IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.102 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 2, February 2024

DOI: 10.17148/IJARCCE.2024.13246

Synchronization Techniques in Real-Time Operating Systems: Implementation and Evaluation on Arduino with FreeRTOS

Sakthivel V[1], Sreeja P[2]

Student, Department of Electronics and Communication, Sri Ramakrishna Engineering College, Coimbatore, India[1]

Student, Department of Electronics and Communication, Sri Ramakrishna Engineering College, Coimbatore, India[2]

Abstract: Real-time operating systems (RTOS) play a pivotal role in facilitating efficient multitasking and resource management in embedded systems. This paper delves into the intricacies of Semaphore and Mutex synchronization techniques within the FreeRTOS framework on Arduino microcontrollers, focusing on their application in synchronizing tasks for LED control. Semaphore serves as a signalling mechanism, allowing tasks to coordinate and synchronize their actions, while Mutex ensures exclusive access to shared resources, mitigating the risks of data corruption and race conditions. Through detailed descriptions, comprehensive implementation codes, and rigorous real-time experimentation, it showcases the robustness and effectiveness of Semaphore and Mutex in ensuring the seamless operation of real-time systems on resource-constrained platforms like Arduino. This study contributes valuable insights into the practical utilization of synchronization primitives in embedded systems, paving the way for enhanced system reliability and performance.

Keywords: Real-time operating systems, Semaphore, Mutex, FreeRTOS, Arduino, Embedded systems, Multitasking, Resource management, Synchronization, Task coordination.

I. INTRODUCTION

Real-time operating systems (RTOS) have emerged as indispensable tools in the development of embedded systems, facilitating efficient multitasking and task prioritization. These systems find applications across various domains, from automotive and aerospace industries to consumer electronics and IoT devices.

The advent of platforms like Arduino has democratized the accessibility of RTOS, enabling their implementation even on low-cost microcontrollers with limited resources. However, the concurrent execution of multiple tasks in RTOS environments introduces synchronization and data access challenges that must be addressed to ensure the correct operation of the system.

Semaphore and Mutex are two fundamental synchronization techniques employed in RTOS to manage access to shared resources and coordinate the execution of concurrent tasks. Semaphore serves as a signaling mechanism, allowing tasks to communicate and synchronize their actions effectively. It comes in two primary types: Binary Semaphore, which restricts access to a shared resource to one task at a time, and Counting Semaphore, which permits multiple tasks to access the resource within defined limits.

On the other hand, Mutex, short for mutual exclusion, provides a means of ensuring exclusive access to shared resources. In an RTOS environment, Mutex plays a critical role in preventing race conditions and conflicts that may arise from simultaneous access to shared resources by multiple tasks. When a task attempts to acquire a Mutex that is already held by another task, it is blocked until the Mutex becomes available, thereby enforcing mutual exclusion.

This paper, delves into the utilization of Semaphore and Mutex synchronization techniques in FreeRTOS on the Arduino board for controlling LEDs. The objective is to explore the practical implementation of these synchronization mechanisms and evaluate their effectiveness in ensuring proper task execution and data consistency in real-time systems.

Through detailed descriptions, implementation codes, and real-time results. The paper aims to provide insights into the intricacies of Semaphore and Mutex usage in embedded applications, laying the groundwork for further exploration in this field.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.102 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 2, February 2024

DOI: 10.17148/IJARCCE.2024.13246

II. DESCRPTION

Semaphore and Mutex are fundamental synchronization primitives employed in real-time operating systems (RTOS) to manage concurrent access to shared resources and coordinate the execution of tasks. Understanding their concepts, types, and implementation is crucial for effectively designing and developing RTOS-based applications. This section, delves into Semaphore and Mutex in detail, exploring their characteristics, usage scenarios, and implementation nuances.

Semaphore serves as a signalling mechanism for task synchronization in RTOS environments. It enables tasks to coordinate their actions and manage access to shared resources effectively. There are two main types of semaphores as shown in Fig. 1 namely: Binary Semaphore and Counting Semaphore.

