IET Software

Research Article

ISSN 1751-8806
Received on 11th April 2017
Revised 13th December 2017
Accepted on 13th January 2018
E-First on 16th February 2018
doi: 10.1049/iet-sen.2017.0048
www.ietdl.org

Performance study of real-time operating systems for internet of things devices

Rafael Raymundo Belleza[1] ✉, *Edison Pignaton de Freitas*[1]
[1]*Institute of Informatics, Federal University of Rio Grande do Sul, Av. Bento Gonçalves, 9500, CP 15064, Porto Alegre CEP: 91501-970, Brazil*
✉ *E-mail: rrbelleza@inf.ufrgs.br*

Abstract: The development of constrained devices for the internet of things (IoT) presents lots of challenges to software developers who build applications on top of these devices. Many applications in this domain have severe non-functional requirements related to timing properties, which are important concerns that have to be handled. By using real-time operating systems (RTOSs), developers have greater productivity, as they provide native support for real-time properties handling. Some of the key points in the software development for IoT in these constrained devices, like task synchronisation and network communications, are already solved by this provided real-time support. However, different RTOSs offer different degrees of support to the different demanded real-time properties. Observing this aspect, this study presents a set of benchmark tests on the selected open source and proprietary RTOSs focused on the IoT. The benchmark results show that there is no clear winner, as each RTOS performs well at least on some criteria, but general conclusions can be drawn on the suitability of each of them according to their performance evaluation in the obtained results.

1 Introduction

The internet of things (IoT) consists, basically, of various types of devices, of heterogeneous hardware and software architectures, connected together and to the internet [1]. Increasing complexity of embedded systems designed for the IoT leads to the necessity of managing the various tasks performed by the system in a consistent manner, a role taken by an operating system (OS). This is an important part of the challenges faced in this context of connected cyber-physical systems in which IoT is immersed [2].

Real-time constraints are a major concern in this context, especially in at least three areas concerning IoT devices: sensing, actuation, and communication. Taking a patient remote monitoring system as an example, it is easy to observe these constraints. In such a system, sensors are attached to the patient's body to collect health parameters, such as blood pressure and oxygen level in the blood. Depending on the type of monitoring and the health problem the patient has, the acquisition of these parameters has to be timely precise, i.e. they must obey specific timing requirements in relation to the acquisition of the samples, so that they can be semantically correlated. This correct correlation provides the desired information to the health personnel monitoring this patient. However, if the data acquisition does not strictly obey the sampling period, for instance, incorrect correlations may be calculated misleading the health personnel to wrong diagnostics. This means that the sampling acquisitions should neither be faster nor slower than what is required, but precisely according to the specifications. Besides these, the use of reliable software is a must for IoT connected devices, as they can be deployed in places where repair or maintenance is difficult. Also, depending on the domain of application, such as avionics systems, industrial control, transportation, as well as for medical devices, software certification is a requirement, and using a pre-certified software stack reduces time-to-market and development costs.

RFC 7228 [3] introduces a standardisation for classifying wireless network devices with constrained system resources, based mainly on memory capacity. The proposed classification separates the devices into three categories:

i. Class 0 devices are the most constrained ones, having <10 kB of data memory (RAM) and <100 kB of code memory (ROM, e.g. Flash). These are typically sensor nodes, which most likely will not communicate directly with the internet, needing a proxy or a gateway in between, and with very basic or none at all management capabilities. In other words, these are deploy and forget nodes, in the sense that either their expected life time is so short or their functionality is so simple that no update should be expected for their entire lifetime.
ii. Class 1 devices have around 10 kB of RAM and 100 kB of ROM. These types of devices do not have full internet protocol stacks, such as hypertext transfer protocol and transfer layer security, but protocols devised for devices of this class have been created, like the constrained application protocol (CoAP) over user datagram protocol.
iii. Class 2 devices have around 50 kB of RAM and 250 kB of ROM. These devices are much less constrained than the other two classes and are capable of supporting most of the same protocols as much larger systems. However, using the same protocols as devices in lower classes makes sense, as these are typically developed with power consumption in mind, and leaves more room for application logic.

In large embedded system platforms (devices beyond class 2), it is suitable to use an OS like Linux, a BSD variant, Windows CE or even Windows 10 IoT Core. For small devices, with size, memory and energy consumption constraints (classes 1 and 2), the overhead of such OSs in memory, storage or processing power has driven to the development of specialised OSs, such as Contiki [4], the Zephyr Project [5], RIOT-OS [6] and many others, as surveyed in [7]. The authors show in their vision, what are the requirements for an OS for IoT devices: small memory footprint, support for heterogeneous hardware, network connectivity, energy efficiency, real-time capabilities, and security. Technical key design choices like general architecture and modularity, scheduling model, memory allocation strategies, network buffer management, programming models, supported programming languages, driver model and hardware abstraction layer, debugging tools, feature set, and testing are discussed. Non-technical aspects such as standards, certification, documentation, maturity, license, and provider are also discussed.

The authors in [7] then present a number of candidate OSs for IoT devices. Those that are open source are Contiki, RIOT, FreeRTOS, TinyOS, OpenWSN, eCOS, mbedOS, L4 family, uClinux, Android, Brillo and some other small RTOSs. Those which are proprietary, not open source, are the following: ThreadX, QNX, VxWorks, Wind River Rocket, PikeOS, embOS, Nucleus, Sciopta, µC/OS-II and µC/OS-III, µ-velOSity, Windows CE and Huawei LiteOS. The authors categorise these OSs into three categories: event-driven, multi-threading OSs, and pure RTOSs. Finally, a case study about the key design choices is shown for one representative of each category: Contiki, RIOT, and FreeRTOS, respectively.

Since the publication of this comprehensive survey, the Zephyr Project [5], from the Linux Foundation, has been released to developers. This project is based on the Rocket kernel, by Wind River.

