INDIAN JOURNAL OF SCIENCE AND TECHNOLOGY

RESEARCH ARTICLE

Benchmarking and Comparison of Two Open-source RTOSs for Embedded Systems Based on ARM Cortex-M4 MCU

Yahia Mazzi[1]*, Ahmed Gaga[2], Fatima Errahimi[1]

1 Laboratory of Intelligent Systems, Georesources and Renewable Energies (LISGRE), Sidi Mohamed Ben Abdellah University Fez, Box 2202 Fez, Morocco. Tel.: +212-622-35-1485
2 Physic department, Sultan Moulay Slimane University Polydisciplinary Faculty of Beni Mellal, Morocco

OPEN ACCESS

Received: 06.03.2021
Accepted: 13.04.2021
Published: 06.05.2021

Citation: Mazzi Y, Gaga A, Errahimi F (2021) Benchmarking and Comparison of Two Open-source RTOSs for Embedded Systems Based on ARM Cortex-M4 MCU. Indian Journal of Science and Technology 14(16): 1261-1273. https://doi.org/10.17485/IJST/v14i16.387

* **Corresponding author**.

Tel: +212-622-35-1485
yahia.mazzi@usmba.ac.ma

Funding: None

Competing Interests: None

Published By Indian Society for Education and Environment (iSee)

ISSN
Print: 0974-6846
Electronic: 0974-5645

Abstract

Objective: To evaluate the performance of two open-source real-time operating systems (RTOSs), Keil RTX5 and FreeRTOS. Besides, a comparison between them has been also established based on four timing metrics: task switching time, pre-emption time, semaphore shuffling time, and inter-task messaging latency. All the tests have been performed on an STM32F429 discovery board based on Cortex-M4 MCUs. **Methods**: To measure the timing metrics, the ARM cycle counter register implemented in the DWT unit was used. **Findings**: The DWT cycle counter allows us to capture the number of cycles that occurred in the execution of a part of the code. Therefore, the time measurements of the metrics selected show that FreeRTOS has good performance with the lowest value of switching time, preemption time, and semaphore shuffling time. Instead, Keil RTX5 has fast message passing. **Novelty**: The study provides an evaluation and comparison of the latest version of the most used open-source RTOSs, Keil RTX5 and FreeRTOS v10.2.0. Furthermore, the timing metrics have been measured accurately with the ARM cycle counter register without using any other hardware or GPIO pin that may disturb the measurement. The comparison is based on four critical timing metrics that affect mostly the performance of any RTOS and define their time capability. Finally, the tests have been made on a low-power ARM MCU.

Keywords: realtime operation system; embedded system; FreeRTOS; Keil RTX5; benchmarking metric; ARM MCU

1 Introduction

From the first modern embedded system "Apollo Guidance Computer" used for the guidance system of the Apollo mission in 1967, until this day, these systems receive high interest from many industrial domains like automotive, aerospace, telecommunication devices, and home appliances that satisfy our daily life, work, and gaming demands. Therefore, this demand imposes adaptability to dynamic operation situations, fast Time to Market, conformity to the industrial standards, reliability, and safety. Besides, embedded systems constitute 99 % of all processors manufactured in the world [(1)].

The growth of sophisticated, low cost and high-performance processors designed for embedded systems[2] offers to developers a good infrastructure for almost all of their applications. But sometimes with a simple or ever-shorter project, it's becoming impossible to control all these devices without the use of advanced software (SW). Conventionally, the embedded system programs are created using super-loop programming (named also beta-metal or background programming), which means that all tasks will be run in an infinite loop all the time and will be executed sequentially. Thus, the microcontroller will stop execution only if there is an interrupt event or power shutting down. But this kind of programming also has a start-up code to perform the hardware initialize, define the interrupt service routines and exception, initialize the stack pointer and allocate memory for it, and so on.

Unfortunately, the rise of embedded SW complexity takes more resources such as power consumption[3],[4], memory usage[5], size, and the number of processors. Consequently, a Real-Time Operating System (RTOS)[6] is a suitable solution to implement such complex embedded SW and cohere it to the target architecture. As illustrated in Figure 1, the architecture of the embedded system contains a software layer and hardware layer, including hardware and software components where the RTOS takes place in the system software layer.

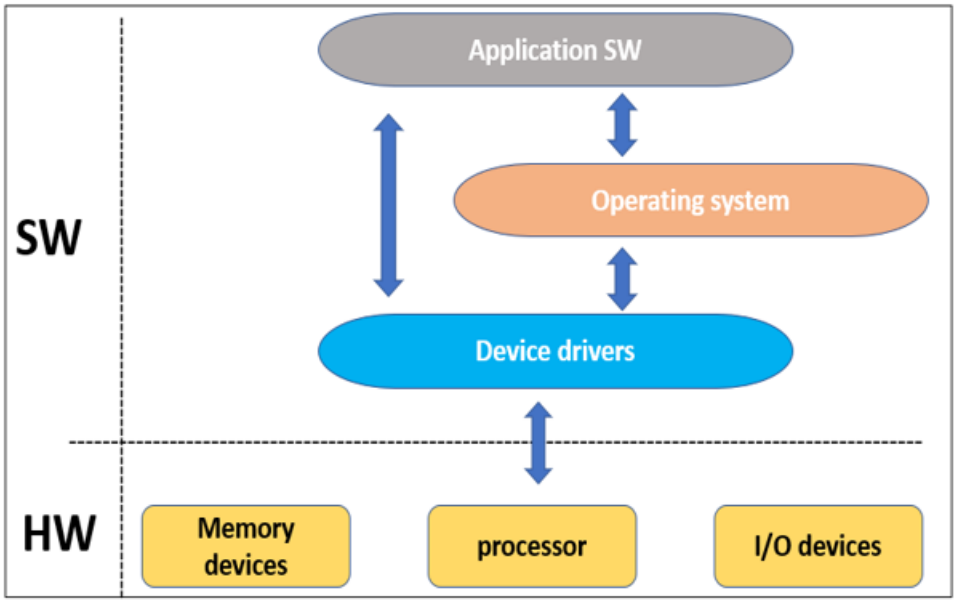

Fig 1. Typical layers of an embedded system

The RTOS contains two terms, Real-time, and operating system. The first one is the real-time system, it is the system that can process the inputs and provide the output in a deterministic time, so it must satisfy the physical interaction with the real world and time requirement of these interactions. Based on this time requirement we can identify two types of RT system: hard-real time system assures that the critical tasks complete their execution on time. However, the critical real-time task in the soft-real time gets priority over other tasks and retains this priority until it completes. For both systems, the delay between the recovery of the stored data to the time that the operating system takes to finish the request made for it must be bounded.

