Analysis of FreeRTOS Overheads on Periodic Tasks

Bruno Dourado Miranda[1], **Rômulo Silva de Oliveira**[1], **Andreu Carminati**[2]

[1] Automation and Systems Department (DAS)
Federal University of Santa Catarina (UFSC) – Florianópolis, SC – Brazil.

[2]Education, Research and Extension Department
Federal Institute of Santa Catarina (IFSC) – Gaspar, SC – Brazil.

b.miranda@posgrad.ufsc.br, romulo.deoliveira@ufsc.br

andreu.carminati@ifsc.edu.br

Abstract. *Real-Time Operating Systems (RTOS) have their own modules that need to be executed to manage system resources and such modules add overhead to task response times. FreeRTOS is used for experimental purposes since its is a widely used open-source RTOS. This work presents the investigation of two important sources of overhead: Function Tick, a FreeRTOS time marker, and the Context Switch between tasks. In this paper we also describe a model for reducing Tick analysis pessimism due to its temporal variation. Experiments measuring the execution time of Tick and Context Switch on ARM-Cortex M4 microprocessor were made to present the Best-Case Execution Time and the Worst-Case Execution time within a periodic task scenario. Measurements are used to validate the analytic models.*

1. Introduction

Real-Time Operating Systems (RTOS) [Hambarde et al. 2014] are used by the industry to implement applications which have soft timing requirements (soft real-time). A Real-Time application is divided in tasks, that are code snippets that have temporal constraints. Each task of a real-time application usually has a execution time (C) and a period (P). Ideally, a RTOS should have no temporal impact on the execution of real-time applications. However, the algorithms and control structures of a RTOS have influence on temporal aspects of a task.

In a more realistic scenario, the RTOS modules should be deterministic, so their temporal influence on applications would be visible and predictable, however that is not our reality. The temporal influences of a RTOS are called overheads, which are executions of internal kernel routines for managing tasks running on a microprocessor. In that way, the objective of the designers of a RTOS is to minimize the overheads imposed by the kernel on the real-time application tasks. Most RTOS are developed for single-core microprocessors, such as FreeRTOS. Therefore, kernel overheads can cause considerable delay in tasks considered time-critical when they execute on these architectures.

The Worst-Case Execution Time (WCET) of a task is the time that task takes from start to finish its own execution in its worst scenario. The Response Time (R) is the time that task takes from arrival to conclusion considering the interference, release jitters and blocking it receives from other system tasks and the kernel itself. For instance, it's showed in [Audsley et al. 1993] algebraic equations to estimate the Worst-Case Response Time

(WCRT) of a task in a certain application. When a RTOS is used, the overhead cause influences in the response time of each task, so it is important to know the temporal behavior of the overheads and how they can influence application response time.

FreeRTOS is an open-source kernel project distributed with a MIT license. In 2017, the Amazon technology company extended the kernel project by programming connectivity modules, which are optional for project adoption for cloud solutions, focused on Amazon Web Services. FreeRTOS is a flexible and adaptable RTOS in many system models, such as: Cyclic executive with or without interruptions, aperiodic or periodic tasks, creation of tasks at run time and fixed or dynamic priorities.

The goal of this paper is to present a temporal analysis of FreeRTOS overheads when it runs on the ARM *Cortex-M*4 architecture. The paper focus on two important sources of overhead: The Tick function, which is a time passage marker programmed on the *Systick* interrupt and also the context switch overhead, implemented on the *PendSV* interrupt, which switches the execution contexts between running application tasks. We provide a model that allows developers to predict the temporal impact of those two sources on the application.

The structure of this paper is as follows: Section 2 describes the related work. In Section 3 we explain some characteristics of FreeRTOS tasks and kernel scheduler logic. Section 4 presents the experimental environment. Section 5 presents the Tick function, its logic and temporal behavior, and its delay sources. Also, in this section we propose the tick temporal partition to reduce temporal pessimism. Section 6 presents the context switch function, its logic and temporal behavior, as well as its delay sources. Section 7 presents the experiments realized on our experimental environment. Section 8 finally presents the conclusions and comments on future work.

