Performance of Open Source Real-Time Operating Systems: A Comparative Study of FreeRTOS and NuttX

YAĞMUR EREN ROBERT HYNASIŃSKI
yagmure | roberthy @kth.se

February 11, 2024

Abstract

The increasing popularity of Internet of Things solutions has made the Real Time Operating System (RTOS) selection process for embedded systems with real-time requirements more difficult. The selection of RTOS must take into account factors like memory footprint or performance of the scheduler and kernel services. In this research, we acquire data on the memory footprint and key parameters of both FreeRTOS and NuttX real-time operating systems related to scheduling and most important kernel services. With this information both RTOS are compared to help with picking an appropriate product for new, emerging projects in the embedded systems industry. In conclusion, the results show that FreeRTOS has lower latency values for context switch time, inter-task message transfer time, and task activation from ISR than NuttX, however, for semaphore shuffling latency, NuttX is almost 1.5 times shorter. FreeRTOS has a lower memory allocation time and it has an almost 2 times smaller memory footprint than NuttX. Taking all these parameters into account, FreeRTOS can be preferred for general purpose applications.

Contents

List of Acronyms and Abbreviations

API Application Programming Interface

DWT Data Watchpoint and Trace

GPIO General Purpose Input/Output

IDE Integrated Development Environment

IoT Internet of Things

ISR Interrupt Service Routine

ITM Instrumentation Trace Macrocell

OpenOCD Open On-Chip Debugger

RAM Random Access Memory

RTOS Real Time Operating System

SWV Serial Wire Viewer

USART Universal Synchronous/Asynchronous Receiver/Transmitter

1 Introduction

A real-time system is a system whose correctness depends on the time at which the result is produced. To ensure the predictability and determinism of such a system, real-time operating systems are used [1]. However, the increasing popularity of Internet of Things (IoT) solutions has made it more difficult to pick an appropriate Real Time Operating System (RTOS) when designing devices of this type. This is due to the fact that IoT devices are usually constrained in terms of memory and computational power to reduce the cost of a single unit. What is more, there does not exist a standardized benchmark that could be used to assess the performance of an RTOS. Due to this, finding information on metrics that are of interest to embedded systems designers is difficult.

There are also aspects that are less technical and still need to be taken into account when picking an RTOS, such as the availability of human resources familiar with a particular technology. Aside from the lack of performance tests, this is one of the reasons for picking NuttX as an RTOS to benchmark, as its Application Programming Interface (API) is compatible with POSIX API popular in Unix-based operating systems.

The intention of this study is to define what metrics are important from embedded systems designers' point of view and to measure those metrics for FreeRTOS and NuttX real-time operating systems since they are popular choices with a long history on the market and are both open-source.

1.1 Theoretical framework

[2] defines and discusses the most important concepts related to real-time operating systems. A brief literature review on RTOS evaluation methods is provided in [3]. The conclusion is that there is no standardized benchmark for research of this type and there are only a few methods that were proposed and used in the past. However, the most recurring way to estimate the performance of an RTOS is the Rhealstone benchmark which is used in multiple papers with modifications to its originally suggested form. [4] and [5] support this conclusion. The original definitions for the components of the Rhealstone benchmark are given in [6]. Based on this paper, we have redefined and picked the following metrics to construct the tests on:

- **Semaphore shuffling time** — The time interval between a release of a semaphore by one of the tasks and the activation of another task that was blocked while waiting for the semaphore. Semaphore shuffling time is used to calculate the overhead of mutual exclusion [6]. Usually, many different tasks compete for the same resources in real-time systems. To ensure that non-shareable resource is accessed by only one task at a time, semaphore based mutual can be used. Therefore, semaphore shuffling has a considerable impact on the performance of the system.

- **Inter-task message transfer latency** — When a nonzero-length data message is transferred from one task to another, there is a delay between the instance of time that the message was sent by the first task and the instance of time that the message was received by a second task. The shortest possible interval of time between those two instances is defined as message transfer latency [6].