Observing this landscape of OSs for IoT devices and the major importance of real-time support for IoT application, this study reports a study on real-time OSs (RTOSs) for IoT devices. Besides other important aspects related to IoT operation, such as energy consumption, the study is focused on real-time properties, as the analysis of other concerns goes beyond the intended work, besides the need for different and complementary methodology for their evaluation.

This study proposes benchmarking selected open and closed source RTOSs for IoT focusing on the performance evaluation of key synchronisation primitives the system provides for applications, such as semaphores and message queues, as well as the system responsiveness to external events, which will dictate the overall performance of devices using these OSs, thus directly impacting the time aspects of the final users' services supported by these IoT devices [8].

Besides this introduction, the paper is organised as follows: Section 2 discusses related works; Section 3 presents the study methodology, describing the benchmark criteria and the rationale behind them; Section 4 describes the experimental environment, setup and the acquired results, which are then critically discussed; finally, Section 5 concludes the paper providing directions for future work.

2 Related works

Among all of an operating system's aspects, inter process communication (IPC) performance is one of the most important criteria to be taken into account for the system to meet its timing constraints and desired performance, as seen in [9]. The authors present a set of three benchmarks to measure the performance of synchronisation primitives, most notably semaphores, on FreeRTOS.

The authors in [10] present a set of benchmarking strategies for IPC mechanisms on RTOSs. The benchmarks are application-based or based on the most frequently used features, which they call fine-grained benchmarking, themselves based on the criteria proposed in [11], which evaluate the system responsiveness to external events, inter-task synchronisation and resource sharing and inter-task data transferring.

The authors in [10] selected the following RTOSs for their study: µITRON, µTKernel, µC-OS/II, EmbOS, FreeRTOS, Salvo, TinyOS, SharcOS, XMK OS, Echidna, eCOS, Erika, Hartik, KeilOS, and PortOS. The criteria used to analyse these RTOSs are: design objective (commercial, research, hobby, etc.); author (individual, organisation or company); scheduling scheme (preemptive, cooperative, other); real-time capability and performance; memory footprint; programming language support; application programming interface (API) richness; OS-aware debugging; licensing; and documentation.

Finally, a performance and memory footprint benchmark is shown in [10] for a selected subset of these RTOSs in the Renesas M16C/62P platform. The chosen performance criteria are task switch time, get and release semaphore time, semaphore passing time, pass and receive message time, inter-task message passing time, fixed-memory acquire and release time, task activation from within an interrupt service routine. All the pseudo-codes used for these measurements are presented. Memory footprint information is obtained for each benchmark through the toolchain reports, after compilation.

In the work reported in [12], the author presents a comparison of the performance of the CMSIS-RTOS layer, developed by ARM to standardise common RTOS system calls, with two RTOSs, the X Real-Time Kernel [13], and a FOSS RTOS kernel, which was not specified by the author. The benchmarks are based on the same methodology as [10].

In [14], the authors present SenSpire OS, an OS for wireless sensor networks, and CSpire, the programming language, which is used to create applications using this OS. They present the various aspects which defined the development of this OS: predictability, flexibility, and efficiency. Predictability is achieved through using the priority ceiling protocol for synchronisation primitives and what they call as two-phase interrupt servicing, which is basically performing the sensitive and critical parts inside the interrupt handlers, and deferring the rest to a task with lower priority. Flexibility is approached by having a hybrid system model, combining event-driven and multi-threaded programming. As for efficiency, SenSpire OS supports dynamically loadable system modules, enabling energy-efficient software. Stack sharing is also used to reduce the memory footprint. A series of benchmarks evaluating interrupt latency, scheduler overhead, and overall system performance is also presented.

Different from [9], which focuses on the performance of semaphores on FreeRTOS, [10], which does a comprehensive study of objective (like performance and language support) and subjective characteristics (like license and documentation) of RTOSs focused on industrial applications, [12], which evaluates the CMSIS-RTOS middleware, the present paper combines selected benchmarks from them and [14], and presents a performance evaluation of key performance components of RTOSs, focusing on RTOSs for IOT connected devices.

3 Study methodology

Among all of the RTOSs present in the survey in [7], three were chosen: FreeRTOS [15], RIOT [6], µC/OS-III [16]. Besides these three, the Zephyr and µC/OS-II [17] RTOSs will also be benchmarked following the same criteria. They were chosen based on their open source license, popularity [18] and out-of-the-box support for the test platform used in this study. Another popular open source RTOSs, like Apache Mynewt, had incipient support for the used platform at the time of developing and writing of this work, and others, like mbedOS, had a setup procedure that was considered to be too cumbersome. Additionally, the mbed web-based integrated development environment (IDE) was not sufficient for the needs of the study here reported. The benchmarks which will be run against the selected RTOSs are what the authors in [10, 11] call fine-grained benchmarks, leading to an evaluation of essential features of an RTOS.

These benchmarks are the task-switching time, the time for getting and releasing a semaphore, the time for passing a semaphore, the time to pass and receive a message, the time to pass a message between tasks, the time to acquire and release a fixed-size memory region, and finally, the time to activate a task from within an interrupt service routine.

In addition to these seven benchmarks, another one, based on the criteria presented in [9], which evaluates the task activation jitter, is also executed.

3.1 Benchmark criteria

3.1.1 Task switching time: There are two tasks in this benchmark. Task A has a higher priority than task B. As soon as task A is awoken, it goes to sleep, and a context switch to task B occur, which in turn awakens task A. The time for the switch between tasks A and B is measured (CS 1 in Fig. 1*a*).

3.1.2 Getting and releasing a semaphore time: In this benchmark, a single task gives and takes a semaphore repeatedly. The semaphore is initialised as taken.

The times for taking and releasing this semaphore are measured separately (grey area in Fig. 1*b*).