2. Related Work

The performance of real-time operating systems is the target of several studies by the academy. Areas such as robotics, aerospace and several industries such as manufacturing and energy are concerned with RTOS performance, evidenced by works such as the [Pinto et al. 2020], in which a RTOS benchmark is created for the robotics area using FreeRTOS as a testing platform. The work in [Parikh et al. 2013] proposes performance parameters to compare open source RTOS as well as benchmark techniques. They specifically consider three real-time operating systems: RT-Linux, FreeRTOS and eCos. The authors conclusion is that the vital parameter of a RTOS is its scheduler, given that the logic of the context switch between tasks is performed by this module.

The work described in [Kumar Reddy et al. 2014] seeks to compare the latency of task scheduling between two real-time operating systems, FreeRTOS and *ChibiOS* [Sirio 2014]. In [Guan et al. 2016], a temporal analysis of overhead is made regarding context switch in the different versions of the FreeRTOS kernel up to version eight. In [Oliveira and Lima 2020] the FreeRTOS scheduler is modified and evaluated regarding the context switch between tasks.

The kernel overhead analysis has been very well explored in the literature like [Cofer and Rangarajan 2002] and [Radojkovic et al. 2008]. Some works are not concerned with the architectural aspects of the operating system that may cause latency. They simply want to find out which operating system presents the lowest latency when a certain set of tasks is executed (like [Ungurean and Gaitan 2018] and [Ungurean 2020].) The work presented in this paper differs from those by not only measuring and comparing execution times but providing a model that explains the logical cause of temporal variability in context switches and in the accounting of time passage in FreeRTOS. We aim to characterize the worst-case scenarios for the generation of these overheads, as well as to propose a model that describes their behavior.

3. FreeRTOS Tasks and Scheduling

In this section we present the properties of real-time tasks in FreeRTOS, as well as the kernel scheduling algorithm.

The classic task concept of real-time systems can be implemented on an ARM running FreeRTOS in two distinct ways: User Tasks (i.e. Tasks and Software Timers) and Interrupts. The ARM microprocessor also has two distinct types of execution. The first one is the Handler Mode where FreeRTOS Tick and context switches are performed. The handler mode is a privileged execution mode. Also, interrupts programmed by the user run only in handler mode, interacting with User Tasks through specific functions. The second type of execution is the Thread Mode, where User Tasks can execute. Tasks, regardless they being periodic or not, are created with function `xTaskCreate`. Each Task on FreeRTOS has a priority and a memory stack. Tasks created by the application designer can implement critical sections using semaphores.

Periodic tasks are created using function `vTaskDelayUntil`. The parameter `xTimeIncrement` of this function is the period of the task in Ticks, provided by macro `pdMS_TO_TICKS`, which calculates the Ticks using the desired period.

Software Timers are executed by task `TmrSvc` which is created automatically by the kernel when a timer is first created. Timers can only be created by function `xTimerCreate`. All timers in FreeRTOS have a priority defined by macro `configTIMER_TASK _PRIORITY`, being that priority assigned to task `TmrSvc`. The timer created can be periodic or not, depending on `uxAutoReload` being set to 1 (periodic) or 0. The function associated to a timer is a C void function and cannot implement critical sections through the use of mutex or semaphore.

The FreeRTOS scheduler algorithm is fully preemptive, if the macro `configUSE_PREEMPTION` is set to 1. When a high-priority task arrives, the scheduler will preempt the low-priority task using the *PendSV Cortex-M* interrupt. In periodic scenarios the scheduler will use Tick, implemented on the *SysTick Cortex-M* interrupt, to release the high-priority task. If two tasks have the same priority then the round-robin algorithm will be used by the scheduler, to share the microprocessor resources, provided that the macro `configUSE_TIME_SLICING` is set to 1. The default round-robin quantum for scheduling is two milliseconds.

4. Experimental Environment

Two experiments were developed based on the execution of tasks on the STMicroelectronics NUCLEO-F446RE Micro-Controller Unit (MCU) [STMicrolectronics 2019], which is a platform commonly used for embedded systems. This MCU has an ARM *Cortex-M*4 single core microprocessor, which has floating point support, a 3-stage pipeline and 240 interrupt entries [Arm 2020]. The test frequency was fixed at 84 MHz, and the maximum execution frequency supported by the microprocessor is 180 MHz.