The second is the Operation System (OS), which is simply defined as a software program that interfaces the user and the hardware components of the computer. Windows, Linux, Unix are examples of the operating system in which they operate under the general-purpose operation system (GPOS), those systems cannot be run on small microcontrollers due to their resource requirements and the use of virtual memory.

A real-time operating system (RTOS) is a part of the software with a set of APIs for users to develop their applications, where the actions are divided into separate threads or tasks. It has the job like any OS, it's responsible for hardware resource management, scheduling program applications, memory management, etc. However, when this OS must handle numerous events and ensure that the response to these events is within a finite and strict time (or timing requirements), we called it an RTOS. In addition to the wide variety of services they offer such as scheduling, I/O operations, and communication between processes, these systems efficiently manage the power consumption of the whole embedded system[7].

Several RTOSs are existing nowadays, divided into commercial, open-source, and proprietary RTOSs, and available for high-performance microcontrollers [8] and small microcontrollers [9],[10]. Examples of RTOS include FreeRTOS, RTX, OSEK, VxWorks, μC/OS-II, μC/OS-III, LynxOS, MbedOS, RT-thread, Nucleus, etc. Furthermore, these RTOSs are implemented in small microcontrollers and widely used in several domains such as automotive [11–13], Avionic [14], mobile, and the internet of things (IoT) [15,16], and robotic [17].

To develop a multitasking application, RTOSs offers several services and features such as the task priority to provide to each task an individual level of priority, so the RTOS can identify which task must enter in the running state. The selection of this task, with the higher priority, is achieved by task management service, constituted of two fundamentals types of scheduling: pre-emptive and non-preemptive. In pre-emptive scheduling, the running task can be pre-empted by a higher priority task that becomes in the ready state. The use of the stack and interrupt management is very important in this case, to ensure the correctness of the context switching since the pre-emption can occur at any time. Thus, when the task resumes its execution, the stack values are restored. Contrarily, in the non-preemptive (or cooperative) scheduling all tasks cooperate to execute the context switching, clearly saying, the running task voluntarily quit the CPU. Even though the non-preemptive scheduling policy is simple, it is less used in common RTOS scheduling because it cannot always meet the time deterministic nature. Therefore, pre-emptive scheduling is the most used in RTOS since it gives a short time response for critical actions.

Some other services offered by RTOS is inter-task communication and synchronization mechanisms such as mutexes, semaphores, message queues, mailboxes, signal events, etc.

Before selecting a specific RTOS for an application, several criteria must be taken into account such as predictability, the Worst-case Response Time (WCRT) [18], the response time of a task to an external/internal event, the inter-task communication, the response to an interrupt service routine, and the synchronization mechanism, etc. In this paper, we aim to measure, compare, and analyze the timing of some of those criteria for two open-source RTOSs: FreeRTOS and Keil RTX5. All the benchmarking tests have been established on a small microcontroller ARM Cortex®-M4 based MCU.

The paper contributes with a new result of the timing comparison of two of the most open-source RTOSs [19] used on the small microcontroller. Moreover, the test codes have been run on an actual MCU (STM32F429 discovery board based on an ARM Cortex®-M4 MCU). The test includes four micro-benchmarking parameters: task switching time, pre-emption time, semaphore shuffling time, and the inter-task messaging latency. Compared to other works, where an oscilloscope is used to perform the time measurements on a GPIO (General-purpose Input/Output) Pin selected from the board, the time measuring in this paper is done by the Cycle Counter feature implemented on the ARM Cortex-M4 MCU in the DWT (Data Watchpoint & trace) unit. Additionally, we select the deterministic Keil RTX5 RTOS based on the new version of CMSIS-RTOS API (CMSIS-RTOS v2) and the new version of FreeRTOS (version 10.2.0).

This paper is organized as follows: section 2 represents a general insight into the real-time operation system, Section 3 provides some related works and the purpose of our work. Following this, Section 4 defines the performance metrics, which will be measured, and the experimental setup. Experimental results and discussions of the benchmark results are drawn in Section 5. The paper finishes with a conclusion in Section 6.

2 Real-Time Operating System

Every RTOS has a Kernel, which is the supervisor core software that provides protection, scheduling, resources management, and other services; it also combines several modules, as shown in Figure 2, including protocol networks, system files, and other devices. The RTOS execution refers to multiple tasks sharing a processor (i.e., multitasking), performed with a set of APIs.

Before giving some details about the RTOSs selected for the benchmarking test and comparison, let's take a look into some related statistics done in 2019 by the Embedded Market study [19] about the use of RTOS for the current and the ongoing projects. The availability of source codes and the cost of the RTOS software is one of the biggest factors in deciding whether to use it in an embedded application, as the study shows that 42% of programmers use an open-source RTOS without any commercial support. The reason for this choice, according to the study, is that the open-source with current solutions works fine for them, and the commercial alternatives are too expensive. For the persons who choose the commercial OS, the real-time capability is the most influential feature of their choice.

For the RTOS selected in this paper, 19% of programmers use FreeRTOS, and 4% use Keil RTX. These numbers will increase for the ongoing projects to 27% and 6% respectively.

