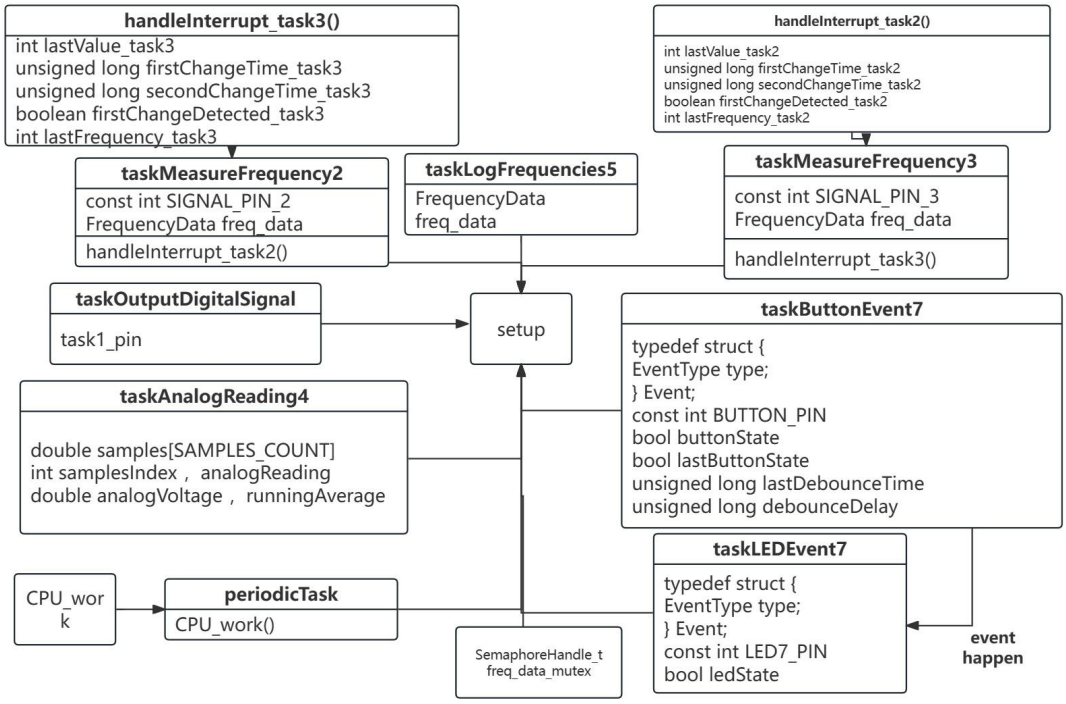

Figure 2 UML diagram of FreeRTO system

3.Results analysis and discuss

3.1 Results analysis

1.Task1

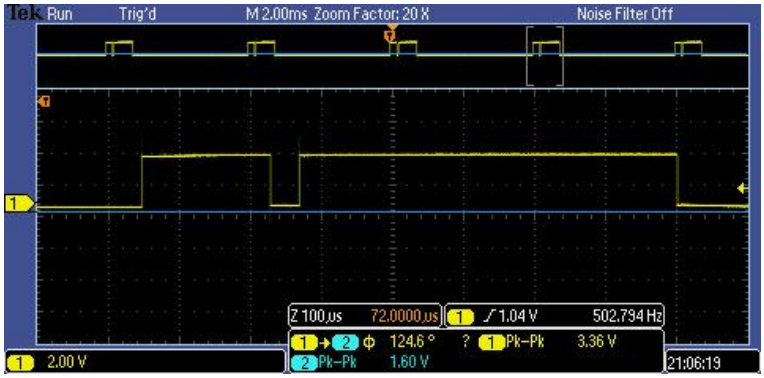

Figure 3 Output digital signal of Task1

The effect of Task 1 is to generate the square wave shown in Figure 3. Through measurements, it can be observed that the durations of the high and low levels within one period are as follows: (high 183, low 38, high 532, low 3248). The errors are minimal, meeting the requirements of Task.

2.Task2&3&5

The test results for Task 2 & 3 & 5 are shown in Table 1. From the test results, we can observe that when the frequency is outside the specified range (Signal 1's range is 333-1000, Signal 2's range is 500-1000), the measurements are accurate and stable. When the voltage is

Signal1	Signal2	Log1	Log2	state
0	0	0,0	0,0	stable
333	500	0,0	1,0	slight fluctuation
666	750	49,50	50,49	slight fluctuation
1000	1000	99,99	99,97	slight fluctuation
1200	1200	99,99	99,99	stable

within the specified range, the accuracy of the measurements is within 2.5%. It is important to note that tasks 2, 3, and 5 need to Obtain the semaphore to protect access to freq_data

```
// task 2&3&5 Obtain the semaphore to protect access to freq_data
if (xSemaphoreTake(freq_data_mutex, portMAX_DELAY) == pdTRUE) {..}
```

3.Task4

In this task, a potentiometer is used, and twisting the potentiometer can change the voltage sampled by the pin. The effect achieved is that when the analog input is greater than 1.65V, the LED lights up, and when it is less than 1.65V, the LED turns off.