- **Context switch time** — The average time for a system to switch between two independent and active tasks with equal priority is defined to be context switch time [6]. Task switching is one of the most prominent metrics for any RTOS, as this is one of the most occurring operations in multi-tasking systems [6].

- **Task activation from Interrupt Service Routine (ISR)** — The average time required by the system to enable a blocked task from ISR's context. It is the latency that occurs when the system is required to react to an external event.

- **Memory allocation time** — The average time required by an RTOS to allocate a block of memory of a particular size. Measured from the instance of a time when the block of memory is requested to the instance of a time when the pointer to the block of memory is made available.

- **Memory footprint** — The minimum possible size of a raw, executable binary that can be achieved through modification of configuration options available in an RTOS.

The performance tests for FreeRTOS and NuttX RTOS are implemented according to the reference manuals [7] and [8].

1.2 Research questions and hypotheses

Embedded system designers who want to build new systems with the use of RTOS for several different purposes need a lead about how to pick an appropriate one by meeting the specifications of their projects. There is a wide selection of open source RTOS available on the market. As such finding an RTOS that fulfills the requirements of a particular project in terms of latency and memory footprint requires manual testing and can introduce unnecessary delays to projects' schedules. Therefore, in this research, the answer for which of the two most popular RTOS choices in embedded systems would be more suitable to use has been investigated in terms of the factors we picked and that matter for embedded systems development.

2 Methods

This section provides an overview of the methods used in the research. Descriptions of hardware and software used are available in Sections 2.2 and 2.3 respectively. The test methodology is presented in Section 2.1.

2.1 Tests

Each test was designed to measure at most one metric. There are 6 tests in total, and each test comes in two versions, one for NuttX and one for FreeRTOS. The differences between each version of the test are kept at the absolute possible minimum to make sure that only features of an RTOS can impact the results. The test programs for each RTOS are available on two separate GitHub repositories, one for FreeRTOS[1] and one for NuttX[2].

2.1.1 Context switch time

Context switch time test is designed in such a way as to fulfill an additional role, which is to act as a check that the run-time environment is configured properly. Due to this, its structure differs from the rest of the tests. For this test, $2 \cdot 10^3$ individual samples are gathered and stored in an array. By keeping track of each sample it is possible to estimate the impact of ISR running in the background (such as system tick ISR) when the test was executed.

The test makes use of three tasks. Two tasks with the same priority level are used to force context switches, with the remaining task with lower priority used to print out the test results. One of the tasks performing context switches is responsible for time stamping the start and stop of a measurement and calculating the duration of context switches. The steps to calculate the time of two context switches with the use of two tasks are the following:

1. Enter task 1. Note down the current (start) time and immediately yield control to the second task, forcing a context switch.

2. Enter task 2 and immediately yield control back to the first task, forcing a second context switch.

3. Enter task 1. Immediately note down the current (stop) time. Calculate the difference between stop and start times and store it in a buffer and loop again.

[1] `https://gits-15.sys.kth.se/roberthy/IL2202-FreeRTOS`
[2] `https://gits-15.sys.kth.se/roberthy/IL2202-NuttX`

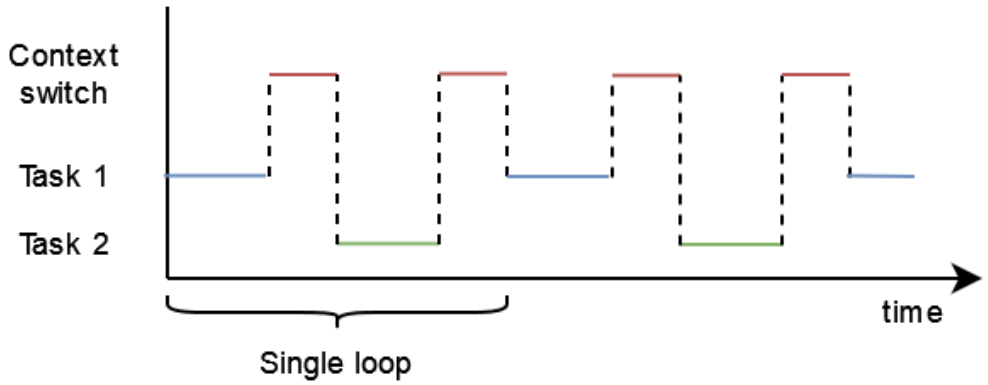

Figure 1: Task flow in context switch time test. Blue line marks when task 1 is active, and green line marks when task 2 is active. Red line marks the context switch.