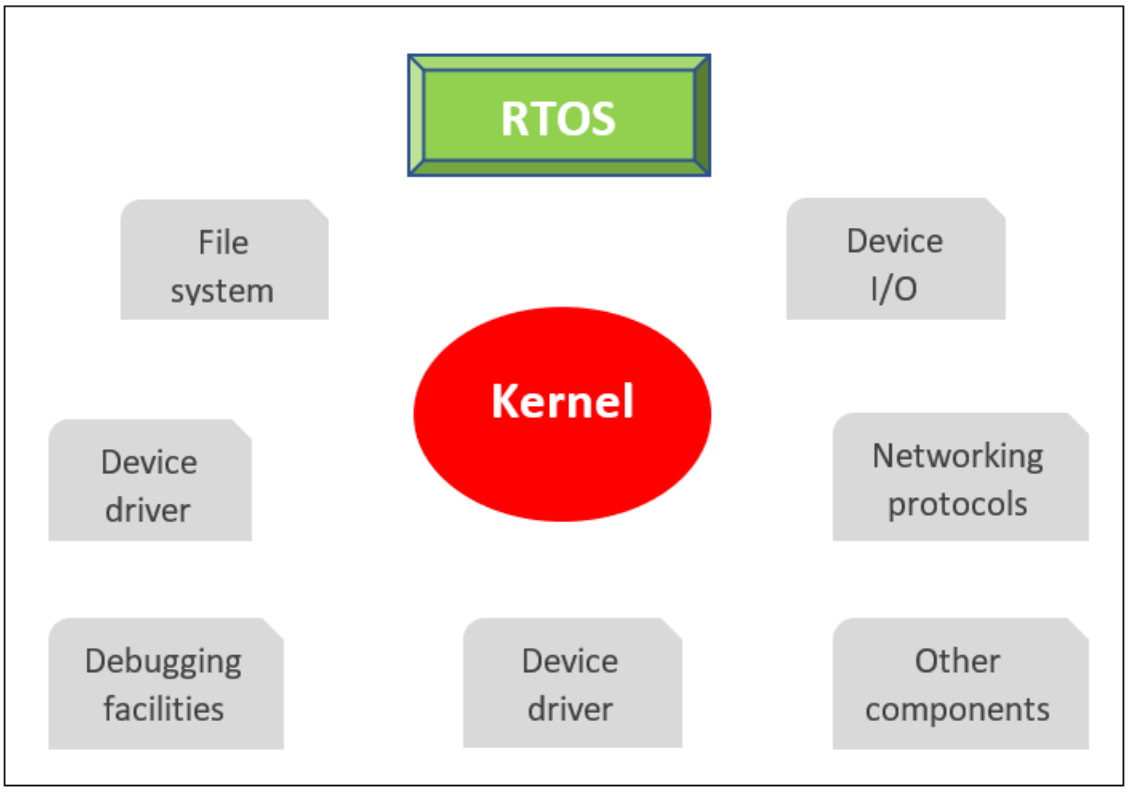

Fig 2. Typical RTOS architecture

2.1 FreeRTOS

FreeRTOS[20] is a fully open-source RTOS distributed under free MIT licensing. Available for over 15 years and now it is the leader in the market of RTOS[19] for the microcontroller and small microprocessors.

FreeRTOS supports more than 40 MCU architectures and more than 15 toolchains, including the newest RISC-V[21] and ARMv8-M (Martin, 2016)[22]. In addition to its simple and easy use, it provides an official demo project for each new MCU architecture, which can be downloaded from the FreeRTOS website.

2.2 Keil RTX5

Designed for ARM and Cortex architecture, the version 5 Keil RTX (RTX5)[23] based on CMSIS-RTOS v2 APIs allows embedded system developers to create a powerful application, well-structured with multiple functions, and easy to maintain. It comes with a free royalty license. The CMSIS2 package for Keil MDK includes the RTX5 Kernel accompanying with necessary file and library.

This advanced RTOS provides various benefits, such as memory management, Interrupt Service Routine (ISR) system management, many kinds of inter-task communications. Additionally, The RTX5 Kernel scheduler supports cooperative, round-robin, and the famous pre-emptive multitasking policies.

2.3 The main characteristics of RTX and FreeRTOS

Table 1 presents a general insight into the main characteristics and differences between Keil RTX5 and FreeRTOS.

Table 1. The main characteristics of Keil RTX5 and FreeRTOS

Parameters	Keil RTX5	FreeRTOS
OS family	RTOS	RTOS
Architectures supported	ARM Cortex-M MCU based devices	ARM, Cortus, RISC-V, PIC, AVR, Renesas, x86, Infineon, …
Scheduling policies	Pre-emptive, round-robin, cooperative	Pre-emptive and cooperative
Synchronization and resources management	Semaphores, Mutexes	Semaphores (binary & counting), Mutexes
Memory allocation for Objects	• Global Memory Pool • Object-specific Memory Pool • Static Object Memory	Automatic or Manual Dynamic Allocation from RTOS heap.
Interrupt	• Interrupt not disabled • Low interrupt latency	Subscription mechanism to handle interrupt sub-routines
Inter-task communication	Signal events, Message Queue, Memory Pool, Mail Queue	Message Queue
Source model	Open-source	Open-source

3 Related work

Besides the statistical study done by "EETimes and Embedded.com"[19] in 2019 that reflects the audience's usage of the real-time operating system on their embedded system projects, there are also several pieces of research and studies for the RTOSs, including their portability on small microcontrollers, timing evaluations and comparison based on some metrics parameters.

By using the verification system Hip/Sleek, the authors[24] present the process used to automatically verify the memory safety and functional correctness of the task scheduler component of the FreeRTOS Kernel. Furthermore, In[25], Xing et al. give the process to build an embedded system based on FreeRTOS on a Cortex-M3 MCU. The paper demonstrates also that the porting method is feasible and reasonable by doing several tests. Likewise, the authors in[26] propose a real-time robot operating system based on FreeRTOS combined with a publisher/subscriber communication mechanism in ROS (Robot Operating System). Another qualitative and quantitative comparison is done in[27] between FreeRTOS V8.0.0 and μC/OS-III. The results illustrated in Table 2 show that μC/OS-III becomes somewhat unpredictable due to the latency of the tick interrupt and FreeRTOS is stable in some metrics. In brief, μC/OS-III slightly outperforms FreeRTOS in most of the measured metrics.

Table 2. Comparison of FreeRTOS and μC/OS-III[27]

Metrics RTOS	Mutex acquisition time (μs)	Mutex release time (μs)	Average Interrupt Latency	Max Semaphore acquire Time (μs)	Max Semaphore release time (μs)
FreeRTOS V8.0.0	25.7	27.4	0.9	65	34
μC/OS-III	53.6	37.3	1.00	20	24

The analyzed results of the study done by[28] show that there is no performance penalty of CMSIS-RTOS when it is designed to follow the standard APIs. So, the authors propose to use an adaptation layer for the X real-time kernel, which can impose a significant performance if the type of services of the kernel is different.

In[29], the authors compare the time switch context between a low priority task and a high priority task for four RTOSs (FreeRTOS, Keil RTX, μC/OS-II, RT-Thread). The results of the experimental tests performed on ARM Cortex®-M0+ and ARM Cortex®-M4 based MCUs show that FreeRTOS has the largest context switching time, compared to Keil RTX which has the best performances. The same authors add FreeRTOS version 10.2.0 and the μC/OS-III RTOS to the experiment in[30] and extend it to measure and compare the performance achieved by those six RTOS for the send/receive latencies of an event, mailbox, and semaphore as shown in Table 3. Additionally, they also measure the time for task switching from a high priority task to a low priority task. The results of the experiment show that μC/OS-III, Keil RTX, and RT-Thread have the best performance, contrary to FreeRTOS which has a low performance even though it is the most open-source RTOS used in embedded applications.