4.Task7

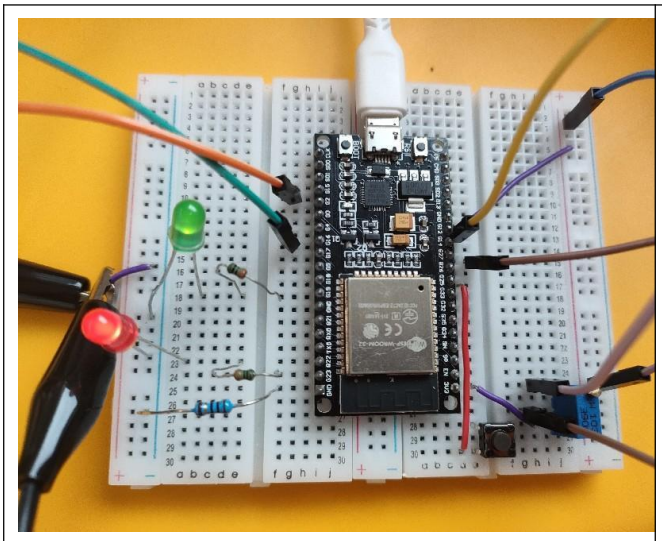

Figure 4 PushButton 1,LED light

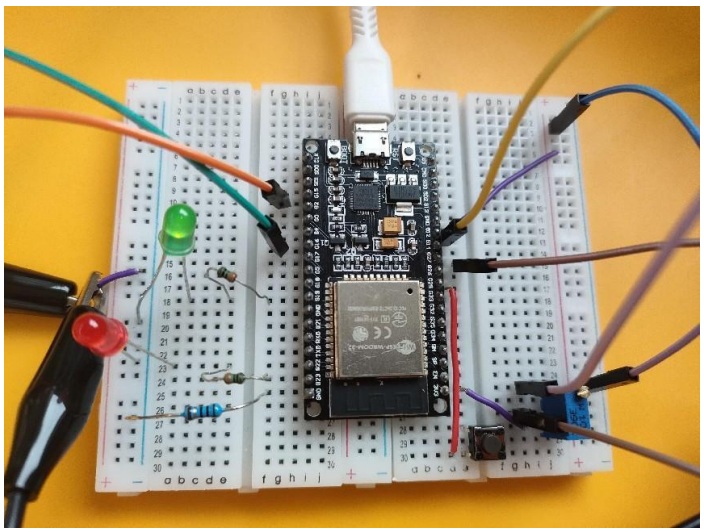

Figure 5 PushButton 2,LED low

Figure 1 depicts the state where the LED is lit after the button is pressed, while Figure 2 illustrates the state where the LED is off after the button is released. The button has been tested for debounce functionality and effective signal reception.

In terms of implementation, I control the LED through two independent tasks. taskLEDEvent7 and 6.taskButtonEvent7. I utilize an event queue to enable communication between these two tasks. The relevant code is as follows:

```
QueueHandle_t eventQueue;// Create an event queue
typedef enum {
  BUTTON_PRESSED} EventType;// Define event types
typedef struct {
  EventType type;} Event;// Define the structure for events
//Some code for taskButtonEvent7
if (buttonState == HIGH) {
          Event event = {BUTTON_PRESSED};
          xQueueSend(eventQueue, &event, portMAX_DELAY);  }
```

5.Task8

In Task8, the main consideration is to ensure that the entire task is completed using a for loop. CPU_work() has been tested and verified to run correctly.

3.2 Discuss

3.2.1.Priorities and Size of stack

When determining the priorities of FreeRTOS tasks and setting their stack sizes, considerations should include the criticality of tasks, timing requirements, resource usage, and dependencies between tasks. Proper adjustment of priorities and stack sizes can help avoid watchdog issues. Below is an explanation of how these settings are determined:

1.Task Priorities:

1. Tasks with stricter timing constraints or hard real-time deadlines should have higher priorities.

2. Task 1 has the highest priority (priority 4) because it has a strict real-time deadline and must be executed within a specified period.

3. Task 2 and Task 3 have slightly lower priorities (priority 3) as they measure the frequency of square wave signals with periods of 20ms and 8ms, respectively. These tasks also have strict real-time deadlines but with longer periods compared to Task 1.

