Implimentation of ERD-light Scheduling Algorithm on FreeRTOS without Kernel Modification

Takaharu Suzuki[1,a)] Kiyofumi Tanaka[1,b)]

Abstract: In scheduling algorithms based on the Rate Monotonic (**RM**) method widely used in development of real-time systems, tasks with shorter periods have higher priorities. In contrast, ones with longer periods are likely to suffer from increased response times and jitters due to their lower priorities. We proposed the Execution Right Delegation (**ERD**) method for uniprocessor systems based on RM where a high-priority server for a privileged (or important) task is introduced to shorten response times of the task. In our previous study, we evaluated algorithms from python-based scheduling simulator. In this paper, we implement simplified ERD, named ERD-light, method on FreeRTOS without kernel modification. While many proposed scheduling algorithms are evaluated by modifying the kernel scheduler, ERD-light was realized by combining APIs without modifying the scheduler. In the evaluation, it is confirmed that behavior of ERD-light is same as python-based simulation while response times of a privileged task are reduced compared with RM method. We also confirmed that footprint penalty is less than 256 byte, and overhead penalty is less than 1% of total CPU usage.

Keywords: Real-time scheduling, RTOS, FreeRTOS

1. Introduction

IoT device production is increasing year by year. Since keeping CPU and memory constraints directly affects device cost, efficient application/software development is required. As a result, the existence of an real-time operating system (RTOS) is essential for sharing various resources among different tasks. For example, ITRON [1] has a long history of open specification RTOS, Mbed OS [2] targets ARM Cortes-M processor, FreeRTOS [3] was acquired by Amazon in recent year, and Azure RTOS ThreadX [4] was acquired by Microsoft.

In order to maintain real-time performance, efficient scheduling algorithms have been studied. The main purpose of real-time scheduling algorithms is to achieve optimal scheduling for tasks which have period, execution time and/or deadline. It is important for the algorithms not only to meet deadline of each task but also to shorten its response time and jitter. For periodic task sets, there are two categories in scheduling algorithms; one is Rate Monotonic (**RM**) for static priority and the other is Earliest Deadline First (**EDF**) for dynamic priority [5]. EDF has the advantage of high CPU utilization, while RM causes less runtime overhead and has predictable behavior. In RM, tasks with shorter periods have higher priorities. In contrast, ones with longer periods are likely to suffer from increased response times and jitters due to their lower priorities.

We proposed Execution Right Delegation (**ERD**) method based on RM where a high-priority server for a particular (or important) task[i] is introduced in the previous study [6]. In this paper, we implement the proposed scheduling algorithms on FreeRTOS without kernel modification. In order to avoid kernel modification, the scheduling algorithm of ERD is implemented as a slightly simplified version, named ERD-light.

This paper consists of six sections. Section 2 describes related work in terms of actual implementation study of real-time scheduling algorithms. Section 3 reviews algorithm of ERD method in brief. Section 4 shows requirement and implementation of ERD-light. Evaluation of the performance is shown in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

2.1 RTOSes and Scheduling Algorithm

The requirements for an RTOS are to ensure that time constraints are met for the operation of a given computation (or process, task). For this purpose, the RTOS must be able to schedule a given task within the expected time without missing the deadline. In addition, it is necessary to prevent that a low-priority task interferes with a high-priority task which causes a deadline miss. Satisfying these time constraints is referred to as ensuring real-time performance. To ensure real-time performance, RTOS implements a real-time scheduler. Many RTOSes, such as ITRON, have

[1] Japan Advanced Institute of Science and Technology, Tokyo/Ishikawa, Japan
[a)] hal-suzuki@jaist.ac.jp
[b)] kiyofumi@jaist.ac.jp

[i] We assume the particular task has relatively lower priority due to its longer period. For example, for CAN messages in integrated ECUs, control messages have a shorter period, but notification messages have a longer period. However, some notification messages are urgent and must receive a higher priority (e.g. warning lamp, low fuel).

a fixed-priority-based scheduler that is designed to prevent a low-priority task from blocking a high priority task.

When designing a real-time system with multiple tasks on an RTOS, there is a theory of scheduling algorithms that models the system in order to examine whether those tasks satisfy the time constraints. In many RTOSes, processing is realized by persistent tasks, periodic tasks, and aperiodic handlers invoked by interrupts. The models in scheduling algorithms mainly treat these processes as periodic tasks [ii]. RM is a typical model for handling periodic tasks in static priority and EDF is for dynamic priority.