Table 3. A recapitulation of the results of[30]

Metrics (μs) RTOS		FreeRTOS 9.0.0		FreeRTOS 10.2.0		RT-thread		Keil V4.82.0	RTX	μC/OS-III		μC/OS-III	
		Cortex™ M0+	Cortex™ M4	Cortex™ M0+	Cortex™ M4	Cortex™ M0+	Cortex™ M4	Cortex™ M0+	Cortex™ M4	Cortex™ M0+	Cortex™ M4	Cortex™ M0+	Cortex™ - M4
Tot	event	23.56	4.13	23.33	3.97	22.8	3.88	16	3.26	19.04	3.66	26.11	5.56
	Semaphore	24.86	5.26	25.06	5.05	19.29	2.76	15.43	3.03	18.77	3.41	23.77	5.26
	mailbox	20.12	3.65	19.05	3.07	24.79	4.43	17.41	3.57	19.33	3.57	24.60	5.38
Tot1	Event	25.43	4.72	24.78	4.99	29.88	5.46	28.75	3.50	16.6	3.25	25.23	5.26
	Semaphore	41.69	7.78	41.23	7.80	28.48	4.93	19.10	3.42	15.41	2.87	22.16	4.79
	Mailbox	19.73	3.56	19.35	3.08	29.72	5.36	19.95	3.47	15.38	2.91	24.01	5.21
Tet	Event	8.70	1.82	8.70	1.75	4.34	0.90	8.38	1.60	5.65	1.1	5.32	1.02
	Semaphore	8.67	2.16	8.81	2.09	3.68	0.66	7.44	1.33	4.09	0.69	4.88	0.86
	Mailbox	6.50	1.37	6.48	1.27	6.56	1.16	10.04	1.73	4.02	0.67	8.77	1.61
Tet1	Event	10.61	2.39	10.61	2.23	5.87	1.19	8.98	1.79	6.31	1.13	6.69	1.27
	Semaphore	6.25	1.41	7.85	1.81	3.68	0.68	8.45	1.43	4.93	0.9	4.34	0.83
	Mailbox	7.70	1.7	7.63	1.55	6.18	1.21	10.49	1.94	5.03	0.96	7.48	1.45

Tot -Time of task context switch from lower priority task to higher priority task is triggered by
Tot1 - Time of task context switch from higher priority task to lower priority task is triggered by
Tet - The execution time for primitive that signal
Tet1 - The execution time for primitive that sent

In this study, we focused just on two types of RTOS: FreeRTOS 10.2.0 and Keil RTX5. Both RTOS selected are free Royalty open-source RTOS. RTX5 is integrated into the Keil MDK-ARM software tools, and FreeRTOS is the most used RTOS in the embedded applications based on a small microcontroller. The selection of these RTOSs refers to their top use for the embedded projects based on small microcontrollers such as ARM Cortex®-Mx MCUs. The efficiency of these MCUs is proved by the great applications built by them, such as in data acquisition and controlling system with an embedded web server for industrial monitoring[31] using Keil RTX and TCP/IP Ethernet running on an ARM Cortex®-M3 MCU. Likewise, in[32] a power well covers monitoring system (PWC-MS) was implemented in an ARM Cortex®-M3 microcontroller with FreeRTOS and NB-IoT technology. Those are just some examples from a lot of applications based on FreeRTOS and Keil RTX, where they reflect their capability to manage a real-time system efficiently.

4 Performance Metrics and Experimental setup

For a specific embedded application, with a particular system requirement, some parameters are most important and others are insignificant. Hence, the benchmarking tool comes to solve this issue and offers to designers a method to evaluate and compare multitasking RTOSs and help them choose the appropriate RTOS for their applications. In this section of the paper, we will describe the experimental setup and explain all the tests done for the benchmarking evaluation, based on the timing measurement. So, in this work we will focus on four micro-benchmarking parameters: task switching time, pre-emption time, semaphore shuffling time, and the inter-task Messaging latency.

4.1 Performance metrics

To measure the timing performance of the RTOS selected, four metrics have been defined for the benchmarking test.

4.1.1 Task Switching Time

It is the average time the kernel takes to switch from one task to another with the same priority. The tasks mustn't be suspended or in sleeping mode. This metric is essential to assess the efficiency with which the kernel manages the data structure in restoring and saving contexts. Figure 3 demonstrates this performance parameter. The two tasks created for this metric have the same priority and switch the CPU between them in every iteration. The task switching time measured will be used to determine the specific time for the other metric that uses the task switch.

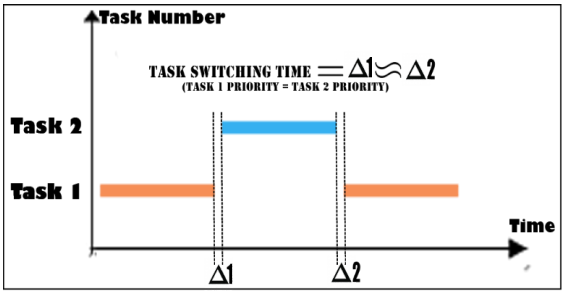

Fig 3. Task switching time

4.1.2 Preemption Time

The preemption time is the average time that takes a higher priority task (HPT) to take the CPU from a lower priority task (LPT) currently running on the CPU. This preemption usually occurs when the HPT responds to an external event and switch from the suspended or the blocked state to the ready state. By way of explanation, it is the average time that the kernel takes to switch control from a running LPT to the HPT activated by an external event. To measure this time, two tasks were created with different priorities. First, the HPT runs and suspends itself, and then the LPT switches to the running state and resumes the execution of the HPT. The process is repeated for the defined iterations. A demonstration of the pre-emption time is illustrated in Figure 4.

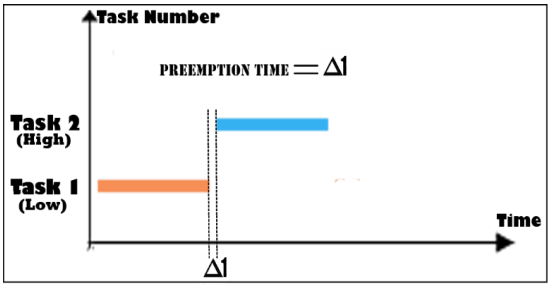

Fig 4. Preemption time

4.1.3 Semaphore Shuffling Time

The time elapsed between a task's release of a semaphore and the activation of another blocked task waiting for this semaphore is called semaphore shuffling time, it measures the overhead related to the mutual exclusion. This parameter gives the time that the kernel spends to provide a non-sharable resource from one task to another. The two tasks used for this metric have the same priority and pass the semaphore between them for a specified number of iterations. From the final time gotten, we subtract the time needed for task switching in order to only get the net semaphore shuffling time. Figure 5 demonstrates this performance metric.