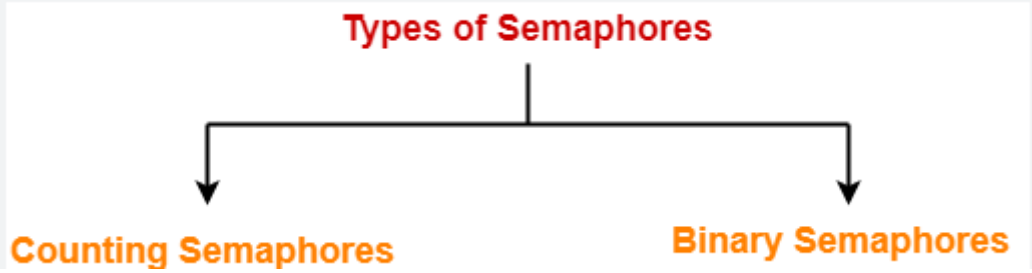

Fig. 1 Types of Semaphores

Binary Semaphore: Binary Semaphore has two states, typically represented by integer values 0 and 1. It acts as a mutex and is commonly used to control access to shared resources, ensuring mutual exclusion. When a task acquires a Binary Semaphore (sets it to 1), it gains access to the resource, while other tasks are blocked until the Semaphore is released (set back to 0).

Counting Semaphore: Counting Semaphore maintains a count of available resources or events. It allows multiple tasks to access a shared resource within defined limits. Each time an event occurs, the Semaphore count is incremented, and tasks can acquire the Semaphore until the count reaches its maximum value. Counting Semaphore is useful for scenarios where multiple resources need to be accessed concurrently.

Mutex, short for mutual exclusion, is a specialized form of Semaphore that ensures exclusive access to shared resources. It prevents race conditions and conflicts arising from simultaneous access by multiple tasks. In an RTOS environment, Mutex is typically implemented as a special type of Semaphore that allows only one task to hold the Mutex at a time. When a task attempts to acquire a Mutex that is already held by another task, it is blocked until the Mutex becomes available.

Mutexes are essential for protecting critical sections of code and ensuring data integrity in multitasking environments. As shown in Fig. 2 , they provide a mechanism for enforcing mutual exclusion, thereby preventing data corruption and maintaining system stability.

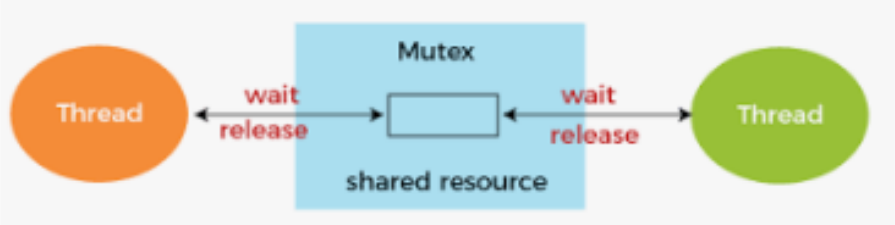

Fig. 2 Mutex

Implementing Semaphore and Mutex in RTOS requires careful consideration of task priorities, resource access patterns, and synchronization requirements. Tasks must be appropriately designed to acquire and release Semaphore and Mutex objects in a coordinated manner to avoid deadlocks and priority inversion issues.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.102 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 2, February 2024

DOI: 10.17148/IJARCCE.2024.13246

In practical implementations, Semaphore and Mutex are created using RTOS-specific APIs provided by the kernel. Tasks interact with Semaphore and Mutex objects through these APIs, acquiring and releasing them as needed to synchronize their execution and access shared resources.

III. IMPLEMENTATION

Fig. 3 shows the basic connection Arduino board with two LEDS anode pins are connected to the digital pins 7 and 8 and push button is connected to digital pin 2. The cathode pins are grounded. After making connections, the codes of semaphore and Mutex are uploaded to Arduino board.