These steps are presented graphically in Figure 1. It has to be noted, that this method results in two context switches being taken into account, so every measurement yields an average from two context switches.

2.1.2 Semaphore passing latency (shuffling)

Semaphore passing test is based on C implementation of the same test from one of the original authors of the Rhealstone benchmark, which is used for iRMX operating system [9]. The test makes use of three tasks. Two tasks with the same priority level, task 1 and task 2, are used to pass semaphores, with the remaining task, task 3, initially having the highest priority level. Task 3 is responsible for calculating the overhead, measuring time spent passing the semaphores and printing out the results. Overhead is specified as the difference of instance of time where task 3 is preempted by tasks 1 and 2, and instance of time where task 3 regained control.

Overhead is calculated by temporarily lowering the priority of task 3 and running tasks that pass the semaphore between themselves for specified number of loops, but with all functions related to semaphores disabled. In this case no semaphores are passed but it is possible to measure time that is spent performing actions not related to semaphore passing itself and that are required to run tests. Examples of such actions are context switches, assignments and loops. When task 1 and task 2 loop for specified number of times, they delete themselves, giving back control to the remaining task 3.

To measure semaphore passing latency, task 3 creates new instances of task 1 and task 2. Latency is measured in the same manner as overhead and between the same points of time, but with functions related to semaphore passing enabled. By taking the difference between the total time that tasks are running and overhead, it is possible to calculate the time spent on semaphore passing. Dividing this difference by number of loops that tasks executed gives average time required to shuffle a semaphore.

The semaphore shuffling procedure assumes that the semaphore is initially available and is the following:

1. Task 3: Mark down current time and give control to task 1.
2. Enter task 1. Wait for semaphore and yield control to task 2.
3. Enter task 2. Wait for semaphore. Semaphore is not available and task 2 gets suspended.
4. Enter task 1. Signal semaphore and yield control to task 2.
5. Enter task 2. Semaphore is taken by task 2. Task 2 yields control to task 2.
6. Enter task 1. Wait for semaphore. Semaphore is not available and task 1 gets suspended.

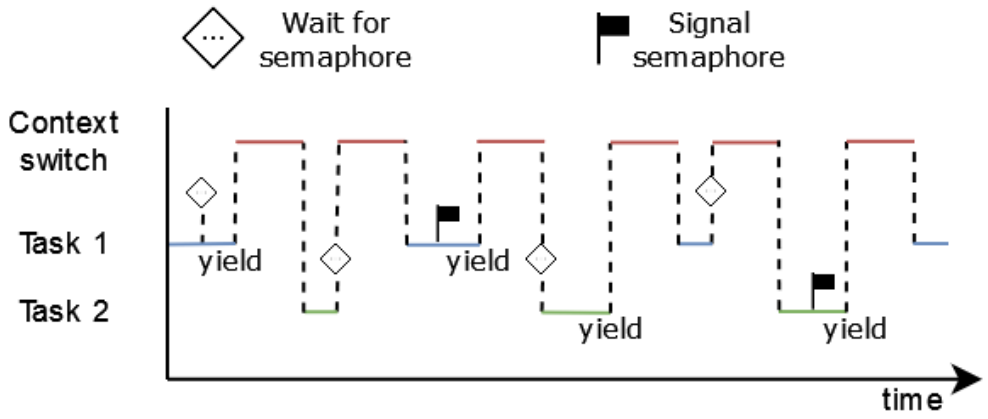

Figure 2: Task flow for a single loop in semaphore passing test. Blue line marks when task 1 is active, green line marks when task 2 is active. Red line marks context switch.

7. Enter task 2. Signal semaphore and yield control to task 1.

8. Enter task 1. Loop the procedure. If specified number of loops was already achieved, delete task 1 and task 2. Enter task 3 and mark down current time.