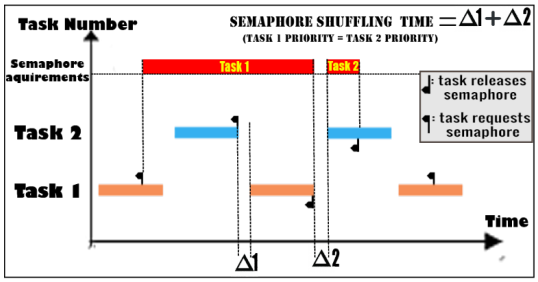

Fig 5. Semaphore shuffling

4.1.4 Inter-task Messaging Latency

The Inter-task Messaging Latency is the time elapsed between sending and receiving a nonzero-length message from one task to another. The receiving task should be suspended while waiting for the message and the sending task should stop execution after sending it, in order to properly measure this latency. Supposing that multiple messages are sent to the same receiving task, in this case, the multitasking kernel provides other alternatives such as queues and pipes, so the receiving task can read an old message before overwriting it by the sending task with a new one.

In Figure 6, we demonstrate this parameter. The LPT stays blocked until it receives a message queue from the HPT and gets the CPU, it turns blocked again when trying to receive another message. This process is also repeated for a specified number of iterations.

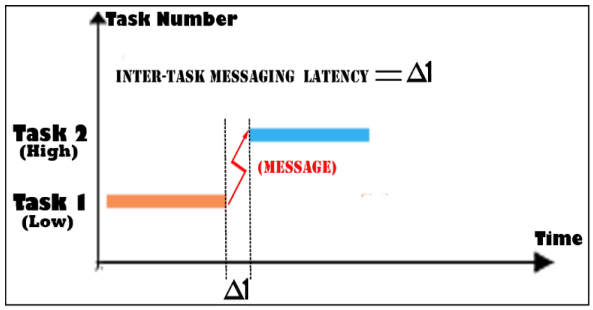

Fig 6. inter-task communication

4.2 Timing measurement process

One of many ways to measure the execution time of a part of code is using an oscilloscope and GPIOs pin. However, an amount of setup must be done before going through the measurement such as find free GPIO pins and configure them as outputs which cannot be possible in some applications where the number of pins is limited. Additionally, the measurement may be affected by some errors. To overtakes the aforementioned limitations, the DWT cycle counter is selected for time measurement. Where the number of CPU cycles can be captured easily and converted to milliseconds.

For all the bench marking metrics measured in this paper, we use the same process to develop the application code. We create two tasks, where the priorities depend on the parameter to measure. And to ensure an accurate measure, the tasks switch back and forth for 50000 iterations, so first, we determine the time needed to perform the for loops as illustrated in Figure 7 (with no work & without task switching). Then, we subtract this time from the total measured time and divide the result by the number of iterations in order to only get the time to perform the benchmark metric.

```
53    T3 = DWT->CYCCNT ;     // start cycle counting
54        for (count1 = 0; count1< 50000 ; count1++)
55        {
56        // noting to do
57        }
58
59      for (count2 = 0; count2< 50000 ; count2++)
60        {
61        // noting to do
62
63     T4 = DWT->CYCCNT ;  // start cycle counting
```

Fig 7. Number of cycles measurement for empty For loops

The figure (Figure 8) illustrates a bloc synopsis of the process to get the execution time of a portion of the code. We select to use a software measurement for the benchmark metrics included in the hardware platform facilities. Besides the RTOS Tick Timer used by the Kernel, we use the DWT cycle counter unit for the timing measurements (chapter 9 of[33]). The value of the other timers increases when an interrupt is issued, so if the program enters in a portion of code with interrupts disabled, the timing measurement will be delayed, thus choosing the DWT cycle counter solves the problem. By using this counter, and running the program in debugging mode, we can capture the number of cycles at the beginning (T1) and in the end (T2) of

the code to measure. The value of T1 and T2 are displayed in the MDK-ARM debug window. The total number of cycles equals T2-T1, then we divide this number by the clock frequency (80 MHz in our case) to get the time in μs.

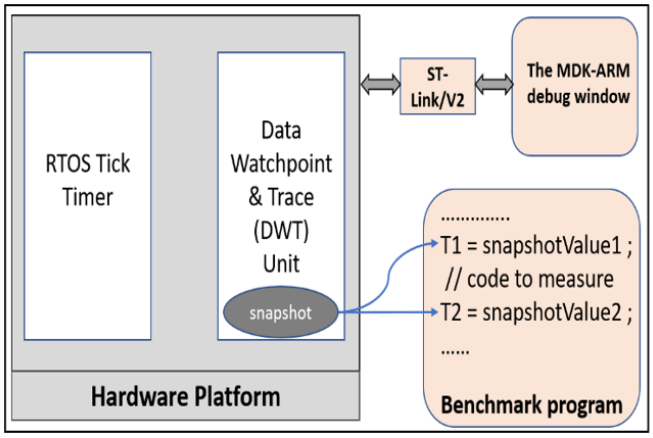

Fig 8. Execution time measurement of a portion of code using DWT cycle counter

Assuming that $\triangle_1 = T4 - T3$ is the number of cycles needed to perform empty for loops with 50000 iterations, and $\triangle_2 = T2 - T1$ is the number of cycles to perform 50000 iterations of task switching, preemption, semaphore passing, or inter-task messaging. Then, we calculate the value of $\triangle_2$value for each metric. The final number of cycles which correspond to the number of one iteration is equaled to:

$$N = \frac{\triangle_2 - \triangle_1}{50000} \tag{1}$$

Hence, the time of benchmark metrics is defined in microseconds (μs) as bellow:

$$Time\ (\mu s) = \frac{number\ of\ cycles}{frequency\ (MHz)} = \frac{N}{80} \tag{2}$$

4.3 Software/Hardware Configuration

The configuration of our project includes the kernel and board configuration, such as memory initialization, stack size initialization, output and input configuration, and so on.

The target hardware used to perform the benchmarking tests is a STM32F429ZIT6 Discovery board[34] Based on a 32-bit Cortex-M4 core[35] in an LQFP144 package with 2 Mbytes of Flash memory, 256 Kbytes of RAM, 180 MHz maximum system clock frequency, and an on-board ST-Link/V2-B. A code for each benchmark metric has been developed for each RTOS selected (FreeRTOS and RTX5) in the Keil uVision5 v5.29 under the same conditions in the same hardware platform. To take the value of the cycles counted by the DWT unit, the codes will be run and executed on debugging mode in order. The system speed frequency is configured at 80 MHz (PLL activated with the high-speed external clock of 8 MHz).