Amazon FreeRTOS Real Time Operating System version 10.2.1 [Barry 2019] was adopted to perform the tests running on the STMicrolectronics MCU. The 8-channel Saleae Logic Analyzer was used at a rate of eight million samples per second in order to measure execution times.

5. Tick

FreeRTOS has an interrupt called Tick which accounts the time passage and assists the scheduling of other tasks. This is the only task with periodic behavior found as part of the RTOS itself. The time value is kept as an integer, named in the kernel as `xTickCount`, that is incremented when a Tick interruption occurs in the ARM *Cortex-M*4 architecture. When the task to be released has a higher priority than the task in execution, a context switch is triggered through register `portNVIC_INT_CTRL_REG`. The handler used for Tick implementation is executed with disabled interrupts. Tick is entirely a critical section.

FreeRTOS uses interrupt handlers and allows specific modules built for the context of interruptions to be implemented in it. It assigns to Tick the lowest priority among the interrupt priorities of the microprocessor, that is, the task that implements Tick has a low priority in relation to the other interrupts created by the user. The kernel uses a macro in the *port.c* file, which is `portNVIC_SYSTICK_PRI`. The Tick activation period is controlled by the kernel through two macros present in the configuration file: `portTICK_PERIOD_MS` and `configTICK_RATE_HZ`. The first is configurable and the second is a fractional constant defined as $\frac{1000}{configTICK\_RATE\_HZ}$. The default minimum period for Tick in the architecture used in this paper is *1* millisecond.

5.1. Delay Sources

It is possible to point out two causes of Tick activation delay. The first cause is the possibility of an application task or a kernel function to disable interrupts to control the access to some critical section. If at the time to activate Tick a system call is made with interrupts disabled, a delaying timing effect will be imposed on Tick, thus causing release jitter. The second cause is user-programmed interrupt handlers. The priority of Tick is the highest among the tasks programmed by the designer in the kernel and is the lowest in relation to all scheduled hardware interrupts. At the time of Tick's activation, it is possible that an interrupt handler programmed by the user is running, this means that Tick must wait for the end of the execution of that handler before starting its execution.

The microprocessor also allows interrupt nesting, so in an even more uncontrolled time scenario, it is possible that Tick has to wait for the execution of all interrupt handlers nested in the handler that prevented its release, because they have higher priority than Tick. In cases where an interrupt of the *SysTick* type is released and begins to be serviced, it is possible that before the machine instructions that disable the interrupts are processed, a higher priority interrupt handler will be serviced by the microprocessor.

5.2. Execution Time

The temporal variation in the execution of Tick is directly influenced by the number of tasks managed by the kernel. Tick is implemented by function `xTaskIncrementTick` in `port.c` kernel file. The temporal variation results from the manipulations of kernel lists, in which Tick removes tasks from the blocked state when the timeout of a task expired and that task needs to be released.

For each task released in its respective beginning of period, its removal of `pxDelayedTaskList` and its subsequent insertion in the respective list of `pxReadyTasksLists[`σ`]` ($\sigma \equiv$ priority of task) through function `prvAddTaskToReadyList` causes a new iteration within function `xTaskIncrementTick`.

The occurrence of this phenomenon in a pessimistic analysis may cause the Tick execution time to be estimated considering the critical instant as described in [Liu and Layland 1973] where all periodic tasks are released at the same time, causing *n* iterations in `xTaskIncrementTick`. Variable n denotes the number of tasks managed by the kernel, not including tasks implemented as interrupt handlers.

5.3. Temporal Algebraic Model

In order to avoid a pessimistic analysis, it is possible to partition Tick's variant behavior into pseudo-tasks. Each γ^i_{Tick} pseudo-task has a computation time corresponding to the time to remove a task γ from `pxDelayedTaskList` and its subsequent insertion in `pxReadyTasksLists[`σ`]`, where σ denotes its priority. This logically causes n application tasks to result in n pseudo-tasks of removing and inserting tasks in Tick lists. As the temporal behavior of γ^i_{Tick} is periodic it is possible to deduce that P(γ^i_{Tick}) is equal to $P(\gamma)$. Regarding the priority of γ^i_{Tick}, it is possible to assume that it has a lower priority than Tick and higher priority than other tasks under management of the RTOS, given that it executes in a disabled interrupts context.