These steps are presented graphically in Figure 2.

2.1.3 Inter-task messaging latency

Same as the semaphore shuffling test, this test is based on C implementation for iRMX operating system [9]. The test makes use of three tasks. Task 1 fulfills the role of a producer that sends a message, while task 2 takes the role of a consumer that received a message and has lower priority than the producer. Task 3 is responsible for calculating the overhead, measuring the time spent passing the semaphores and printing out the results. Both overhead and actual latency are measured in the same way as specified in Section 2.1.2.

The inter-task messaging test transfers 8 bytes long messages. It is assumed that the queue is initially empty, can hold only one message and that receive/send operations are blocking. The procedure for the test is the following:

1. Task 3: Mark down the current time and give control to task 1.

2. Enter task 1. Send message. Loop inside the task's body and attempt to send the next message. The queue is full, so task 1 is suspended.

3. Enter task 2. Receive message. The queue is not full anymore, so task 1 preempts task 2.

4. Enter task 1. If the specified number of loops was achieved, delete both tasks and give control to task 3 to mark down the current time. Otherwise loop.

To calculate the inter-task messaging latency, the time of context switch happening after task 2 was preempted has to be subtracted from the final result.

2.1.4 Task activation from ISR latency

The test makes use of three tasks. The initialization task ensures the initial state of the board's General Purpose Input/Output (GPIO) pins. Task 1 is responsible for triggering an interrupt by setting an output GPIO pin to high. This output GPIO is connected to the input pin that interrupts the microcontroller. ISR unlocks task 2, where task 2 immediately sets second GPIO pin high.

Task activation from ISR latency is measured as the difference between a time instance when GPIO pin controlled by task 1 is set high and a time instance when the GPIO pin controlled by task 2 is set high. This

difference in time is measured with an oscilloscope. A theoretical example of output from an oscilloscope is provided in Figure 3. As the change from the low state to the high state does not happen immediately and the rising edge is not perfectly vertical, measurements are done from and to the moment the rising edge reaches 50% of pin's nominal voltage (3.3 V).

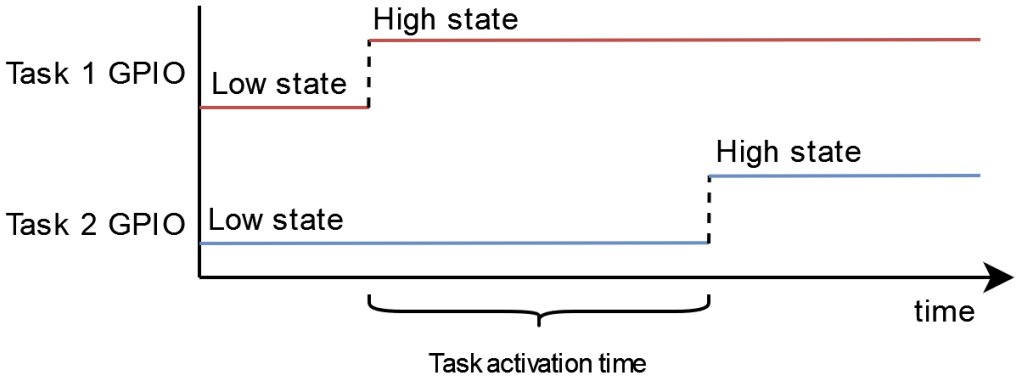

Figure 3: Theoretical oscilloscope output for a single loop in task activation test. Red line marks state of GPIO set in task 1. Blue line marks state of GPIO set in task 2.

2.1.5 Memory allocation latency

The role of the test is to compare memory allocation schemes used by FreeRTOS and NuttX in terms of introduced latency and to assess if memory allocation in FreeRTOS and NuttX can be deterministic. FreeRTOS comes with 5 allocation schemes that can be used for end-program. In this paper only `heap_4` is tested, as it is the memory allocation scheme enabled by default. NuttX provides only one memory allocation scheme.