In the program software, we configured static priority-based scheduling for multitasking mechanism for both FreeRTOS and Keil RTX5 RTOSs, and a tick rate of 1000 Hz to generate a tick interrupt every 1ms, which makes a good balance between the overhead of the task switching and task speed. For FreeRTOS we used timer 6 to generate this RTOS kernel tick. As well, for Keil RTX5, the unified System Tick Timer (SysTick Timer) has been used. To put all the tests on the same conditions, we have used a minimum stack size of 512 Bytes for each task.

5 Experimental results and discussion

In this section, the results of the bench marking tests discussed in the previous sections will be presented and discussed.

The comparison of FreeRTOS and RTX5 has been done based on four parameters: task switching time, pre-emption time, Inter-task messaging latency, and semaphore shuffling time. For each parameter, we measure the total cycle for several iterations (ex: 50000) and calculate the time taken to perform one iteration. The cycle counter used gives us the number of cycles needed to establish a part of the code and displays it in the debugging window of uVision5 ARM-MDK as shown in Figure 9.

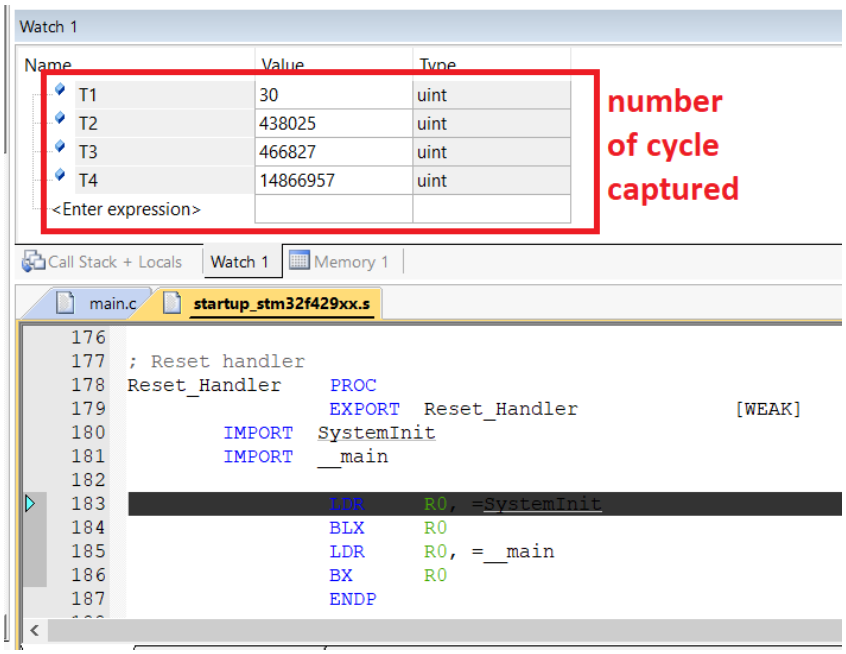

Fig 9. debugging window with the number of cycles captured

The results of the time measurement for each RTOSs in microseconds are illustrated in Figures 10, 11, 12 and 13. Table 4 shows the number of cycles of each benchmark parameter for both FreeRTOS and Keil RTX5.

Table 4. the number of cycles of each benchmark parameters for both FreeRTOS and Keil RTX5

RTOS Metrics		Task switching time	Preemption time	Semaphore shuffling time	Inter-task messaging latency
FreeRTOS	For 50000 iterations	6980810	18136803	14437209	76826808
	For one iteration	139.616	362.736	288.744	1536.536
Keil RTX5	For 50000 iterations	14106402	18609605	65959600	18051611
	For one iteration	282.128	372.192	1319.192	361.032