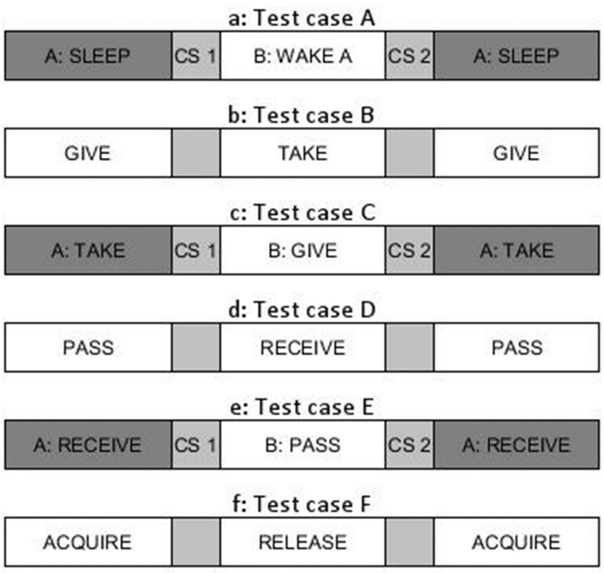

Fig. 1 *Test cases A–F*
(a) Test case A, ***(b)*** Test case B, ***(c)*** Test case C, ***(d)*** Test case D, ***(e)*** Test case E, ***(f)*** Test case F

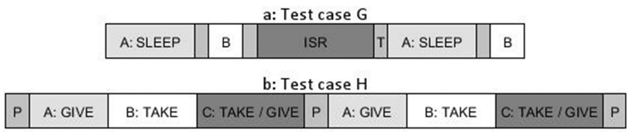

Fig. 2 *Test cases G and H*
(a) Test case G, ***(b)*** Test case H

3.1.3 Semaphore passing time: There are two tasks in this benchmark. Task A has a higher priority than task B. The semaphore is initialised as taken.

As soon as task A is awoken, it tries to take the semaphore. A context switch to task B occurs, which gives the semaphore, causing a context switch to task A. The time for the switch between tasks A and B is measured (CS 1 in Fig. 1*c*).

3.1.4 Pass and receive a message time: In this benchmark, a single task passes and receives messages repeatedly. The message queue is initialised as empty. The times for passing and receiving a message are measured (grey areas in Fig. 1*d*).

3.1.5 Inter-task message passing time: There are two tasks in this benchmark. Task A has a higher priority than task B. The message queue is initialised as empty.

As soon as task A is awoken, it asks for a message, and a context switch to task B occurs, which in turn passes a message to task A, causing a context switch to task A. The time for the switch between tasks A and B is measured (CS 1 in Fig. 1*e*).

3.1.6 Fixed-size memory acquire and release time: In this benchmark, a single task acquires a fixed-size region of memory and releases it. The times to acquire and to release this region are measured separately (grey areas in Fig. 1*f*).

3.1.7 Task activation from within an interrupt service routine (ISR): In this benchmark, there are two tasks. Task A has higher priority than task B. As soon as task A is awoken, it goes to sleep, causing task B to be executed indefinitely. When an interrupt occurs, its handler awakes task A.

The time taken for the task A to be resumed from within the interrupt handler is measured (T in Fig. 2*a*).

3.1.8 Task activation jitter induced by priority inversion: This benchmark has a similar setup to test case A, but with the addition of a third task, task C, which has lower priority and no interaction at all with the other two tasks.

Task C makes use of a second semaphore, initialised as taken, and does an infinite sequence of give and take operations on it.

Ideally, due to the absence of interaction between task C and the other tasks, the timing of the other tasks should not be altered in any way. However, as shown in [9], FreeRTOS has an in-kernel critical region for protecting concurrent accesses to its own data structures.

This can introduce jitter in the task activation timings if the tick interrupt occurs while the kernel is inside one of its own critical regions. The times between P in Fig. 2*b* are measured.

3.2 Rationale behind the criteria

Benchmark A is extremely important, as it shows the system performance on the most basic context switching scenario, without involving pre-emption, as the tasks simply enter in the suspend state and later are resumed.

Benchmarks B and D show the performance of the provided IPC directives without incurring in context switches showing the underlying quality of these primitives. While benchmarks C and E show the performance of the IPC directives, this time involving two different tasks. As shown later, the results are, in almost all cases, the results of benchmark A, added together with the results of benchmarks B and D, with a little overhead.

Benchmarks C and E show how a generic device will respond to internal events, like posting a message to a send queue, locking and unlocking a critical section in the networking stack, passing a received message to a treatment task, etc.

While not common in traditional, single-purpose real time embedded systems, dynamic memory allocation can be an important factor for IoT systems, enabling them to perform different tasks. Nevertheless, a consistent timing and behaviour are still expected for this kind of system, and this is what benchmark F shows.

Perhaps the most important benchmarks, G and H show the behaviour of the system under different conditions. In benchmark G, an external event, serviced by its interrupt handler, wakes up a task to perform work based on this event, showing how well the system responds to events. Benchmark H, in turn, shows how well the system respects its timing constraints under heavy load.

Considering, for instance, a people counter application, these two benchmarks can be used to determine how well this application would react to surges in the number of people circulating in the monitored areas. If the devices take too long to service interrupts, it is possible that a miscount will occur. Also, if the devices do not respect the timing constraints under this unexpected surge, system functions, like communications, could be negatively impacted.

4 Experiments and results

4.1 Hardware configuration

The hardware chosen to conduct this study was the NXP/Freescale FRDM-K64F board [19], which contains a Kinetis K64F, an ARM Cortex-M4F based microcontroller, capable of running at 120 MHz, with 1 MB of flash memory and 256 kB of RAM, with a great set of peripherals: USB, Ethernet, SDHC card reader, connectors for readily available Nordic nRF24L01+ wireless and JY-MCU Bluetooth modules and connectors for Arduino compatible shields. This platform was chosen due to the fact that it has out-of-the-box support for all of the selected RTOSs.