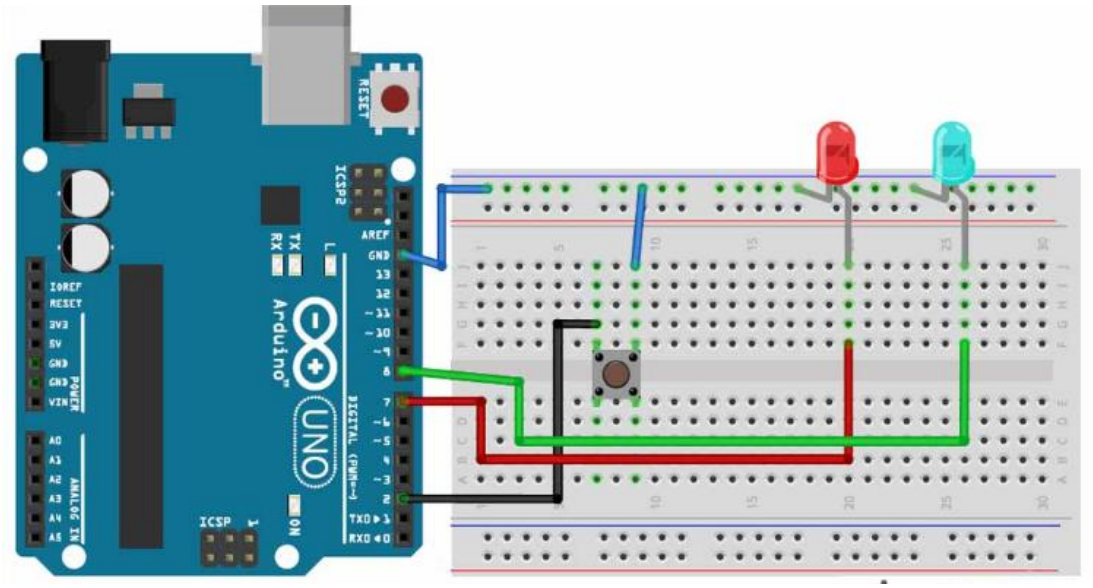

Fig. 3. Schematic diagram of the circuit

Semaphore Implementation: The code begins with including the Arduino_FreeRTOS.h header file, which provides access to FreeRTOS APIs for Arduino-compatible boards. This header file is essential for utilizing FreeRTOS functionalities. A variable of type SemaphoreHandle_t is declared to store the values of the semaphore. This variable will be used to create and manipulate the semaphore throughout the code. Within the void setup() function, two tasks, TaskLED and TaskBlink, are created using the xTaskCreate() API provided by FreeRTOS.

These tasks are responsible for controlling the LEDs connected to the Arduino board. Additionally, a semaphore is created using the xSemaphoreCreateBinary() function. This semaphore will be used to synchronize access to shared resources between tasks. The priority of tasks is set to be equal initially, and this can be adjusted for experimentation. Pin 2 is configured as an input, with the internal pull-up resistor enabled, and an interrupt is attached to this pin. Finally, the scheduler is started to begin task execution.

The ISR (Interrupt Service Routine) function is implemented to handle the interrupt triggered by pin 2. This function ensures the proper functioning of the interrupt by addressing the debounce issue of the pushbutton. This is typically achieved using millis() or micros() functions to measure time and adjust the debouncing time accordingly. Once the debounce issue is resolved, the interruptHandler() function is called. Inside the interruptHandler() function, the xSemaphoreGiveFromISR() API is invoked to give a semaphore to TaskLED, indicating that it can proceed with its operation, such as turning on the LED.

The TaskLed() function is created to handle the LED control task. Within the while loop of this function, the xSemaphoreTake() API is called to attempt to take the semaphore. If the semaphore is successfully taken (indicated by pdPASS, which equals 1), the LED is toggled. This ensures that the LED operation is synchronized with the availability of the semaphore, preventing concurrent access issues. Additionally, a function is created to blink the other LED connected to pin 7. This function operates independently of the semaphore mechanism and serves as a secondary task for LED control, demonstrating multitasking capabilities.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.102 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 2, February 2024

DOI: 10.17148/IJARCCE.2024.13246

Mutex Implementation: The header files required for mutex functionality are included, ensuring that the necessary functions and definitions are available for the code. These header files remain consistent across different implementations. A variable of type SemaphoreHandle_t is declared to store the Mutex values. This variable serves as a handle to the Mutex, allowing for its creation and manipulation within the code. Within the void setup() function, the serial monitor is initialized with a baud rate of 9600. This enables communication with the Arduino board via the serial interface, facilitating debugging and data output. Additionally, two tasks, Task1 and Task2, are created using the xTaskCreate() API provided by FreeRTOS. These tasks are responsible for executing specific functions concurrently. Furthermore, a Mutex is created using the xSemaphoreCreateMutex() function. This Mutex will be utilized to synchronize access to shared resources between the two tasks. Both tasks are assigned equal priorities initially, ensuring fair execution.