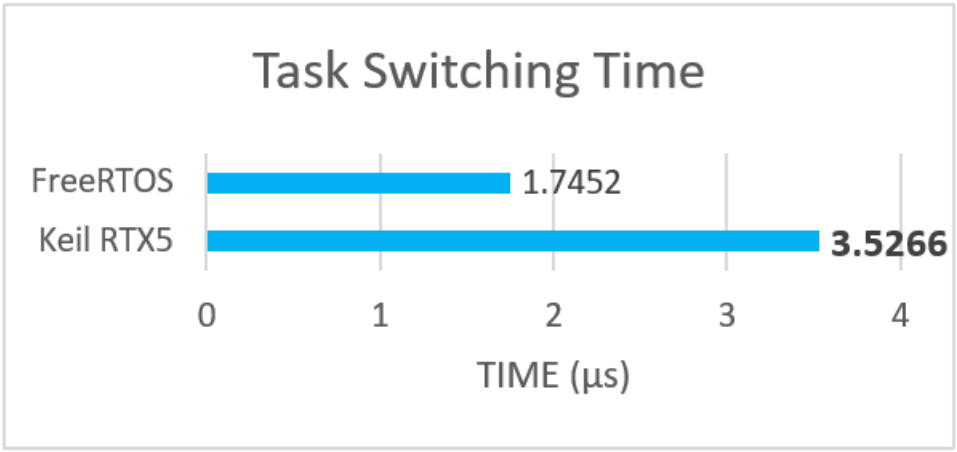

Fig 10. Measured task switching time

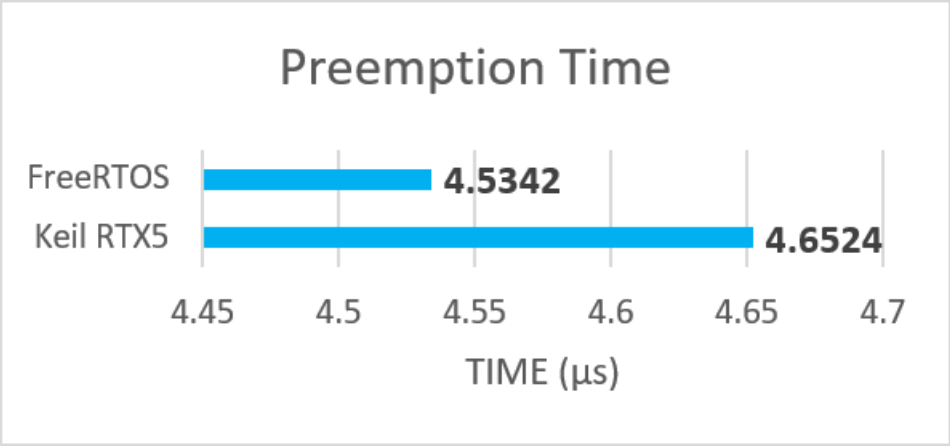

Fig 11. Measured preemption time

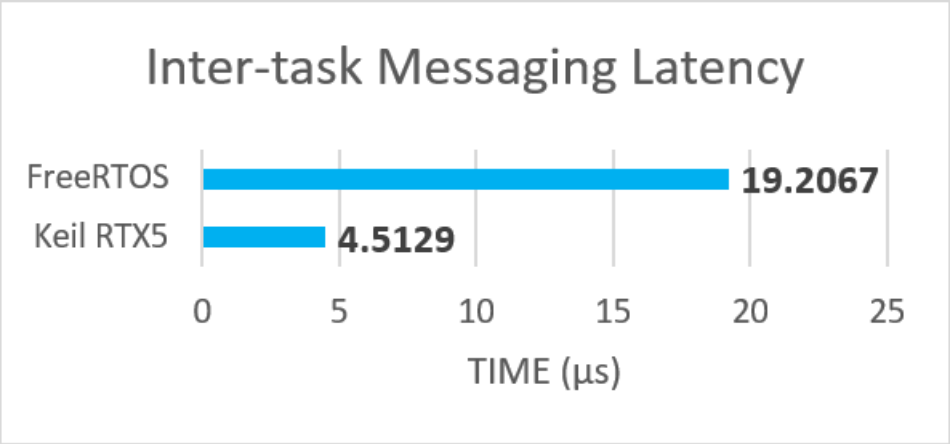

Fig 12. Measured inter-task messaging latency

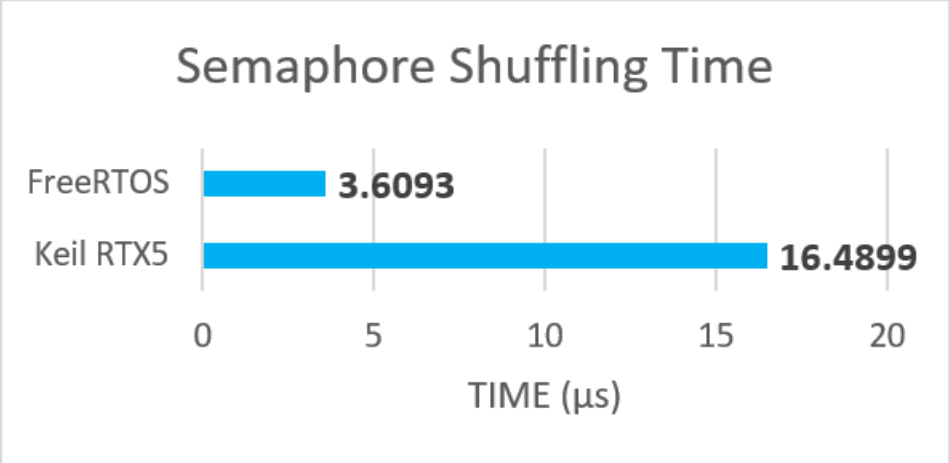

Fig 13. Measured semaphore shuffling time

Analyzing the results, we see that the slow task switching (Figure 10) is presented by RTX5. For the preemption time (Figure 11), both RTOSs have approximately the same value. Therefore, FreeRTOS has a great capability to switch between tasks in both cases; if they have the same priority or different priorities. FreeRTOS also has good performance in managing shared resources with semaphore with a semaphore shuffling time (Figure 13) of 3.6093 μs, contrarily to the large value of RTX5 which equal to 16.4899 μs. For the message passing (Figure 12), the RTX5 RTOS handles it in a short time with an inter-task messaging latency of 4.5129 μs. On the other hand, FreeRTOS has a value of 19.2067 μs.

6 Conclusion

This study presents a short overview of RTOSs, their features, and fundamentals. It gives further some description of the RTOSs selected here, which are Keil RTX5 and FreeRTOS. A general insight into the main characteristics and differences between them have been given. Before starting our implementation of the benchmarking tests, we gave the important experimental setups and a definition of the performance metrics that will be measured. The tests have been performed in an STM32F429ZI discovery board based on ARM Cortex®-M4 MCU. The task switching time, pre-emption time, semaphore shuffling time, and inter-task messaging metrics have been measured using the DWT cycle counter unit of the ARM Cortex®-M4. RTX5 is an open-source RTOS very optimized so we have expected that it will give us powerful results against FreeRTOS, but the results obtained show that FreeRTOS is fast in preemption and switching between tasks and reacts rapidly to unblock a task waiting for a semaphore. Concerning RTX5 RTOS, it has good communication and message passing time. This study contributes with a new evaluation and benchmarking results of the most used open-source RTOSs (FreeRTOS and Keil RTX5) based on four metrics, reflecting the main timing performances of a real-time operating system. The time measurement is done with the DWT cycle counter that gives an accurate time value, contrarily to the oscilloscope.

This study may represent a time indicator to help programmers on choosing a suitable RTOS to use, but we must also look into other principles, such as memory footprint, deadlock break time, interrupt latency, and licensing, as the goal for our next works, in which the MCU used in this study and other MCUs, will be evaluated for the performances when the embedded architecture changes.