4.2 OSs setup and configuration

4.2.1 FreeRTOS: The 9.0.0 version of FreeRTOS was chosen for this work. Despite this version introducing static allocation of threads, semaphores and message queues, the old style API, with dynamic memory allocation was used. The main configuration parameters are given in Table 1.

4.2.2 RIOT: The default configuration of RIOT for the FRDM-K64F was modified to disable one of the timer modules, which used the channels 2 and 3 of the programmable interrupt timer

(PIT). Despite this, sema was the only additional module used, for enabling semaphores. The version used for the performed tests was at the 4fbe9b1d11 Git commit, from 2016-11-19.

4.2.3 μC/OS-II and μC/OS-III: The code for the benchmarks was based on the default project created by the Kinetis Design Studio IDE when used in conjunction with the Kinetis SDK. The modified configuration parameters for μC/OS-II and μC/OS-III are given in Table 2. Flag OS_CFG_TMR_TASK_RATE_HZ is only present in μC/OS-III because, in *μ*C/OS-II, the software timer rate is the same as the tick rate, while in μC/OS-III, it can be an integer fraction of the tick rate. The versions used were 2.92.11 for μC/OS-II and 3.05.01 for μC/OS-III. Both versions were supplied by Freescale, on the Kinetis SDK builder webpage.

4.2.4 Zephyr: As in RIOT, the default configuration for the FRDM-K64F board was used with no modifications. Like the other systems, the tick rate was configured to be 1000 Hz. The used version was at the 79f020be54 Git commit from 12-11-2016.

4.3 Results

All timings were measured using the channels 2 and 3 of the PIT, which has 32 bits per channel and is free-running and downwards counting timers running at the same clock as the main processor bus, at 60 MHz, half of the core clock. For each benchmark, 128 batches of 1024 samples were captured, resulting in a total of 131,072 measurements.

Table 1 FreeRTOS configuration parameters

Configuration flag	Value
configUSE_PREEMPTION	1
configUSE_IDLE_HOOK	0
configUSE_TICK_HOOK	0
configTICK_RATE_HZ	1000
memory heap management strategy	heap_4.c

Table 2 μC/OS-II and μC/OS-III configuration parameters

μC/OS-II	μC/OS-III	Value
OS_TASK_TMR_PRIO	OS_CFG_TMR_TASK_PRIO	5
OS_TICKS_PER_SEC	OG_CFG_TICK_RATE_HZ	1000
—	OS_CFG_TMR_TASK_RATE_HZ	1000

4.3.1 Task switching: Observing the benchmark statistics in Tables 3 and 4 and Fig. 3, RIOT has the best results in this benchmark because of its clean implementation and being a tickless RTOS. Looking only at the means, μC/OS-II and FreeRTOS, both of which also have a simple implementation, follow. μC/OS-III and Zephyr are next, with mixed results for both.

While Zephyr clearly had a quicker response to task suspension, resuming is almost 50% slower than μC/OS-III, and almost 100% slower than RIOT.

The high switching times of Zephyr can be explained by its rich API, which has many layers of abstraction. While the API call used in this benchmark does not present this to the user, the underlying code makes use of these flexible APIs. Besides this, the execution of two interrupt service handlers, the supervisor call (SVC) and the system-level service handler (PendSV), is necessary to make the task switch, while for the other RTOSs tested, the context switches are performed by the PendSV handler. RIOT branches to the SVC handler from within the PendSV, but an interrupt is not generated, eliminating the need for the tail-chaining operation in the ARM Cortex-M4 core [20].

Even though the quickest response is a desirable characteristic, in an RTOS, determinism is much more important. Looking at the standard deviation, RIOT again is clearly the winner, with the lowest variation. μC/OS-III comes next. FreeRTOS and Zephyr follow with mixed results in both measurements. μC/OS-II is the RTOS with the worse results for these benchmarks.

Table 5 presents the system API calls used in this benchmark.

4.3.2 Getting and releasing a semaphore: Like the previous benchmark, the results here (Fig. 4*a*) show a difference of 100% between two or more systems. Here, though, there is not a clearly faster OS. FreeRTOS and Zephyr have the highest times for releasing a semaphore, followed by μC/OS-III.

FreeRTOS semaphores are based on message queues, which pass NULL messages, explaining the slower performance. μC/OS-III uses the underlying OS_Post function, which is common to various other functions, and must handle a multitude of situations, which uses a more complex logic, and in turn, a slower performance for posting a semaphore. On the other hand, μC/OS-

Table 3 Task suspension statistics

	FreeRTOS	RIOT	μC/OS-II	μC/OS-III	Zephyr
mean	6.255775	5.049999	5.586434	8.441663	7.419267
Std	0.1203	0.000322	1.527383	0.041686	0.095662
min	5.85	4.933333	5.05	7.983333	7.1
1%	6.233333	5.05	5.466667	8.4	7.416667
25%	6.233333	5.05	5.466667	8.4	7.416667
50%	6.233333	5.05	5.466667	8.4	7.416667
75%	6.266667	5.05	5.483333	8.483333	7.416667
99%	6.266667	5.05	5.483333	8.483333	7.416667
max	7.983333	5.05	27.966667	8.483333	10.966667

Table 4 Task resume statistics

	FreeRTOS	RIOT	μC/OS-II	μC/OS-III	Zephyr
mean	5.720775	5.338889	5.710462	8.475	11.820378
std	0.099118	0.007857	1.573594	0.008334	0.12047
min	5.7	5.333333	5.583333	8.466667	11.783333
1%	5.7	5.333333	5.583333	8.466667	11.783333
25%	5.7	5.333333	5.583333	8.466667	11.816667
50%	5.716667	5.333333	5.583333	8.475	11.816667
75%	5.716667	5.35	5.6	8.483333	11.816667
99%	5.733333	5.35	5.6	8.483333	11.816667
max	7.383333	5.366667	28.1	8.5	15.366667