One task is used for the test. This task repeatedly allocates and frees blocks of memory, where requested blocks have sizes of 8, 64, 512 and 1024 bytes. The latency of memory allocation is measured from the instance of a time when memory allocation is requested to the instance of a time when the pointer to the allocated memory block is returned.

2.1.6 Memory footprint

The test checks the minimum memory footprint of a binary containing an RTOS. RTOS is made to include as little functionality as possible, using configuration options provided by the vendor. Compiled test program runs indefinitely in an empty loop and does not make use of any features available in an RTOS. For the test, only the size of `.text` section is compared, as it is entirely stored in flash memory.

2.2 Hardware

STM32 Nucleo-144 development board with STM32F767ZI microcontroller is used as the platform for test execution. This particular board and microcontroller are picked due to:

- Lack of performance tests executed for ARMv7-M architecture (for both FreeRTOS and NuttX)
- Availability of Data Watchpoint and Trace (DWT) module that allows to easily track the number of cycles for which the microcontroller is running.
- Direct support for FreeRTOS and NuttX (no need to port RTOS to a new platform).
- Large pool of Random Access Memory (RAM) and flash memory (512 kB and 2 MB respectively), meaning no limit on use of any configuration option available in FreeRTOS and NuttX.
- Embedded ST-LINK2 debug probe.

2.2.1 Data Watchpoint and Trace module (DWT)

DWT is used as a cycle counter to measure the time required to run tested code. As DWT stores the current cycle count in a 32 bit register, the counter can run only for a limited amount of time before it overflows. STM32F767ZI's clock is configured to run at 216 MHz, imposing a limit of approximately 19.8 seconds for a single measurement.

2.2.2 Cache and tightly coupled memory (TCM)

STM32F767ZI has 16 kB of instruction/data cache memory. To avoid unpredictable cache misses and dependency between test results and the layout of a program in the memory, the cache is disabled for all of the tests. This has an additional benefit in that test results can be used to estimate worst case latency more reliably.

Apart from cache memory, STM32F767ZI contains 128 kB of tightly coupled RAM memory for data and 16 kB of tightly coupled RAM memory for instructions. As both of these always map to the same regions of memory and are predictable in use, they are enabled for all tests.

2.3 Software

FreeRTOS Kernel v10.2.1 [10] and Apache NuttX 11.0.0 [11] are tested. The software toolchain for FreeRTOS consists of:

- **STM32CubeIDE Version 1.10.1** — Integrated Development Environment (IDE) used for management of test code for FreeRTOS and automation of both loading and building of FreeRTOS test programs.
- **GNU ARM Embedded Toolchain (10.3-2021.10)** — provides a cross compiler and linker for building and linking of FreeRTOS test programs.
- **Windows 10 Version 10.0.19044** — operating system for running STM32CubeIDE.

The software toolchain for NuttX consists of:

- **GNU ARM Embedded Toolchain (10.3-2021.07)** — provides a cross compiler and linker for building and linking of NuttX test programs.
- **Open On-Chip Debugger (OpenOCD) Version 0.11.0** — used for loading of NuttX test programs.
- **Picocom 3.1** — serial terminal emulator, used to gather test output through USART.
- **Ubuntu 22.04.1 LTS** — operating system for running build system of NuttX.

The only element of the toolchain that has to be consistent to achieve reproducibility is GNU ARM Embedded Toolchain. Two different versions of the GNU ARM Embedded Toolchain are used, depending on the host operating system where the code is developed and loaded (Ubuntu or Windows). However, the only difference between the two versions is the provision of mitigation for the VLLDM instruction security vulnerability in processors based on ARMv8-M architecture [12]. As STM32F767ZI is based on ARMv7-M architecture and is not affected by this vulnerability [13], both versions are considered to be equal in functionality.

Other elements of the toolchain are chosen specifically to speed up the development process of test code and do not affect the test results. Ubuntu is chosen due to better support of the build system that NuttX uses but it is possible to use a toolchain based on the Windows operating system to achieve the same results.