The task functions for Task1 and Task2 are implemented to define their behavior. Within the while loop of each task function, the Mutex is acquired using the xSemaphoreTake() function before any critical operation is performed. This ensures that only one task can access the shared resource at a time, preventing conflicts and race conditions. Once the Mutex is successfully acquired, the task proceeds with its operation, which in this case involves printing a message to the serial monitor. After completing the critical section, the Mutex is released using the xSemaphoreGive() function, allowing other tasks to acquire it. A delay is typically added after releasing the Mutex to ensure proper task scheduling and prevent resource contention. The void loop() function remains empty in this implementation as it is not necessary for the functionality of the code. The tasks created in the setup() function continue to execute indefinitely, interacting with the shared resource using the Mutex for synchronization.

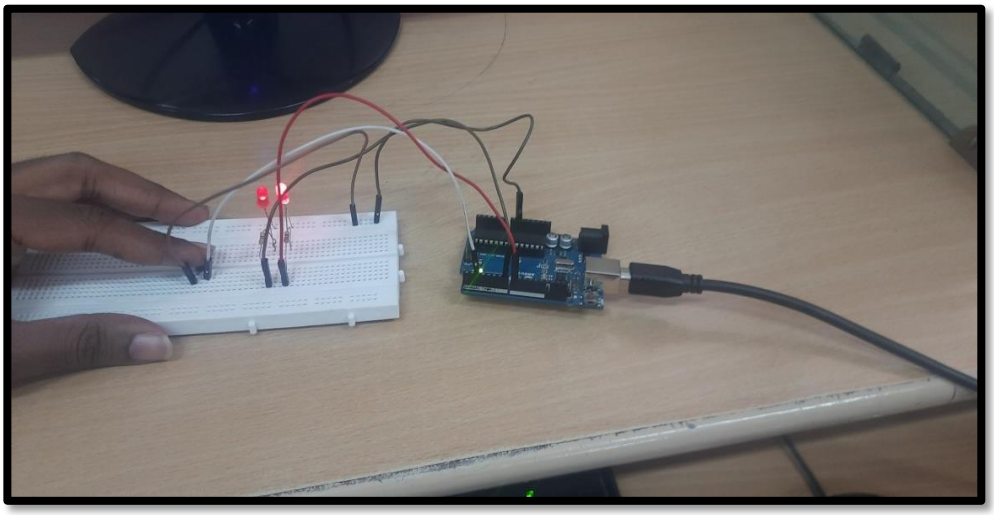

Fig. 4. Real Time Implementation

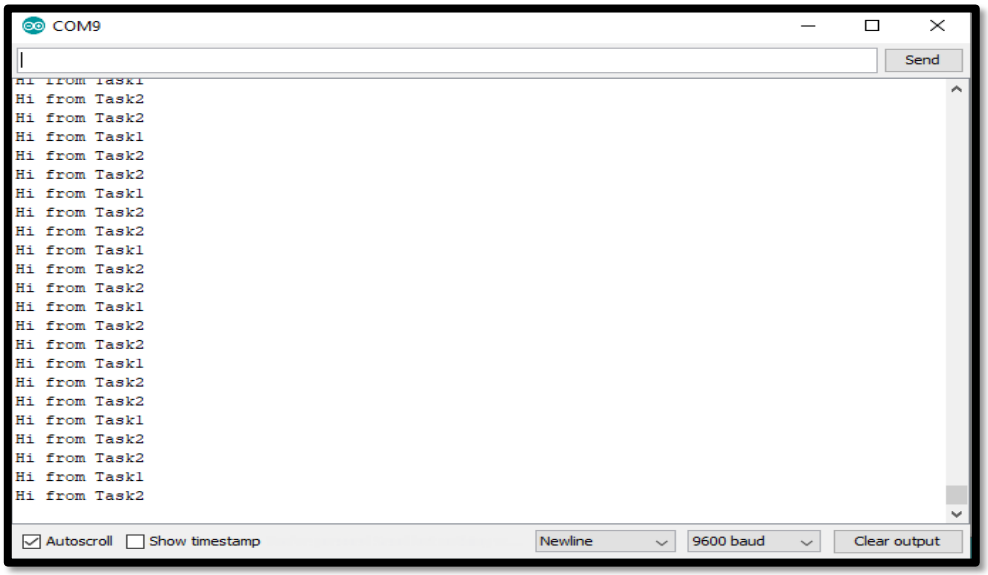

Fig. 5. Mutex Output

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.102 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 2, February 2024