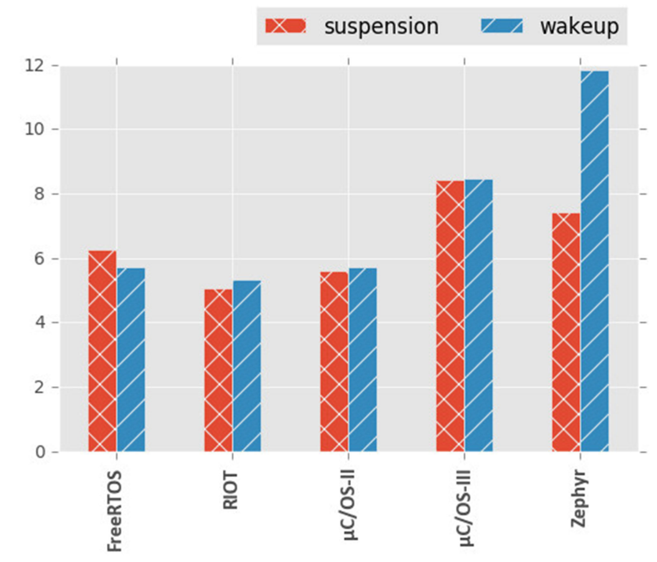

Fig. 3 *Task suspension and wake-up times, in µs*

Table 5 System APIs used in benchmark A

FreeRTOS	vTaskSuspend	vTaskResume
RIOT	thread_sleep	thread_wakeup
µC/OS-II	OSTaskSuspend	OSTaskResume
µC/OS-III	OSTaskSuspend	OSTaskResume
Zephyr	k_thread_suspend	k_thread_resume

III has the fastest performance for taking a semaphore. The common function OS_Post is also used, but because of its lean implementation, it has the best performance of all.

4.3.3 Semaphore passing: This benchmark results (Fig. 4*b*) show the performance when using semaphores to synchronise two tasks. For RIOT, µC/OS-II and µC/OS-III and Zephyr, the results are, basically, the results of benchmarks A and B added together (suspension with take, wakeup with give) in between 1 and 2 µs of the overhead, as shown in Fig. 4*c*.

The biggest difference occurs with taking a semaphore in FreeRTOS, with almost 6 µs of overhead. The reasons for this result are not clear. The system API calls used in benchmarks B and C are shown in Table 6.

4.3.4 Passing and receiving a semaphore: As benchmark B, this benchmark shows the performance solely of the message queue implementations, not taking in consideration task switching caused by message passing, though, in this benchmark, the results (Fig. 5*a*) are more uniform than on benchmark B.

FreeRTOS and Zephyr, again, have almost the same results for posting and receiving while *µ*C/OS-III shows the same difference as in benchmark B. Again, µC/OS-II has the lowest times, followed by RIOT.

4.3.5 Inter-task message passing: Like in benchmark C, the results of this benchmark (Fig. 5*b*) show the performance when using message queues to do inter-task communication. Again, like in benchmark C, the results are almost the same as benchmarks A and D added together (suspension with receive, wakeup with post) with between 1 and 2 µs of the overhead, as shown in Fig. 5*c*.

Again, the biggest difference occurs when receiving a message in FreeRTOS, with almost 5 µs, with no clear reason to why this occurs. The system API calls used in benchmarks D and E are shown in Table 7.

4.3.6 Fixed-size memory acquire and release: Fig. 6 shows the results of this benchmark. µC/OS-II, µC/OS-III and RIOT have almost the same times. While µC/OS-II and µC/OS-III use a predefined memory pool, with fixed block size, RIOT has no means of releasing a memory region, when using its base package. It is possible to use a two-level segregated fit allocator, which has deterministic time, but it is necessary to make another download and it has a memory overhead for each memory pool.

The heap_4 strategy used in FreeRTOS tries to avoid memory fragmentation, using a predefined memory pool also, but using a first fit algorithm, and coalescing adjacent free memory blocks into single larger blocks. This implementation, though, is not deterministic.

Zephyr also uses a predefined memory pool, and like FreeRTOS, is not deterministic. The main difference is that the Zephyr allocation scheme uses a recursive function to search for free blocks of memory.

Table 8 presents the system API calls used in this benchmark.