It's possible to build a temporal algebraic model of Tick. Because Tick executes with disabled interrupts, the preemption by interrupt handlers with a higher priority than Tick is not possible. Also, no blocking point is observed because Tick does not share any resources with another task or another interrupt handler. The behavior of Tick is shown in Figure 1. In the timeline presented in Figure 1, the impact of the architecture latency time, which in the *ARM Cortex-M4* is *12* cycles of clock, is represented by *L*, corresponding to the time interval when the register bank saves the previous task execution context and reloads the execution context of Tick.

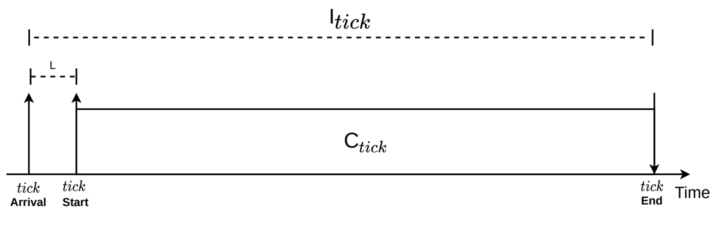

Figure 1. Tick Behavior

The temporal influence of Tick on a task programmed by the user is denoted by I_{Tick} and described by Equation 1.

$$I_{Tick} = L + C_{Tick} \tag{1}$$

6. Context Switch

Context switch operations are normally considered a black box by developers, because they are not concerned with architectural aspects of the operating systems. They only observe the logical effect of sharing the processor among tasks. Application designers more attentive to the application temporal behavior can measure the execution time of the context switch itself, although they are not usually concerned with its source.

Ideally, the context switch in any RTOS would have a execution time equal to zero, thus not interfering with tasks or adding sources of response time variation. However, what really can be done is to minimize the temporal effects of such scheduling operation, accelerating or optimizing context switch activities, given that it is impossible to do it instantly. FreeRTOS has a context switch structure that is programmed based on interrupt *PendSV* of the ARM *Cortex-M*4 architecture. Function `vTaskSwitchContext` (in `task.c` kernel file) is mostly implemented in the C language mixed with instructions specific to the *Cortex-M* architecture in assembly.

6.1. Delay Sources

Considering a fully preemptive scheduling scenario, release jitter can occur for the context switch interrupt handler when Tick disables interrupts or another interrupt handler with higher priority than *PendSV* is executing. The function `xTaskIncrementTick` has the purpose of increasing the variable that counts the passage of time and changing task states from blocked or suspended to ready, as well as moving the pointer to the task that will own the processor from list `pxDelayedTaskList` to list `pxReadyTasksLists[`σ`]`.

When a context switch is required (a higher priority task is ready) the kernel activates interrupt *PendSV*. If interrupts are disabled, the context switch execution will not be immediate, it will take a time for the operation to actually happen. Context switch delays due to interrupts being disabled by other tasks depends on the system design.

6.2. Execution Time

The greatest source of time variation caused by the context switch algorithm is through the function `vTaskSwitchContext` that selects the highest priority task, using the macro `taskSELECT_HIGHEST_PRIORITY_TASK`, implemented in `task.c` kernel file. This temporal variation is caused by searching $pxReadyTasksLists[\sigma]$ for a list that contains a task ready, from a variable that stores the highest priority of a ready task at that moment, `uxTopReadyPriority`.

The greatest execution time for this function happens when a task with the highest priority of the system performs a context switch to the idle task. The lowest execution time is the other way around, when the idle task leaves the processor and the task that will be executed is the highest-priority task of the system. Obviously, the execution of the algorithm is also subject to the temporal variability caused by the processor's acceleration mechanisms. However, a more sensitive temporal variation is perceived in the times of the task selection function, which are dependent on the priority value of the highest priority ready task at the moment. The C_{cs} time diagram of a context switch is shown in Figure 2, where function `vTaskSwitchContext` selects the highest priority ready task.

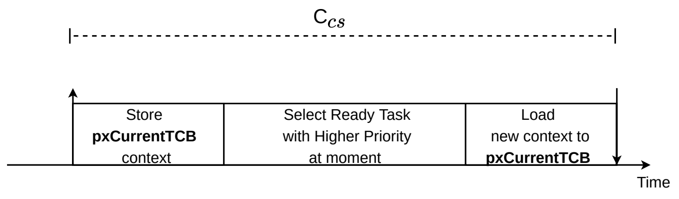

Figure 2. Context Switch Logic

6.3. Temporal Algebraic Model

Figure 3 shows the influence I_{cs} of a context switch in the system.