References

1) EETimes. EETimes - Embedded Processors by the Numbers. 1999. Available from: https://www.eetimes.com/embedded-processors-by-the-numbers.
2) Guessi M, Nakagawa EY, Oquendo F, Maldonado JC. Architectural description of embedded systems: a systematic review. In: Proceedings of the 3rd international ACM SIGSOFT symposium on Architecting Critical Systems. 2012;p. 31–40. doi:10.1145/2304656.2304661.
3) Jheng GC, Duh DR, Lai CN. Real-time reconfigurable cache for low-power embedded systems. *International Journal of Embedded Systems*. 2010;4(3/4):235–247. Available from: https://dx.doi.org/10.1504/ijes.2010.039027.
4) Tan TK, Raghunathan A, Jha NK. Energy macromodeling of embedded operating systems. *ACM Transactions on Embedded Computing Systems*. 2005;4(1):231–254. Available from: https://dx.doi.org/10.1145/1053271.1053281.
5) Oliveira L, Mattos JCB, Brisolara L. Survey of Memory Optimization Techniques for Embedded Systems. *2013 III Brazilian Symposium on Computing Systems Engineering*. 2013;p. 65–70. doi:10.1109/SBESC.2013.35.
6) Shukla AK, Sharma R, Muhuri PK. A Review of the Scopes and Challenges of the Modern Real-Time Operating Systems. *International Journal of Embedded and Real-Time Communication Systems*. 2018;9(1):66–82. Available from: https://dx.doi.org/10.4018/ijertcs.2018010104.
7) Fei Y, Ravi S, Raghunathan A, Jha NK. Energy-optimizing source code transformations for operating system-driven embedded software. *ACM Transactions on Embedded Computing Systems*. 2007;7(1):1–26. Available from: https://dx.doi.org/10.1145/1324969.1324971.
8) Kouty SY, Mishra S. Virtual Integration in Avionics Systems-RTOS. *INCOSE International Symposium*. 2019;29(S1):180–193. Available from: https://dx.doi.org/10.1002/j.2334-5837.2019.00678.x.
9) Tan S, Anh TNB. Real-time operating system (RTOS) for small (16-bit) microcontroller. *2009 IEEE 13th International Symposium on Consumer Electronics*. 2009;p. 1007–1011. doi:10.1109/ISCE.2009.5156833.
10) Atmadja W, Liawatimena S, Lukas J, Nata EPL, Alexander I. Hydroponic system design with real time OS based on ARM Cortex-M microcontroller. *IOP Conference Series: Earth and Environmental Science*. 2017;109(1):012017. Available from: https://dx.doi.org/10.1088/1755-1315/109/1/012017.
11) Choi Y. Model checking Trampoline OS: a case study on safety analysis for automotive software. *Software Testing, Verification and Reliability*. 2014;24(1):38–60. Available from: https://doi.org/10.1002/stvr.1482.
12) ZHANG H, AOKI T, CHIBA Y. Verifying OSEK/VDX Applications: A Sequentialization-Based Model Checking Approach. *IEICE Transactions on Information and Systems*. 2015;98(10):1765–1776. Available from: https://dx.doi.org/10.1587/transinf.2015edp7043.
13) Dietrich C, Hoffmann M, Lohmann D. Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis. *ACM Transactions on Embedded Computing Systems*. 2017;16(2):1–35. Available from: https://dx.doi.org/10.1145/2950053.
14) Baldovin A, Graziano A, Mezzetti E, Vardanega T. Kernel-level time composability for avionics applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. 2013;p. 1552–1554. doi:10.1145/2480362.2480651.
15) Klingensmith N, Banerjee S. Hermes: A Real Time Hypervisor for Mobile and IoT Systems. In: Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications. 2018;p. 101–106. doi:10.1145/3177102.3177103.
16) Belleza RR, Freitas EPD. Performance study of real-time operating systems for internet of things devices. *IET Software*. 2018;12(3):176–182.
17) Yu C, Ma X, Fang F, Qian K, Yao S, Zou Y. Design of controller system for industrial robot based on RTOS Xenomai. *2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. 2017;p. 221–226. doi:10.1109/ICIEA.2017.8282846.

18) Murikipudi A, Prakash V, Vigneswaran T. Performance Analysis of Real Time Operating System with General Purpose Operating System for Mobile Robotic System. *Indian Journal of Science and Technology*. 2015;8(19):1–6. Available from: https://dx.doi.org/10.17485/ijst/2015/v8i19/77017.
19) Evanczuk S. 2019 Embedded Markets Study reflects emerging technologies, continued C/C++ dominance. 2019. Available from: https://www.embedded.com/2019-embedded-markets-study-reflects-emerging-technologies-continued-c-c-dominance.
20) FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. *FreeRTOS/indexhtml*. 2021.
21) 'FreeRTOS for RISC-V RV32 and RV64'. 2021. Available from: FreeRTOS./Using-FreeRTOS-on-RISC-V.html.
22) Martin T. ARMv8-M. In: Martin T, editor. The Designer's Guide to the Cortex-M Processor Family. 2016;p. 445–455.
23) 'RTX v5 Implementation'. 2021. Available from: https://arm-software.github.io/CMSIS_5/RTOS2/html/rtx5_impl.html.
24) Ferreira JF, He G, Qin S. Automated Verification of the FreeRTOS Scheduler in HIP/SLEEK. *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*. 2012;p. 51–58. doi:10.1109/TASE.2012.45.
25) Xing Y, Wang D, Zhao Y. Analysis and Implementation of an Embedded System Platform Based on FreeRTOS and Cortex-M3. In: Proceedings of the 2017 2nd International Conference on Communication and Information Systems. 2017;p. 350–354. doi:10.1145/3158233.3159350.
26) Shao L, Wang C, Chu C, Song Y, Hu H, Yang Y, et al. Design and implementation of real-time robot operating system based on freertos. *J Phys: Conf Ser*. 2020;1449:12115. doi:10.1088/1742-6596/1449/1/012115.
27) Peng L, Guan F, Perneel L, Timmerman M. Behaviour and performance comparison between FreeRTOS and μC/OS-III. *International Journal of Embedded Systems*. 2016;8(4):300. Available from: https://dx.doi.org/10.1504/ijes.2016.077774.
28) Renaux DPB, Pöttker F. Performance evaluation of CMSIS-RTOS: benchmarks and comparison. *International Journal of Embedded Systems*. 2016;8(5-6):452–463. doi:10.1504/IJES.2016.080389.
29) Ungurean I, Gaitan NC. Performance analysis of tasks synchronization for real time operating systems. *2018 International Conference on Development and Application Systems (DAS)*. 2018;p. 63–66. doi:10.1109/DAAS.2018.8396072.
30) Ungurean I. Timing Comparison of the Real-Time Operating Systems for Small Microcontrollers. *Symmetry*. 2020;12(4):592. Available from: https://dx.doi.org/10.3390/sym12040592.
31) Yogaraj A, Sivanthiram CS, Dhananjeyan S, Suresh S. Keil Rtos Based Embedded Web Server For Real Time Industrial Monitoring. *International Journal of Mechanical Engineering and Technology*. 2017;8(10):553–560. Available from: https://iaeme.com/MasterAdmin/Journal_uploads/IJMET/VOLUME_8_ISSUE_10/IJMET_08_10_062.pdf.
32) Wang W, Xie T, Hu X. Design of power well cover wireless monitoring system based on freeRTOS and NB-IoT technology. *IOP Conf Ser: Mater Sci Eng*. 2020;853:12038. doi:10.1088/1757-899X/853/1/012038.
33) Arm Cortex-M4 Processor Technical Reference Manual Revision r0p1. 2021. Available from: https://developer.arm.com/documentation/100166/0001/?search=5eec6e71e24a5e02d07b259a.
34) 32F429IDISCOVERY - Discovery kit with STM32F429ZI MCU. 2021. Available from: https://www.st.com/en/evaluation-tools/32f429idiscovery.html.
35) Cortex-M4. 2021. Available from: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4.