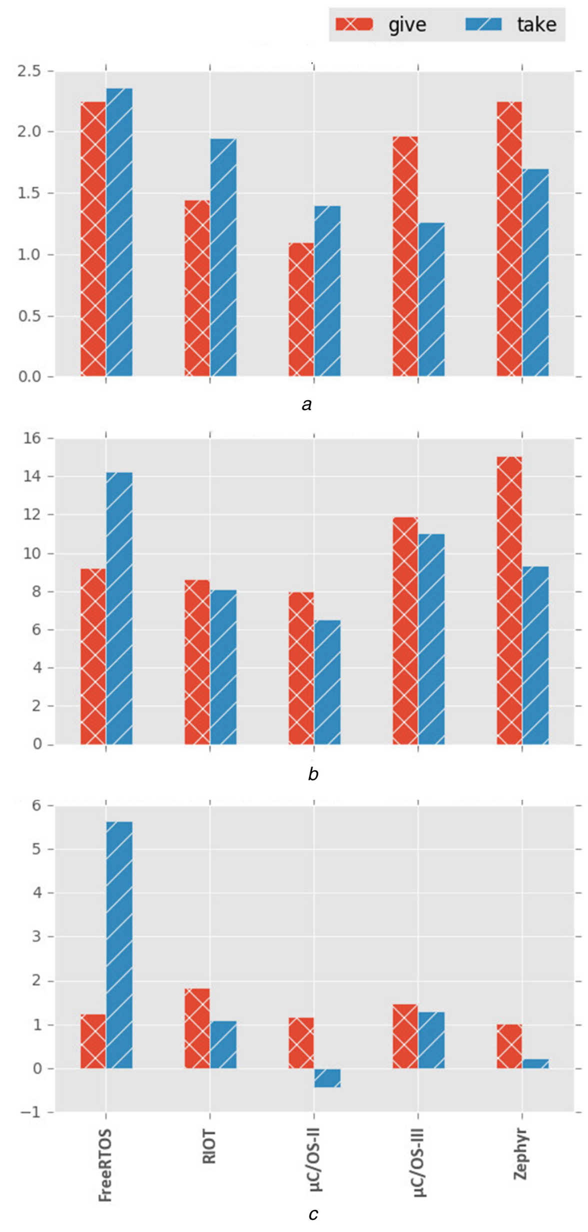

Fig. 4 *Semaphore give and take times, in µs*
(a) Benchmark B, ***(b)*** Benchmark C, ***(c)*** Difference of benchmarks A and B added together, with regard to benchmark C

Table 6 System APIs used in benchmarks B and C

FreeRTOS	xSemaphoreTake	xSemaphoreGive
RIOT	sema_wait	sema_post
µC/OS-II	OSSemPend	OSSemPost
µC/OS-III	OSSemPend	OSSemPost
Zephyr	k_sem_take	k_sem_give

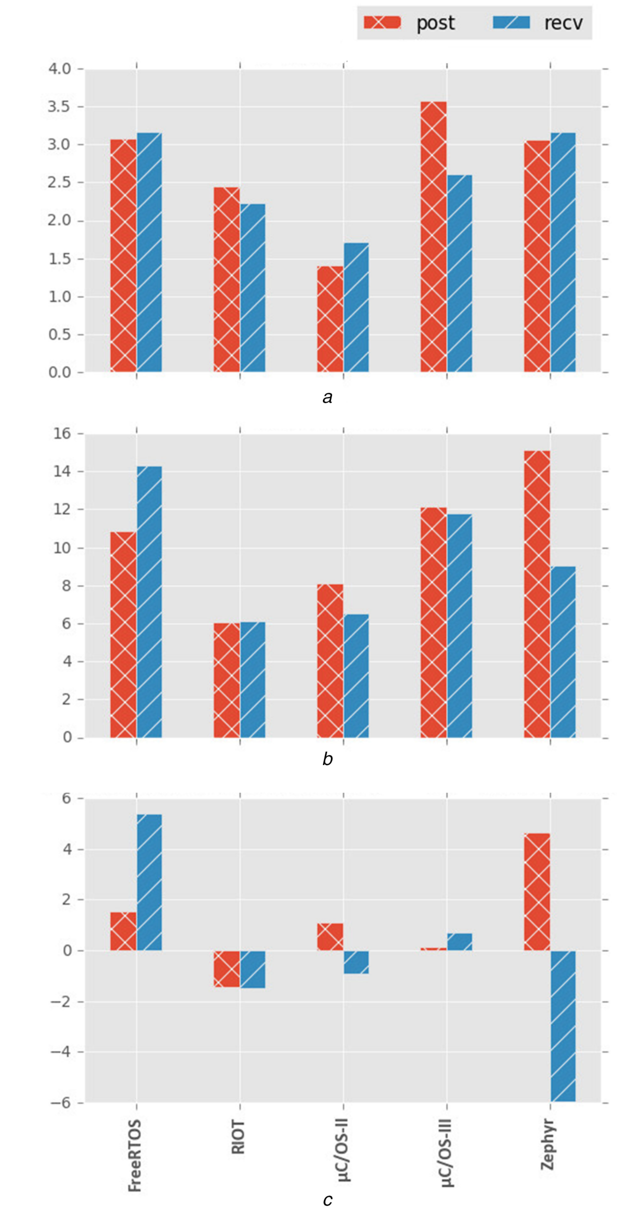

Fig. 5 *Message post and receive times, in μs*
(a) Benchmark D, ***(b)*** Benchmark E, ***(c)*** Difference of benchmarks A and B added together, with regard to benchmark E

Table 7 System APIs used in benchmarks D and E

FreeRTOS	xQueueReceive	xQueueSend
RIOT	msg_receive	msg_send
μC/OS-II	OSQPend	OSQPost
μC/OS-III	OSQPend	OSQPost
Zephyr	k_msgq_get	k_msgq_put

Table 8 System APIs used in benchmark F

FreeRTOS	pvPortMalloc	vPortFree
RIOT	Malloc	—
μC/OS-II	OSMemGet	OSMemPut
μC/OS-III	OSMemGet	OSMemPut
Zephyr	k_malloc	k_free

4.3.7 Task activation from interrupt service routine: This benchmark was initially supposed to run on FreeRTOS, RIOT, Zephyr and μC/OS-III, though, due to limitations in the API from μC/OS-III, calling OSTaskResume directly from an interrupt service routine is not possible. Due to this, μC/OS-II was added to the list of systems to be benchmarked, to keep the comparison fair between the families of the chosen OSs.