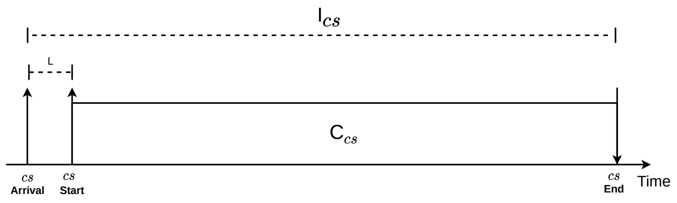

Figure 3. Context Switch Behavior

We use L to denote the time taken to respond to the interrupt in which the current execution context is stored. The C_{cs} variable represents the execution time of `vTaskSwitchContext` function. Equation 2 presents the time influence of a context switch I_{cs} on a task programmed by the user.

$$I_{cs} = L + C_{cs} \tag{2}$$

7. Experiments

In this section, temporal experiments involving Tick are presented, as well as temporal experiments involving context switch between tasks on the FreeRTOS kernel.

7.1. Tick

Table 1 shows ten periodic tasks under FreeRTOS management for testing on the ARM *Cortex-M*4 microprocessor.

In order to reduce pessimism in this model, Tick can be partitioned into ten new pseudo-tasks with the same priority than Tick, which is a lower priority than the priority of interrupt handlers. Since each arrival of a task influences the execution time of Tick, so, we create pseudo-task γ^i_{Tick} for each task γ_i. To define the execution time of the pseudo-tasks, it is necessary to measure the loop time in `xTaskIncrementTick`. The

Task	Period	Priority
A	05 ms	2
B	10 ms	2
C	15 ms	2
D	20 ms	2
E	30 ms	2
F	40 ms	2
G	23 ms	2
H	12 ms	2
I	45 ms	2
J	50 ms	2

Table 1. Tasks executing on FreeRTOS

insertion time and removal time of each task have a variation caused by acceleration mechanisms. Therefore, the execution time of γ^{i}_{Tick} is assumed to be the worst measured iteration time when inserting any task to `pxReadyTasksLists` and removing any task from `pxDelayedTaskList`. Figure 4 shows Tick execution time over 250 seconds of execution on ARM *Cortex-M*4.

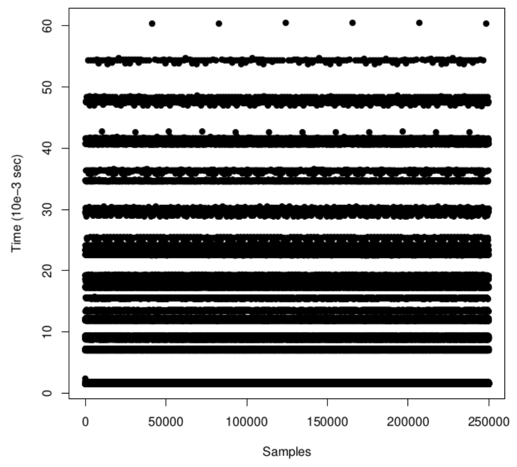

Figure 4. Tick Execution Time samples

The graph shows six execution peaks caused by the simultaneous arrival of all tasks in the system, the so-called critical instant by [Liu and Layland 1973]. The observed WCET of Tick for ten tasks is approximately 60.75 μs and the observed Best-Case Execution Time is approximately 1.5 μs. The Best-Case Execution Time is observed when there is no manipulation of lists in the kernel, which in the optimistic scenario. If we assume the pseudo-Tick as the most frequent value on measurements (i.e. mode), the pseudo-Tick execution is 1.5 μs because no task is unblocked or arrives. When some task arrives, the worst measured value to introduce this task into its respective ready list is 3.7μs, so, in this scenario if each pseudo-task has 3.7 μs, the total overhead on Tick is 37 μs.