In the RM model, task $\tau_i (1 \leq i \leq n)$ releases an infinite sequence of jobs. Once job is released, it runs during the defined time C_i. Job is released every period T_i. Notation of a task is $\tau_i = (C_i, T_i)$. A set of tasks is denoted as $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, where the smaller subscript figure a task has, the shorter period and higher priority it has (i.e. $T_1 \leq T_2 \leq ... \leq T_n$). A deadline miss occurs if job does not finish by the next task release. These are theories which provide the correctness of timing guarantee from a mathematical point of view. In RM, utilization-based exam [5] and Response Time Analysis [7] are well known.

2.2 Scheduling Algorithm in RTOS

RM is easy to realize on static-priority-based RTOSes such as ITRON, Mbed OS and FreeRTOS by associating tasks' priorities with their periods. In contrast, EDF requires specific implementation inside RTOS. There are a few RTOSes that implement EDF (e.g. MaRTE OS [8], Plan 9 [9]). Linux [10] supports not only time-sliced scheduling but also static priority scheduling by SCHED_FIFO/SCHED_PR policy. EDF is also supported by SCHED_DEADLINE policy by Linux since version 3.14.

While many scheduling algorithms have been proposed, the evaluation was done on a simulation basis, or was performed on real OS with the changed scheduler. Sape et al. studied the implementation of the real-time scheduling algorithm in Plan 9 and discussed findings that come from actual operations including the cause of deadline misses ([11] [12]).

In an embedded system with memory and CPU constraints, it is important to reduce the amount of memory used by the OS and to reduce the overhead during operation. Some proposals for scheduling algorithms have been evaluated by modifying the Linux kernel, but the evaluations have focused only on testing task response time and schedulability, not on overhead of scheduling cost itself ([13], [14]). Saranya et al. evaluated partitioning based scheduling algorithm for multicore processors with respect to operational overhead by modifying Linux [15].

While there are proposals to change the kernel scheduler, most RTOSes in production are well tested and their operation is guaranteed. In particular, some RTOSes, such as eT-Kernel [16] and QNX [17], are compliant with ISO-26262 [18] in consideration of automotive applications in order to ensure a certain level of safety. It is not desirable from a safety standpoint to modify the OS scheduler even to introduce a better scheduling algorithm. In this paper, we utilize the APIs provided by the RTOS to realize a simplified version of the ERD algorithm proposed by the authors in the past, without modifying the kernel.

3. Execution Right Delegation (ERD)

3.1 Definition of ERD

Before implementing ERD on actual RTOS, we show ERD algorithm in this section. [6] shows detailed definitions, theorems and examples. ERD is a method to shorten response time and jitter of privileged task, τ_p, in a task set by using a high-priority virtual server, VS, which has capacity of C_s and period of T_s while satisfying all deadlines of the task set. We assume τ_p has a relatively lower priority due to its longer period. By making the priority of VS high with short T_s, τ_p can be executed at the high priority while consuming C_s. The behavior of VS is based on Priority Exchange (PE) [19].

In ERD, the target system model is based on fixed task priority where each task τ_i has its execution time C_i and period T_i, and its deadline D_i is equal to its period. A task does not have a phase, which means that the first job is released at time instant $t = 0$. The scheduling rule follows RM except for a privileged (target) task which the proposed virtual server (**VS**) is applied to.

Definition 3.1 (Delegation of Execution Right): *In ERD method, VS is scheduled based on RM rule. When VS is given the execution right, a privileged task τ_p is executed instead of VS. This situation is called Delegation of Execution Right. If a job of τ_p has already finished when the execution right is available, the behavior follows PE rule where jobs of the other tasks are executed while the server capacity is accumulated at the priority level of the running job.*

Next, the following definitions gives the algorithm for finding C_s and T_s.

Definition 3.2 (Response Time Analysis (RTA) [7] of RM): *In a fixed-priority scheduling, the longest response time*[iii]*, R_i, of task τ_i is computed as:*

$$R_i = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil C_j. \quad (1)$$

Definition 3.3 (Candidates of VS): *Let Γ be a schedulable task set and τ_p in Γ be a privileged task whose response time and jitter should be shortened. Then, RTA in Definition 3.2 is applied to $\tau_1, ...,$ and τ_p. From the relation between R_p and $T_1, ..., T_{p-1}$, C_s and T_s for VS are obtained as follows.*

[ii] Although some scheduling algorithms deal with aperiodic tasks and sporadic tasks that have a minimum operation time instead of period, this paper deals only with periodic tasks.