4. Task 4, 5, 7, and 8 have lower priorities. Compared to Task 1, 2, and 3, these tasks have softer real-time requirements and can be preempted by higher priority tasks.

2.Stack Sizes:

1. Stack sizes should be chosen based on the memory requirements of each task, considering factors such as local variables, function call depth, and stack usage during context switches. Ensure that the stack size is sufficient to accommodate the worst-case stack usage of each task to prevent stack overflow issues.

2. Task 1, Task 2, and Task 3 are allocated larger stack sizes (1024 or 2048 bytes) as they may involve more complex computations or data processing.

3. Task 4, 5, 7, and 8 are allocated smaller stack sizes (1024 or 1000 bytes) as they may have simpler functionalities and may not require as much stack space.

3.2.2.Global structure

In general, to protect access to global structures, I utilized a **mutual exclusion semaphore** in tasks involving shared global variables to ensure mutual access when updating or accessing these global variables.

For Task 2, 3, and 5:These tasks involve monitoring signal states and measuring frequencies. I use **global structure (struct)** to store the frequencies measured.To protect access to global structures, I employed the **semaphore `freq_data_mutex`** to ensure mutual access when updating the `freq_data` global variable. Since these tasks need to share the `freq_data` variable, it is crucial to protect it to prevent data races and inconsistent states.

```
3.    SemaphoreHandle_t freq_data_mutex;//semaphore
4.    typedef struct {
5.      int freq_task2;
6.      int freq_task3;
7.    } FrequencyData;
8.    FrequencyData freq_data;// Global variable to store the frequencies
9.    // task 2&3&5 Obtain the semaphore to protect access to freq_data
10.   if (xSemaphoreTake(freq_data_mutex, portMAX_DELAY) == pdTRUE) {..}
```

For Task 1, 4, 7, and 8:These tasks only control the states of specific pins using pins and do not involve shared global resources.

3.2.3.Task7 Time Analysis

The worst-case delay (response time) between the time the push button is pressed and the

time the LED is toggled depends on the debounce mechanism and the scheduling of the taskButtonEvent7 task.The worst-case delay occurs when the button is pressed just after the debounce delay ends and just before the task is ready to check the button state again. In this scenario:

1. The debounce mechanism resets the debounce timer when the button state changes, meaning the task needs at least debounceDelay milliseconds to recognize a stable button state change.

2.After the debounce delay, the taskButtonEvent7 task's scheduling delay is 10 milliseconds (vTaskDelay(pdMS_TO_TICKS(10))).

3. Therefore, the worst-case delay is the sum of the debounce delay and the task delay:
Worst-case delay = Debounce delay + Task delay = 50 milliseconds + 10 milliseconds = 60 milliseconds.

So, the worst-case delay between the time the push button is pressed and the time the LED is toggled is **60 milliseconds.**

3.2.4 Compare between FreeRTOS implementation and custom cyclic executive

1. Flexibility and Scalability: FreeRTOS offers better implementation of multitasking projects and allows for easier addition of tasks compared to a custom cyclic executive implementation. In contrast to a custom solution, FreeRTOS provides a more flexible and scalable solution. It offers a wide range of features such as task priority assignment, dynamic task creation, inter-task communication mechanisms (queues, semaphores), and timers.

2. Resource Utilization: FreeRTOS may consume more memory and processing power compared to a custom cyclic executive implementation due to stack and priority settings. However, it can better utilize resources, leading to performance optimization in a multitasking environment.

3. Development Time and Effort: FreeRTOS provides ready-made components and APIs, significantly reducing development time and effort.

4. Real-Time Performance: Both FreeRTOS and custom cyclic executive implementations can achieve real-time performance. FreeRTOS can better achieve this goal through effective task scheduling and management.

4.Summary

This experimental project aims to create a machine monitoring system using FreeRTOS and an ESP32 kit. The system comprises eight tasks, each with strict real-time deadlines, to monitor and control various aspects of the machine. These tasks involve outputting digital signals, measuring signal frequencies, sampling analog inputs, logging frequency information, and controlling LEDs via buttons. The system must ensure stable operation while promptly detecting and handling any anomalies.

The entire experiment implementation consists of two stages: first, implementing individual tasks, and then adjusting settings to achieve multitasking parallelism. During the implementation of individual tasks, I encountered issues such as hardware contacts not being properly made, resulting in interference with signal output. In the multitasking parallelism phase, I faced watchdog exceptions, which I resolved by adjusting task priorities, understanding task implementation principles, and adjusting stack capacities. Leveraging the flexibility and robust features of FreeRTOS, multiple tasks can be efficiently managed to ensure timely completion and coordination.