2.3.1 Build system configuration

In spite of using different build systems, the compilation process has to be performed with the same settings for both real-time operating systems. As such:

- No link time optimizations are used.
- No debug symbols are generated.
- All test programs are built with GCC '-O3' flag.
- Macro `NDEBUG` is defined globally (for every compiled file).

FreeRTOS test code is developed using STM32CubeIDE. Complete, self-contained STM32CubeIDE projects are provided in GitHub repositories and do not require any additional configuration. NuttX test code is provided in form of a folder with NuttX applications that can be easily integrated with its build system [14]. In this case, however, the aforementioned optimizations have to be set manually in the build system. Additionally, the use of cache memories has to be explicitly disabled, as stated in section 2.2.2.

2.3.2 Gathering test output

All tests, with the exclusion of memory footprint tests, are fully automatic and do not require any manual intervention for them to complete. For these automatic tests results are printed out after the main body of the test is executed, as not to trigger any hardware interrupts that could influence the measurements.

For FreeRTOS test results are printed out with the use of Instrumentation Trace Macrocell (ITM) module available on Cortex-M7 microcontrollers and read with Serial Wire Viewer (SWV) console available in STM32CubeIDE. NuttX uses Universal Synchronous/Asynchronous Receiver/Transmitter (USART) to print out test results. Specifically, USART3 interface is available on the Nucleo-144 board and is chosen for this task, as it is directly connected to the board's embedded ST-LINK2 debug probe and can be connected without any additional configuration through a Virtual COM port. The configuration of the serial COM port can be found in Table 1.

Table 1: Configuration for the serial COM port used to retrieve test data from FreeRTOS tests.

Parameter	value
Baud rate	115 200 bit/s
Data frame size	8 bits
Parity	none
Stop bits	1 bit
Flow control	none

For memory footprint tests there are two ways to retrieve information about how much memory is used by a particular feature of an RTOS. The first one is to manually inspect the linker map file generated during the build process of a test program. The second was to use `arm-none-eabi-size`, a helper program from GNU ARM Embedded Toolchain, with a compiled executable file containing the test program. Due to the ease of use of the GNU toolchain, the latter option is chosen.

3 Results and Analysis

The latency results of inter-task message transfer, semaphore shuffling, task activation from ISR and task switching were obtained. Moreover, the minimum memory footprint and memory allocation times for both FreeRTOS and NuttX RTOS were measured. Two thousand samples are measured for the task switching data for both RTOS. For the semaphore passing and inter-task message transfer latency, total

time and overhead time are recorded as given in Table 2 and Table 3. These results are given in clock cycles. Table 4 shows the minimum memory footprint values in bytes.

The mean values for all measured latency parameters were compared in a bar graph that is given in Figure 4. The parameters were measured in clock cycles, however, all mean values were converted to microseconds in Figure 4. The y-axis shows the average/mean latency measured in microseconds and the x-axis indicates the name of the performance parameter. The mean values for each indicator of FreeRTOS can be seen in pink bars and for NuttX, in cyan bars.

Table 5 compares the memory allocation latency in cycles for both RTOS and for different block sizes (in bytes).

Table 2: Inter-task message transfer test results for 10^5 loops

RTOS	FreeRTOS (cycles)	NuttX (cycles)
Total transfer time:	$4.35 \cdot 10^8$	$6.27 \cdot 10^8$
Overhead time:	4623	57555
Average time:	4338	6264

Table 3: Semaphore shuffling test results for 10^5 loops

RTOS	FreeRTOS (cycles)	NuttX (cycles)
Total shuffling time:	$8.82 \cdot 10^8$	$12.17 \cdot 10^8$
Overhead time:	$1.61 \cdot 10^8$	$6.47 \cdot 10^8$
Average time:	7208	5707