[iii] If the first jobs of all tasks are released simultaneously at the instant $t = 0$, response time of the jobs becomes the worst-case for the corresponding task (Critical Instant [5]).

$$C_s = \begin{cases} C_p, & \text{if } R_p \le T_{p-1} \quad (2) \\ idle'(T_s), & \text{otherwise} \quad (3) \end{cases}$$

$$T_s = \begin{cases} T_h, & \text{if } R_p \le T_{p-1} \quad (4) \\ t \in \Psi, & \text{otherwise} \quad (5) \end{cases}$$

where

$$\begin{aligned} \Psi &= \{T_1, T_2, ..., T_{p-1}\} \\ T_h &= min(\ \{t \mid t \in \Psi,\ R_p \le t\}\ ) \\ idle'(t) &= t - \sum_{j=1}^{p-1} \lceil \frac{t}{T_j} \rceil Cj \quad (t \in \Psi) \end{aligned}$$

The following example shows how ERD works.

Example 3.1 (ERD method)**:** With $\Gamma = \{(1,5),(2,6),(4,13)\}$ and $\tau_p = \tau_3$, RTA gives $R_1 = 1$, $R_2 = 3$, and $R_3 = 10$ (**Fig. 1**). VS is given by the equations (3) and (5) as $R_3 > T_2$. With $\Psi = \{T_1, T_2\}$, $idle'(T_j)$ is calculated for each period. Since $idle'(T_1) = 2$, $\text{VS}_1 = (2,5)$ is derived. Similarly, $\text{VS}_2 = (2,6)$ is obtained from $idle'(T_2) = 2$. In this time, we apply VS_1 for example.

Fig. 2 shows scheduling result of ERD. At the instants 1, 2, 6, and 7, delegation of execution right can be confirmed. As a result, response time of τ_p's first job is reduced from 10 to 7.

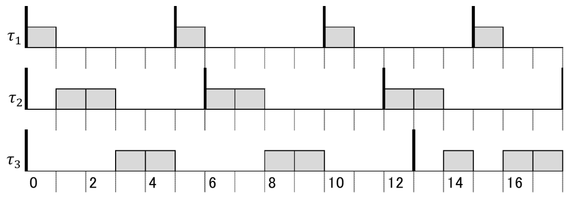

Fig. 1 Schedule by RM for $\Gamma = \{(1,5),(2,6),(4,13)\}$.

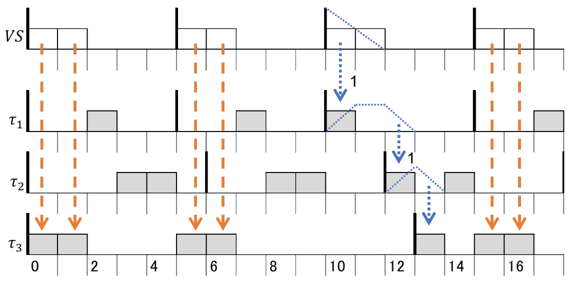

Fig. 2 Schedule by ERD with VS_1 for Γ.

It is worth noting that Deadline Monotonic (**DM**) [20] scheduling does not shorten the response time of τ_3 with the above Γ. DM is effective only when the relative deadline of τ_3 can be made less than or equal to the deadline of a higher-priority task. For Γ in this example, if the deadline is set to be less than or equal to T_2, τ_2's job results in missing its deadline.

The worst-case response time (**WCRT**) of a task in RM is response time of the task's first job (Critical Instant), which is calculated by (1). Unlike RM, Critical Instant of ERD is not given by task's first job. [6] describes WCRT, RTA and simulation based results in detail.

3.2 Priority Exchange and ERD-light

VS in ERD operates according to the rules of PE. Specifically, when VS has execution right and τ_p is running (or ready), VS's priority capacity is used for τ_p. Otherwise, when VS has execution right but τ_p is not running (or not ready), VS's priority capacity is exchanged to the task's capacity in ready que with the next highest priority. This behavior is confirmed from the time instant 11 to 12 of Fig. 2. At the instant 11, VS has execution right but τ_p's job is not running (already finished). VS's priority capacity is exchanged to τ_1's priority capacity, with the amount of 1. At the instant 12, the VS's capacity is discarded due to no existence of jobs. Then, τ_1's priority capacity is exchanged to τ_2's priority at the instant 13 since τ_3 is not ready. At the instant 14, τ_3 is ready and τ_2's priority capacity is used for τ_3 as the rule of PE. This PE rule is strictly required to keep schedulability. If no exchange is performed but deferred execution right is allowed, τ_2's 3rd job misses its deadline (**Fig. 3**).