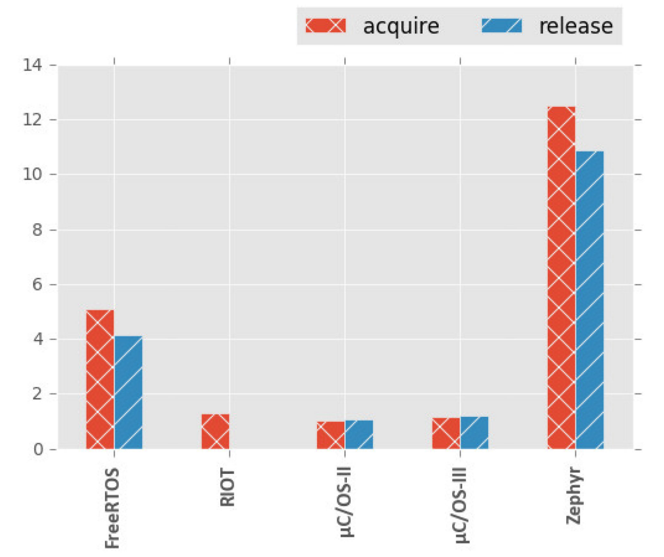

Fig. 6 *Memory allocation and release times, in μs*

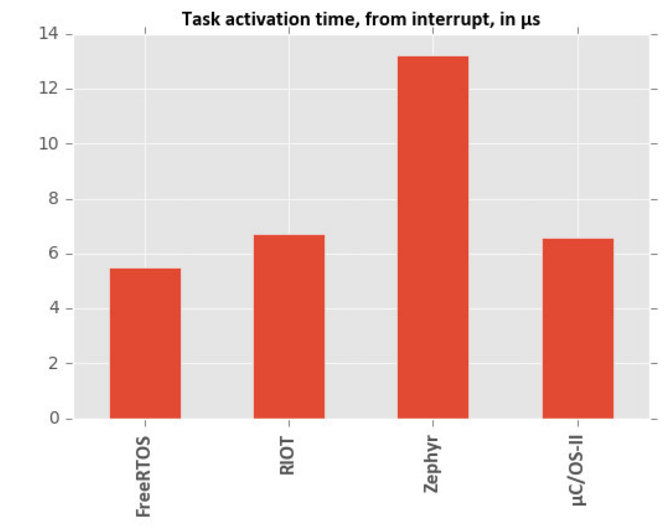

Fig. 7 *Task activation from ISR time, in μs*

Fig. 7 shows that FreeRTOS has the smallest time because the kernel does not keep track of interrupts, ISR safe APIs are provided instead, like xTaskResumeFromISR. RIOT and μC/OS-II have around the same delay, about 1 μs more than FreeRTOS, while Zephyr has the highest of all, at almost 14 μs, which is the same behaviour found in benchmark A.

4.3.8 Task activation jitter by priority inversion: Fig. 8 shows the jitter in the activation times of task A. It is possible to observe that μC/OS-II and μC/OS-III have the jitter centred in 0 μs and have the lowest variance, followed by RIOT, which, even though has the jitter centred around 4 and 5 μs, the variance is still low. The jitter in FreeRTOS and Zephyr have the greatest variance, with the whiskers of the boxplots in almost 1.5 μs.

The reasons for this behaviour in FreeRTOS are already discussed in [9]. Analysing the code for the k_sem_take and k_sem_give functions in Zephyr, we can see that interrupts are disabled right after entering these functions, so the same reason for FreeRTOS applies to Zephyr.

RIOT semaphores are based on message queues, as in FreeRTOS, but the critical sessions where interrupts are disabled are smaller and used only where strictly necessary. In addition, the periodic wakeup in RIOT is not based on the SysTick, but another timer, so the 4.5 μs delay can be subtracted from the wakeup period, to have it even closer to 0.

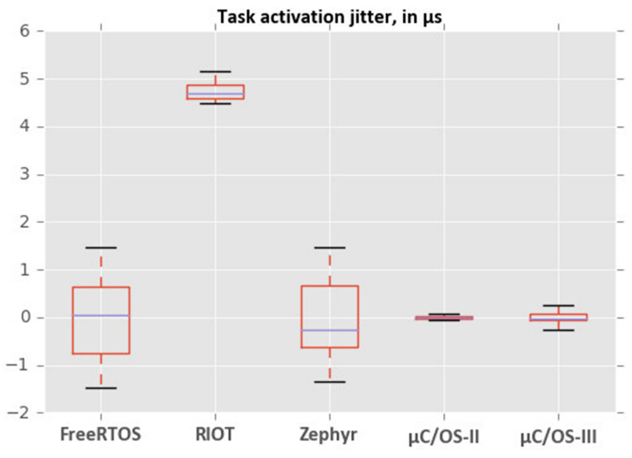

Fig. 8 *Task activation jitter, in µs*

µC/OS-III has a small variance for the same reason as RIOT, the critical sessions with interrupts disabled in OSSemPend and OSSemPost functions are small. µC/OS-II has the smallest variance because OSSemPend and OSSemPost functions have very small critical sessions.

5 Conclusion

Reliable software is an important requirement for IOT devices. The set of benchmarks presented in this study measure the performance of features and primitives of RTOSs focused on low-end IoT devices.

For high-integrity and safety-critical systems, there is no doubt that µC/OS-II and µC/OS-II should be chosen, as these systems have commercial support available, and are pre-certified in standards concerning avionics systems (DO-178B), industrial control (IEC61508) and medical devices (ISO62304).

For commercial applications, open-source OSs, such as RIOT, FreeRTOS, and Zephyr can be an advantage because concerning IoT devices, privacy and security are fundamental and open-source software can address easily these two characteristics due to the collaborative way the development takes place.

FreeRTOS, while not as consistent as RIOT, also had good results. Commercial support is available through OPENRTOS [21], a distribution which removes all GPL restrictions from the base system, and SAFERTOS, a safety-critical, pre-certified kernel for several applications, such as transportation and rail, nuclear and automotive.

Despite being categorised under the multi-threaded OSs and not the RTOS group in [7], in the benchmarks presented in this study, RIOT had more consistent results than the other systems. Zephyr, while showing response times much greater than the other systems, also had low variations in its behaviour. Both coming pre-packed with network stacks for multiple protocols, such as 6LoWPAN [22], RPL [23], OpenWSN [24] and CoAP [25], and supporting a huge number of development boards, in RIOT ranging from the Texas Instruments EZ430-Chronos smart watch [26], to the FRDM-K64F, and in Zephyr ranging from the Arduino Due [27] to the Intel Galileo [28]. These facts represent a huge advantage over the other systems, in which the developers would need to integrate all the drivers and networking subsystems.