DOI: 10.17148/IJARCCE.2024.13246

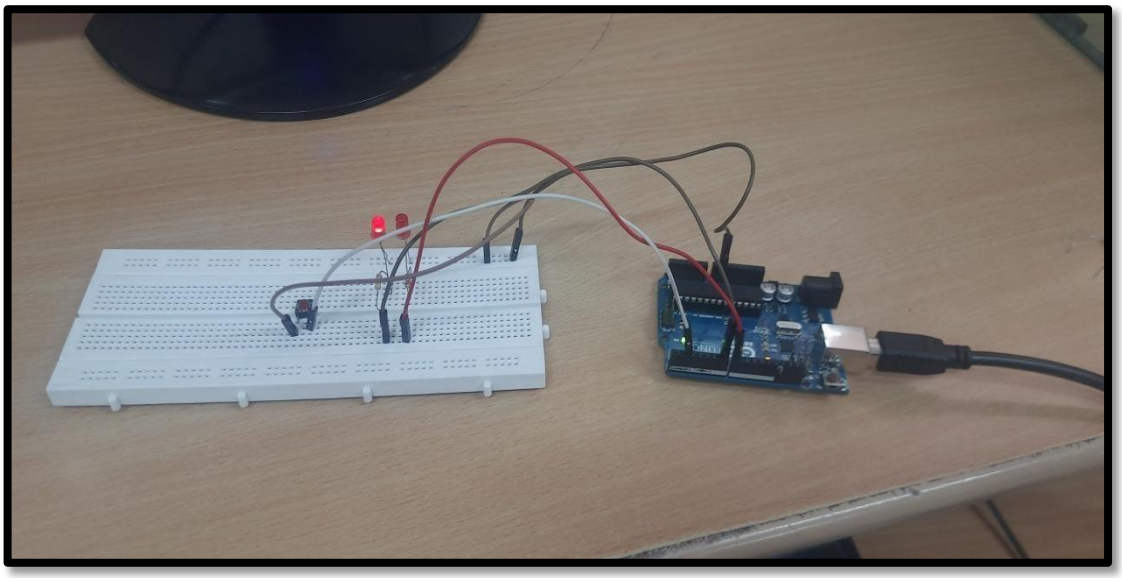

Fig. 6. Semaphore Output when button is not pressed

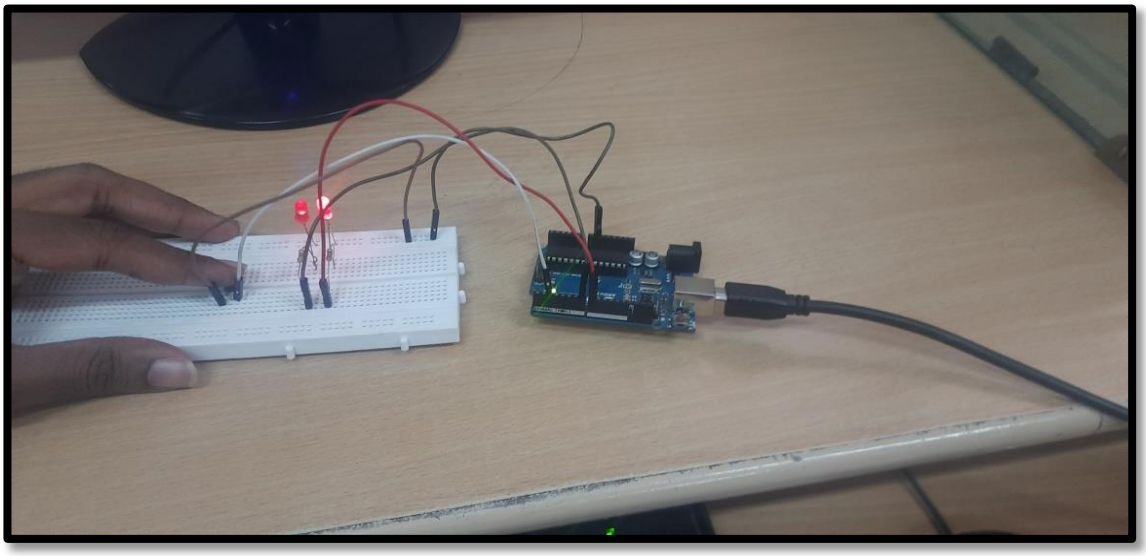

Fig. 7. Semaphore Output when button is pressed

IV. CONCLUSION

In conclusion, the paper has provided a comprehensive exploration of Semaphore and Mutex synchronization techniques in FreeRTOS on the Arduino board for controlling LEDs. Through detailed descriptions, implementation codes, and real-time results, It has demonstrated the practical application and effectiveness of these synchronization mechanisms in real-time embedded systems.