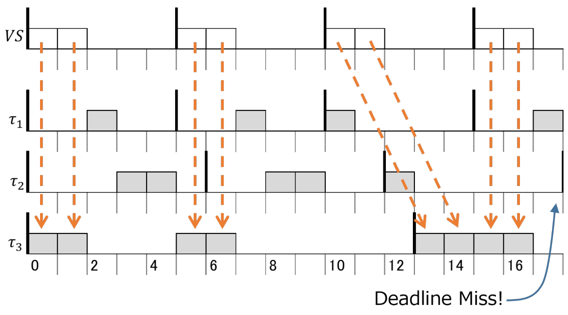

Fig. 3 Deferred ERD is not allowed.

In order to realize ERD in actual RTOS without kernel modification, we relax priority exchange rule and named it ERD-light. In ERD-light, when VS has execution right, capacity of VS is discarded unconditionally if τ_p's job is not ready. In other words, priority exchange does not occur. **Fig. 4** shows an example. The left figure shows the original ERD with priority exchange, and the right figure shows ERD-light. At the instant 25 to 27, priority exchange is confirmed in the left figure. On the other hand, VS's first execution capacity is discarded at the instant 25 in the right figure. As a result, behaviors of τ_2 and τ_3's jobs are different between the left and right figures. Nevertheless, response time of τ_3's job is the same, 31.

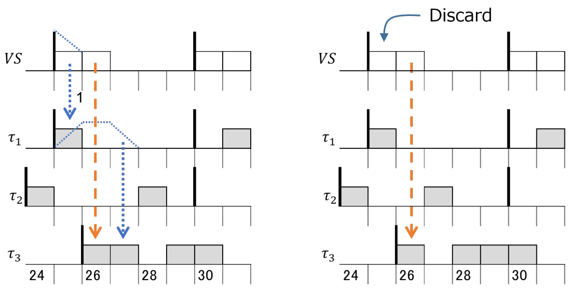

Fig. 4 Priority Exchange vs ERD-light.

4. Requirements and Implementation

In this section, we show requirements and implementation methods of ERD-light on RTOS without kernel modification. We use FreeRTOS V10.4.4 [iv] for reference implementation in this paper. In regard of requirements, first, we have the following three premises:

A1 A task is implemented as a periodic task.
A2 A task with shorter period has higher priority.
A3 Priority of a task can be changed dynamically.

Most of RTOSes support the above requirements except A3. In the case of FreeRTOS, A1 is satisfied. A periodic task τ_i is created by xTaskCreate() as a normal task at first. Once specific work that requires time period C_i is finished, the task calls vTaskDelayUntil() in order to change the state to blocked/sleep until T_i period. A2 is also met by that priority of a task can be given by xTaskCreate(). In FreeRTOS, vTaskPrioritySet() satisfies A3.

Second, ERD-light requires RTOS which supports the following APIs (or System Call/Service Call):

B1 API which provides the current system tick.
B2 API which provides the elapsed time in ticks that was consumed by a task.

In regard of B1, most RTOSes provide this function. On the contrary, a few RTOSes provide B2. For FreeRTOS, xTaskGetTickCount() and vTaskGetInfo() satisfy the above requirements, respectively.

Finally, ERD-light requires two timer handlers that most RTOSes support. The timer handlers are invoked every VS period and specific tick, which is discussed in the next paragraph. In the implementation of ERD-light, the task information (WCET, period) and VS are given in advance. The WCRT ($=R_S$) of VS is assumed to be calculated by the formula (1). There is no additional description in the task code to realize ERD-light. ERD-light can be implemented by adding two timer handlers that control priority of τ_p and a few tens of bytes of memory to store status of operation.