However, the Worst-Case measured Tick time is 60.75μs, what means that the estimated overhead differs 22.07 μs from the Tick Worst-Case measured time. It happens because statements before the unblocking task function are not measured in our experimental environment, once that it's outside of task unblock function. This code block is not time variant in the algorithm, thus, it is acceptable to divide this remaining time between pseudo-tasks. Considering this, if we sum the time to move the task from the delayed list to the ready list, each pseudo-task takes 5.9 μs to execute. Table 2 presents the modeling of pseudo-tasks resulting from the Tick division, as well as their respective periods.

	Period	**C**
γ_{Tick}	01 ms	1.5 μs
γ_{Tick}^{A}	05 ms	(3.7 + 2.207) μs
γ_{Tick}^{B}	10 ms	(3.7 + 2.207) μs
γ_{Tick}^{C}	15 ms	(3.7 + 2.207) μs
γ_{Tick}^{D}	20 ms	(3.7 + 2.207) μs
γ_{Tick}^{E}	30 ms	(3.7 + 2.207) μs
γ_{Tick}^{F}	40 ms	(3.7 + 2.207) μs
γ_{Tick}^{G}	23 ms	(3.7 + 2.207) μs
γ_{Tick}^{H}	12 ms	(3.7 + 2.207) μs
γ_{Tick}^{I}	45 ms	(3.7 + 2.207) μs
γ_{Tick}^{J}	50 ms	(3.7 + 2.207) μs

Table 2. Pseudo-Tasks created from Tick

7.2. Context Switch

For the Tick time analysis it is irrelevant to define task priorities. As we showed with the Tick experiment, task arrivals are fundamental to Tick execution time variability. However, the execution time variation is not observed for context switch if we use the same test of the previous experiment given that system task priorities are the same. So, unlike Tick, the context switch time variation is linked to task priorities. Considering a scenario in which four application tasks implemented on FreeRTOS compete for processing resources, measurements of context switch between tasks is necessary. Table 3 shows the tasks, their periods and their priorities.

Task	**Period**	**Priority**
A	05 ms	6 (High)
B	06 ms	5 (Middle-High)
C	07 ms	4 (Middle)
D	08 ms	3 (Middle-Low)
IDLE	-	0 (Low)

Table 3. Tasks to test on Context Switch experiment

Assume CS(α,β) is the time to context switch from task α to task β. Task A has five distinct context switch times to other tasks. The context switches *CS(B,A), CS(C,A), CS(D,A)* and *CS(IDLE,A)* have similar timing. That behaviour of the context switch is defined by `uxTopReadyPriority` which is the highest ready priority in the system, causing only one loop iteration. For those switches the only time variation is caused by hardware acceleration mechanisms. The other context switch times are distinct: *CS(A,B), CS(A,C), CS(A,D)* and *CS(A,IDLE)*.

Task B presents six different execution times for its context switch. Scenarios *CS(C,B), CS(D,B), CS(IDLE,B)* have a similar timing. All other time context switches involving this task are timely distinct: *CS(A,B), CS(B,A), CS(B,C), CS(B,D)* and *CS(B,IDLE)*.

Task C presents six different execution times for the context switch. Switches *CS(D,C)* and *CS(IDLE,C)*, from lower-priority tasks to task C, are similar. Switches *CS(C,A)* and *CS(C,B)*, from C to higher-priority tasks, are also similar. Each other possible context switch presents a different execution time: *CS(A,C), CS(B,C), CS(C,D),* and *CS(C,IDLE)*. Similar considerations can be made for tasks D and IDLE.

Table 4 shows the number of loops and the worst-case measured time of context switches among tasks through function `taskSELECT_HIGHEST_PRIORITY_TASK` in Table 3 after *1000* seconds. The number of loops performed in selecting the highest priority task in a context switch is a result of subtracting the priority of the highest priority task *X* that leaves the processing resources for task *Y* with low priority that will receive the processing resources. When the reverse occurs and a low priority task has its processing resources allocated to a high priority task, the number of loops is *1*. In this situation *uxTopReadyPriority* receives the priority value of the task *X*, which is the highest priority task at the time of context switching.