Here, is the summary of the key findings and implications in the study:

Effective Task Synchronization: Semaphore and Mutex have proven to be effective in synchronizing tasks involved in controlling LEDs. By managing access to shared resources, these synchronization techniques prevent conflicts and race conditions, ensuring proper task coordination and data consistency.

Prevention of Data Corruption: Mutex, in particular, plays a crucial role in preventing data corruption by enforcing mutual exclusion. By allowing only one task to access a shared resource at a time, Mutex prevents simultaneous writes or reads that could lead to data inconsistencies.

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.102 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 2, February 2024

DOI: 10.17148/IJARCCE.2024.13246

Real-time Performance: The implementation of Semaphore and Mutex in FreeRTOS on the Arduino board has demonstrated satisfactory real-time performance. Tasks were able to synchronize effectively without significant delays or interruptions, showcasing the efficiency of these synchronization mechanisms in embedded systems.

Scalability and Flexibility: The work's modular design allows for scalability and flexibility in accommodating additional tasks and shared resources. Semaphore and Mutex can be easily adapted to control access to various resources beyond LEDs, making them versatile solutions for synchronization in diverse embedded applications.

Enhanced System Reliability: By ensuring proper task synchronization and data integrity, Semaphore and Mutex contribute to enhanced system reliability and robustness. Embedded systems relying on FreeRTOS can benefit from the implementation of these synchronization techniques to minimize errors and failures.

The paper lays the groundwork for further exploration and enhancements in the realm of task synchronization in embedded systems. Future research could focus on optimizing Semaphore and Mutex implementations for specific application scenarios, as well as exploring advanced synchronization techniques and strategies.

REFERENCES

[1]. L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," in IEEE Transactions on Computers, vol. 39, no. 9, pp. 1175-1185, Sept. 1990.
[2]. B. Lehman and S. Dhingra, "Comparison of Real-Time Operating Systems for Embedded Systems," in International Journal of Engineering Trends and Technology (IJETT) – Volume 4 Issue 8- August 2013.
[3]. A. D. Carullo, L. Abeni and A. Cilardo, "Improving RTOS performance by combining priority ceiling emulation and stack resource policy," 2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Montreal, QC, 2012, pp. 1-6.
[4]. A. N. Toosi, "A survey on real-time operating systems," 2012 International Symposium on Industrial Electronics, Hangzhou, 2012, pp. 497-501.
[5]. M. Antoni and S. Evain, "Real-time operating systems for embedded computing: A survey," 2015 International Conference on Control, Decision and Information Technologies (CoDIT), Paris, 2015, pp. 516-521.
[6]. P. Fan, Z. Zhang and S. Zhang, "The design and implementation of real-time operating system based on FreeRTOS," 2010 2nd International Conference on Advanced Computer Control (ICACC), Shenyang, 2010, pp. 206-209.
[7]. F. M. Soares, "Design and implementation of a real-time operating system kernel," 2008 XXVI Brazilian Symposium on Computer Networks and Distributed Systems, Campos do Jordao, Brazil, 2008, pp. 133-144.
[8]. P. Langendoerfer and M. Wollschlaeger, "Operating Systems for Wireless Sensor Networks: A Survey," in IEEE Internet of Things Journal, vol. 4, no. 6, pp. 2062-2073, Dec. 2017.
[9]. S. Kim, H. Shin and Y. Shin, "Survey of Real-Time Operating Systems for Wireless Sensor Networks," 2011 International Conference on Information Science and Applications, Jeju, 2011, pp. 1-6.
[10]. L. Benini and G. De Micheli, "Dynamic voltage scaling and power management for portable systems," in IEEE Design & Test of Computers, vol. 17, no. 5, pp. 32-41, Oct. 2000.