Algorithm 1 describes pseudo code of the timer callback handler which corresponds to VS. The callback is invoked every VS's period. The objective of this handler is to raise τ_p's priority. First, the absolute tick when the callback is invoked is given by B1 at line 1. At line 2, the number of ticks that are spent by τ_p is retrieved by B2. Finally, priority of τ_p is changed to the same priority as VS at line 3 (requirement of A3). By changing the priority of τ_p, τ_p can operate as a high-priority task, and behave in the same way as the example in the previous section.

Next, pseudo code of the timer handler, whose role is to restore priority of τ_p to the original (base priority), that is invoked every specific tick is described in **Algorithm 2**. In our implementation, period of the handler is 100 ticks [v].

[iv] API and configuration is described in [21].
[v] Fewer cycles allow for more precise priority control, but they also increase the overall overhead of the system. From our evaluation, 100 ticks are enough to keep accuracy.

Algorithm 1 VS Callback

```
1: $st = current\_tick()$
2: $ccon = consumed(\tau_p)$
3: $\tau_p.priority = vs.priority$
```

Algorithm 2 Tick Callback

```
1: if $\tau_p.priority = \tau_p.base\_priority$ then
2:     return
3: end if
4: $now = current\_tick()$
5: if $st + R_S \leq now$ or $consumed(\tau_p) - ccon > C_S$ then
6:     $\tau_p.priority = \tau_p.base\_priority$
7: end if
```

Line 1 checks whether priority of τ_p is changed. If not, it quits the operation. Recalling Section 3.2, unused capacity is required to be discarded. If the capacity is not discarded, the response time of task τ_h, whose priority is higher than τ_p but lower than VS, is increased.

Because of the implementation complexities of discarding capacity, we adopt the following approach. To eliminate the influence on τ_h, it is necessary to ensure that τ_p does not run with VS's priority beyond the WCRT(R_s) of VS. Regardless of whether the capacity of VS has been consumed or not, the requirement is satisfied by restoring the priority of τ_p to the original at the time instant R_s has passed since the VS was released (the first condition of line 5). In addition, priority of τ_p must be restored to the original when VS's capacity is fully consumed (the second condition of line5).

5. Evaluation

In our evaluation, we use FreeRTOS on Raspberry Pi Pico [22] Single Board Computer (SBC). Source code is available at [23]. We test 4 task sets (**Table 1**), which include τ_p whose response time cannot be shortened by Deadline Monotonic except Test Set1. We compare the results those of our simulation that is presented by our previous work. Even though the evaluation is performed on actual RTOS and SBC, not python-based simulation, we implement every task by using busy loop until its C_i to *simulate* consumption of CPU resource since timing correctness is required in our study. In the busy loop, 1 tick is 12,475 Clock Count (e.g. a period of 10,000 ticks requires 12,475 * 10,000 Clock Count). In the evaluation, WCRT of τ_p is compared to the simulation-based ERD (priority-exchange version).

Table 2 shows the results. In the table, ERD-l(RT) represents WCRT values of FreeRTOS implementation of ERD-light, and ERD(Sim) represents WCRT of python-based simulation, respectively. The table also contains results of RM (fixed-priority based result, without ERD-light implementation) on FreeRTOS, and python-based simulation of RM. With a comparison, the implementation of ERD-light

Table 1 Task Sets.

		C_i	$T_i, (R_p)$	pri
Test Set1	τ_1	2000	4000	3
	τ_2	3000	12000	1
	τ_p	3000	14000	0
	VS	3000	12000, (12000)	2
Test Set2	τ_1	2000	5000	2
	τ_2	2000	7000	1
	τ_p	2000	10000	0
	VS	1000	5000, (1000)	3
Test Set3	τ_1	1000	5000	2
	τ_2	2000	6000	1
	τ_p	4000	13000	0
	VS	2000	5000, (2000)	3
Test Set4	τ_1	1000	5000	4
	τ_2	1000	6000	3
	τ_3	2000	8000	1
	τ_p	4000	14000	0
	VS	2000	8000, (4000)	2

on FreeRTOS is confirmed to be correct since WCRTs of each task are almost the same as the simulation (excluding calculation/clock ratio errors). Importantly, WCRT of τ_p is shortened compared with the results of RM.

Table 2 Evaluation Result (WCRT).