Loop Iterations	**Context Switch**	**Time**
1	*CS(A,B), CS(B,A), CS(B,C), CS(C,A), CS(C,B), CS(C,D), CS(D,A), CS(D,B), CS(D,C), CS(IDLE,A), CS(IDLE,B), CS(IDLE,C)* e *CS(IDLE,D).*	$\approx 1.5us$
2	$CS(A,C)$ e $CS(B,D)$.	$\approx 2.875us$
3	$CS(A,D)$ e $CS(D,IDLE)$.	$\approx 3.255us$
4	$CS(B,IDLE)$ e $CS(C,IDLE)$.	$\approx 3.875us$
5	$CS(A,IDLE)$.	$\approx 4.375us$

Table 4. Context Switch times in four tasks

8. Conclusions and Future Work

In this paper, two internal FreeRTOS modules that add overheads to tasks were analysed: Tick, a time-passage counter, and the context switch operation. The overhead caused by Tick has its execution time variation linked to the priority of the periodic tasks present in the system. The WCET of Tick happens with the simultaneous arrival of all tasks. Context switch operations have a temporal variation linked to the priority of tasks, with their worst case execution time being the context switching from the highest priority task to the lowest priority task of the system.

We did several experiments in order to validate the considerations on Tick and context switch overheads, which were described in Sections 5 and 6, respectively. Experiments comprised several periodic FreeRTOS tasks running on an ARM Cortex-M4 microprocessor. The experiments corroborated our overhead considerations. Tick measurements presented in Table 2 illustrate our theoretical model of Tick execution, which is based on pseudo-tasks and reduces the pessimism of the analysis.

In this paper we analysed kernel overhead and its impact on the response time of application tasks. A natural evolution of this research thread is to model the impact of application tasks on the response time of other application tasks. This is particularly challenging since FreeRTOS allows the designer of a real-time application to implement tasks using several different abstractions, such as Timers, Interrupts and User Tasks. Ideally we would like to have equations for the response time of a task capturing all these aspects of FreeRTOS.

9. Acknowledgement

This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

Arm (2020). *Cortex-M4.*

Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. (1993). *Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal*, 8(1):284–292.

Barry, R. (2019). *Mastering the FreeRTOS real time kernel.* https://freertos.org.

Cofer, D. and Rangarajan, M. (2002). Formal verification of overhead accounting in an avionics rtos. USA. IEEE Computer Society.

Guan, F., Peng, L., Perneel, L., and Timmerman, M. (2016). Open source freertos as a case study in real-time operating system evolution. *Journal of Systems and Software*, 118:19–35.

Hambarde, P., Varma, R., and Jha, S. (2014). The survey of real time operating system: Rtos. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pages 34–39.

Kumar Reddy, B. M., Satyanarayana, G. S. R., and Seetaramanjaneyulu, B. (2014). Scheduling latency comparison of two open-source rtoss on cortex-m3. In *2014 International Conference on Embedded Systems (ICES)*, pages 59–62.

Liu, C. and Layland, J. W. (1973). *Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment. Journal of the Association for Computing Machinery*, 20(1):46–61.

Oliveira, G. and Lima, G. (2020). Evaluation of scheduling algorithms for embedded freertos-based systems. In *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8.

Parikh, H., Shah, R., Shah, U., and Deshmukh, S. (2013). Performance parameters of rtoss; comparison of open source rtoss and benchmarking techniques. In *2013 International Conference on Advances in Technology and Engineering (ICATE)*, pages 1–6.

Pinto, M., Wehrmeister, M. A., and Oliveira, A. (2020). Real-time performance evaluation for robotics. *Journal of Intelligent and Robotic Systems*, 101.

Radojkovic, P., Cakarevic, V., Verdú, J., Pajuelo, A., Gioiosa, R., Cazorla, F. J., Nemirovsky, M., and Valero, M. (2008). Measuring operating system overhead on cmt processors. In *2008 20th International Symposium on Computer Architecture and High Performance Computing*, pages 133–140.

Sirio, G. D. (2014). *ChibiOS/RT The Ultimate Guide*.

STMicrolectronics (2019). *STMicrolectronics NUCLEO-F446RE*.

Ungurean, I. (2020). Timing comparison of the real-time operating systems for small microcontrollers. *Symmetry*, 12(4).

Ungurean, I. and Gaitan, N. C. (2018). Performance analysis of tasks synchronization for real time operating systems. In *2018 International Conference on Development and Application Systems (DAS)*, pages 63–66.