For future works, a performance study of the network communication and routing protocols provided by the studied RTOSs, such as 6LoWPAN, RPL and CoAP could give more insights into which OS is best suited for different kinds of applications. Another important direction for future work is a precise study on energy consumption associated with the networked operation of the IoT devices, as well as the analysis on security aspects, which is a must regarding IoT applications.

6 References

[1] Al-Fuqaha, A., Guizani, M., Mohammadi, M., *et al.*: 'Internet of things: a survey on enabling technologies, protocols, and applications', *IEEE Commun. Surv. Tutor.*, 2015, **17**, (4), pp. 2347–2376

[2] Mosterman, P.J., Zander, J.: 'Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems', *Softw. Syst. Model.*, 2016, **15**, (1), pp. 5–16. available at https://doi.org/10.1007/s10270-015-0469-x

[3] I. E. T. Force: 'Terminology for constrained-node networks', available at http://www.ietf.org/rfc/rfc7228.txt, accessed 1 December 2016

[4] Dunkels, A., Gronvall, B., Voigt, T.: 'Contiki – a lightweight and flexible operating system for tiny networked sensors'. 29th Annual IEEE Int. Conf. on Local Computer Networks, 2004, pp. 455–462

[5] L. Foundation: 'Zephyr project', available at https://www.zephyrproject.org/, accessed 1 October 2016

[6] Baccelli, E., Hahm, O., Gunes, M., *et al.*: 'RIOT OS: towards an OS for the internet of things'. 2013 IEEE Conf. on Computer Communications Workshops (INFOCOM WKSHPS), 2013, pp. 79–80

[7] Hahm, O., Baccelli, E., Petersen, H., *et al.*: 'Operating systems for low-end devices in the internet of things: a survey', *IEEE Internet Things J.*, 2016, **3**, (5), pp. 720–734

[8] Mohamad, R., Aziz, M.W., Jawawi, D.N.A., *et al.*: 'Service identification guideline for developing distributed embedded real-time systems', *IET Softw.*, 2012, **6**, (1), pp. 74–82

[9] Bertolotti, I.C., Kashani, G.G.Z.: 'On the performance of open-source RTOS synchronization primitives'. 2015 IEEE 1st Int. Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), 2015, pp. 398–402

[10] Tan, S.L., Nguyen, B.A.T.: 'Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers', *IEEE Micro*, 2009, **PP**, (99), pp. 1–1

[11] Garcia-Martinez, A., Conde, J.F., Vina, A.: 'A comprehensive approach in performance evaluation for modern real-time operating systems'. Proc. EUROMICRO 96. 22nd Euromicro Conf. on Beyond 2000: Hardware and Software Design Strategies, 1996, pp. 61–68

[12] Renaux, D.P.B.: 'Comparative performance evaluation of CMSIS-RTOS'. 2014 Brazilian Symp. on Computing Systems Engineering, 2014, pp. 126–131

[13] eSysTech: 'X real time kernel', available at http://www.freertos.org/, accessed 9 December 2017

[14] Dong, W., Chen, C., Liu, X., *et al.*: 'Senspire OS: a predictable, flexible, and efficient operating system for wireless sensor networks', *IEEE Trans. Comput.*, 2011, **60**, (12), pp. 1788–1801

[15] 'FreeRTOS – market leading RTOS (real time operating system) with internet of things extensions', available at http://www.freertos.org/, accessed 2 December 2016

[16] L. Foundation: 'µC/OS-III documentation home', available at https://doc.micrium.com/ucosiiidoc/, accessed 2 December 2016

[17] L. Foundation: 'µC/OS-II documentation home', available at https://doc.micrium.com/ucosiidoc/, accessed 2 December 2016

[18] AspenCore: '2017 embedded markets study', available at https://m.eet.com/media/1246048/2017-embedded-market-study.pdf, accessed 9 December 2017

[19] 'FRDM-K64F|freedom development platform|kinetis mcus|nxp', available at http://www.nxp.com/products/software-and-tools/hardware-development-tools/freedom-development-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F, accessed 2 December 2016

[20] Yiu, J.: 'A beginner's guide on interrupt latency – and interrupt latency of the arm cortex-m processors', available at https://community.arm.com/processors/b/blog/posts/beginner-guide-on-interrupt-latency-and-interrupt-latency-of-the-arm-cortex-m-processors, accessed 9 December 2017

[21] W. high integrity systems: 'OPENRTOS, part of embedded FreeRTOS – OPENRTOS – SAFERTOS family', available at https://www.highintegritysystems.com/openrtos/, accessed 2 December 2016

[22] I.E.T. Force: 'Transmission of ipv6 packets over IEEE 802.15.4 networks', available at http://www.ietf.org/rfc/rfc4944.txt, accessed 12 December 2016

[23] I.E.T. Force: 'RPL: Ipv6 routing protocol for low-power and lossy networks', available at http://www.ietf.org/rfc/rfc6550.txt, accessed 12 December 2016

[24] 'OpenWSN', available at https://openwsn.atlassian.net/wiki/, accessed 12 December 2016

[25] I.E.T. Force: 'The constrained application protocol (COAP)', available at http://www.ietf.org/rfc/rfc7252.txt, accessed 12 December 2016

[26] T. Instruments: 'Chronos: wireless development tool in a watch', available at http://www.ti.com/tool/ez430-chronos, accessed 12 December 2016

[27] 'Arduino Due', available at https://www.arduino.cc/en/Main/ArduinoBoardDue, accessed 12 December 2016

[28] 'Intel Galileo Board', available at https://software.intel.com/pt-br/iot/hardware/galileo, accessed 12 December 2016