Table 4: Memory footprint test results

RTOS	FreeRTOS (bytes)	NuttX (bytes)
Minimum:	6964	14010

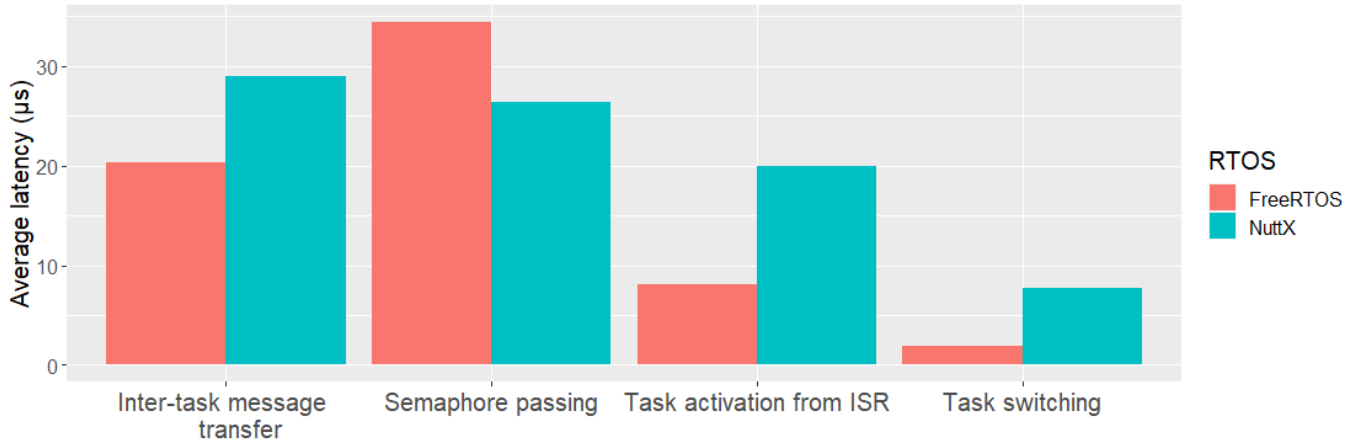

Figure 4: Comparison of tested performance indicators for FreeRTOS and NuttX

Table 5: Memory allocation time test results

Size (bytes)	FreeRTOS (cycles)			NuttX (cycles)		
-	Max	Min	Avg	Max	Min	Avg
8	1122	556	563	1277	1259	1259
64	1093	556	563	1315	1071	1151
512	1078	556	563	1172	956	982
1024	1095	556	563	1262	930	1004

For both FreeRTOS and NuttX, four different latency values have been obtained. The mean values of each parameter were compared in Figure 4. Task switching is a fundamental performance indicator and as can be seen in Figure 4, FreeRTOS has almost 4 times better task switching latency than NuttX. The latency of task activation from ISR was measured as 730.34 ns without RTOS. FreeRTOS has a latency of 8.04 µs more than 2 times better than NuttX with 19.93 µs latency.

If we look into inter-task message transfer results, with the value of 20 µs, FreeRTOS has lower latency for inter-task message transfer than NuttX. This difference between the two RTOS can be explained as a trade-off: NuttX has higher latency than FreeRTOS, however, NuttX provides flexibility with its message queue mechanism by being able to send variable-length messages (up to 22 bytes) whereas FreeRTOS has constant size queues.

4 Discussion

From the results, it can be seen that the average semaphore passing time for FreeRTOS is significantly higher than the value for NuttX. According to the FreeRTOS reference manual, semaphores are a "heavy" construct that imposes a lot of overhead, so instead task notification mechanism [15] can be used to act similarly to semaphores if better performance is required. In our test, we used binary semaphores in FreeRTOS and counting semaphores in NuttX as it is the only option available in NuttX. However, the results indicate that NuttX is still better in this aspect. The interpretation might be that for some multitasking systems NuttX can be a preferred pick if semaphore-based mutual exclusion is of importance.

The tests for memory footprint were done with the simplest program without any feature to obtain the smallest footprint possible. As given in Table 4, NuttX has almost 2 times more memory footprint than FreeRTOS. Therefore, FreeRTOS can be chosen when a microprocessor is not flexible with memory space.

The memory allocation times were measured for different sizes of memory blocks given in Table 5. The results where the same average values can be seen for the memory allocation in FreeRTOS can be caused by the structure of the test. Since a block of memory is allocated and freed immediately, memory allocation scheme used in FreeRTOS might always find the same unoccupied section of memory. When the average time values for both RTOS are compared, it can be concluded that NuttX is not deterministic and FreeRTOS can be more reliable in that sense, however still without any guarantees. As a result of the interpretation of all the data acquired for this research, FreeRTOS might be a better pick for general-purpose applications.