		ERD-l(RT)	ERD(Sim)	RM(RT)	RM(Sim)
Test Set1	τ_1	1998	2000	1998	2000
	τ_2	11990	12000	6994	7000
	τ_p	6994	7000	11990	12000
Test Set2	τ_1	2998	3000	1998	2000
	τ_2	4996	5000	3997	4000
	τ_p	5995	6000	9992	10000
Test Set3	τ_1	2999	3000	999	1000
	τ_2	4997	5000	2998	3000
	τ_p	8992	9000	9992	10000
Test Set4	τ_1	999	1000	999	1000
	τ_2	1998	2000	1998	2000
	τ_3	7996	8000	3997	4000
	τ_p	9992	10000	13989	14000

Table 3 shows results of GNU size command [vi] to ERD-l(RT) and RM(RT) program images for Test Set1. It is confirmed that footprint penalty of ERD-light is less than 256 bytes. In regard of runtime overhead penalty, we gain stat information by using vTaskGetRunTimeStats() API. We could not find any differences in CPU usage since the timer call back usage was less than 1%.

Table 3 Program sizes (bytes).

	text	data	bss
ERD-light	43,212	36	53,544
RM	43,008	36	53,508

6. Conclusion

We proposed ERD-light, a simplified version of ERD, and showed requirement definitions for implementing it on RTOS without kernel modification, by combination of APIs. With FreeRTOS on Raspberry Pi Pico, ERD-light was implemented and evaluated with several task sets. We confirmed the correctness of the requirement and implementation from the results of experiment that shows the same result as the python-based simulation. Implementations penalty of footprint is less than 256 bytes. We also measured runtime overhead by stat API and confirmed the overhead is less than 1% of CPU usage.

In regard of the algorithm, a real-time penalty of discarding capacity in ERD-light is not compared to ERD (priority change version) in our evaluation since the aim of this paper is to propose requirement and implementation of our proposed algorithm on actual RTOS without kernel modification. Comparing ERD with ERD-light from real-time performance point of view with large task sets is future work.

[vi] arm-none-eabi-size Version 2.27.

Acknowledgments

This work is supported by JSPS KAKENHI Grant Number JP 19K11873.

References

[1] http://www.tron.org/ja/wp-content/themes/dp-magjam/pdf/specifications/ja/WG024-S001-04.03.03.pdf
[2] https://os.mbed.com/mbed-os/
[3] https://www.freertos.org/
[4] https://docs.microsoft.com/azure/rtos/threadx/
[5] Liu, Chung Laung, and James W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment." Journal of the ACM (JACM) 20.1 (1973): 46-61.
[6] Takaharu Suzuki and Kiyofumi Tanaka, "Response Time Analysis of Execution Right Delegation Scheduling." Proc. of Asia Pacific Conference on Robot IoT System Development and Platform, pp.32–38, 2020.
[7] Joseph, Mathai, and Paritosh Pandya. "Finding response times in a real-time system." The Computer Journal 29.5 (1986): 390-395.
[8] https://marte.unican.es/
[9] http://doc.cat-v.org/plan_9/
[10] https://www.kernel.org/
[11] Mullender, Sape J., and Pierre G. Jansen. "Real time in a real operating system." Computer Systems. Springer, New York, NY, 2004. 213-221.
[12] Mullender, Sape, and Jim McKie. "Real Time in Plan 9." Proceedings of the 1st International Workshop on Plan. Vol. 9. 2006.
[13] Brandt, Scott A., et al. "Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes." RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003. IEEE, 2003.
[14] Abeni, Luca, and Tommaso Cucinotta. "EDF scheduling of real-time tasks on multiple cores: adaptive partitioning vs. global scheduling." ACM SIGAPP Applied Computing Review 20.2 (2020): 5-18.
[15] Saranya, N., and R. C. Hansdah. "Dynamic partitioning based scheduling of real-time tasks in multicore processors." 2015 IEEE 18th International Symposium on Real-Time Distributed Computing. IEEE, 2015.
[16] https://www.esol.co.jp/embedded/et-kernel.html
[17] https://blackberry.qnx.com
[18] https://www.iso.org/standard/68388.html
[19] Lehoczky, John P., Lui Sha, and Jay K. Strosnider. "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments." Proc. of IEEE Real-Time Systems Symposium, pp.261–270, 1987.
[20] Audsley, Neil C., et al. "Hard real-time scheduling: The deadline-monotonic approach." IFAC Proceedings Volumes 24.2 (1991): 127-132.
[21] https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf
[22] https://www.raspberrypi.org/products/raspberry-pi-pico/
[23] https://github.com/takahalsuzuki/erd_freertos