5 Conclusion

The purpose of the current study was to determine which RTOS performs better than the other one in metrics relevant to embedded systems design. Six different tests were implemented, where each test was only concerned about one metric. This study has shown that FreeRTOS provided lower latency for task switching, inter-task message transfers, and task activation from ISR tests. However, NuttX's semaphore passing mechanisms were more optimized. For memory allocation times, NuttX showed a less deterministic pattern compared to FreeRTOS depending on the size of the memory allocated. Moreover, FreeRTOS has a smaller memory footprint result than NuttX. Despite NuttX's worse results in those categories, it provided

good prospects as a real-time operating system, especially when considering its reliance on POSIX API. However, as real-time operating systems are complex pieces of software, it is necessary to perform a more diverse array of tests that would include different metrics that those presented in this paper. Due to this, this may constitute the object of future studies: an evaluation of different kernel services available in NuttX, such as clocks, timers, pthreads and signal interfaces.

References

[1] C. Walls, *Embedded RTOS Design: Insights and Implementation*. San Diego: Elsevier Science & Technology, 2020. ISBN 9780128228517

[2] J. J. Labrosse, *MicroC/OS-II : the real-time kernel*, 2nd ed. San Francisco, CA: CMP Books, 2002. ISBN 1-57820-103-9

[3] D. P. Renaux and F. Pöttker, "Performance evaluation of CMSIS-RTOS: benchmarks and comparison," *International journal of embedded systems*, vol. 8, no. 5/6, p. 452, 2016.

[4] T. J. Boger, "Rhealstone benchmarking of FreeRTOS and the Xilinx Zynq extensible processing platform," Master's thesis, Temple University, 2013. [Online]. Available: http://hdl.handle.net/20.500.12613/825

[5] I. Ungurean, "Timing comparison of the real-time operating systems for small microcontrollers," *Symmetry*, vol. 12, no. 4, 2020. doi: 10.3390/sym12040592. [Online]. Available: https://www.mdpi.com/2073-8994/12/4/592

[6] R. P. Kar and K. Porter, "Rhealstone: A real-time benchmarking proposal." [Online]. Available: http://archive.retro.co.za/CDROMs/DrDobbs/CD%20Release%2012/articles/1989/8902/8902a/8902a.htm

[7] *The FreeRTOS™: Reference Manual API Functions and Configuration Options*, Amazon Web Services. [Online]. Available: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf

[8] G. Nutt, *NuttX Operating System User's Manual*, Apache Software Foundation. [Online]. Available: https://cwiki.apache.org/confluence/display/NUTTX/User+Guide

[9] R. P. Kar, "Implementing the Rhealstone real-time benchmark." [Online]. Available: http://archive.retro.co.za/CDROMs/DrDobbs/CD%20Release%2012/articles/1990/9004/9004d/9004d.htm

[10] Amazon Web Services, "FreeRTOS Kernel 10.2.1 Source Code." [Online]. Available: https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/V10.2.1

[11] Apache Software Foundation, "Apache NuttX 11.0.0 Source Code." [Online]. Available: https://github.com/apache/nuttx/tree/releases/11.0

[12] "Release notes for GNU ARM Embedded Toolchain (10.3-2021.10)," https://developer.arm.com/downloads/-/gnu-rm, [Online; last accessed 25-11-2022].

[13] *VLLDM Instruction Security Vulnerability*, ARM Limited. [Online]. Available: https://developer.arm.com/Arm%20Security%20Center/VLLDM%20Instruction%20Security%20Vulnerability

[14] *Custom Apps How-to*, Apache Software Foundation. [Online]. Available: https://cwiki.apache.org/confluence/display/NUTTX/User+Guide

[15] "Rtos task notifications," Amazon Web Services. [Online]. Available: https://www.freertos.org/RTOS-task-notifications.html