**DOI.** 10.1109/ROSE.2014.6953035

# Hard Real-Time Execution Environment Extension for FreeRTOS

Cristina S. Stangaciu, Mihai V. Micea, Vladimir I. Cretu
"Politehnica" University of Timisoara
Computer and Software Engineering Department
2, Vasile Parvan Blvd., 300223, Timisoara, Romania
certejan@dsplabs.cs.upt.ro, mihai.micea@cs.upt.ro, vladimir.cretu@cs.upt.ro

***Abstract*— In this paper, a hard real-time execution environment extension is proposed for an open source real-time operating system, FreeRTOS, in order to support a special case of hard real-time tasks, called ModXs. The goal is to obtain a real-time system which has both the capabilities offered by a dynamic, preemptive, priority based scheduling and execution environment and the determinism and predictability of a hard real time execution environment. This paper also presents an implementation of the system which was tested and validated on a hardware platform EFM32-G8900-STK.**

***Keywords—hard real-time (HRT), real-time operating system (RTOS), execution context, jitter***

## I. Introduction

Most of the control systems, including sensor based control system used in robotics and industrial automation, have a real-time functionality and are based on real time operating systems (RTOSs). As it is well known, a real-time system's correct functionality is not given only by the correct outcome of the system but also by its correct timing functionality (i.e. all the time constraints must be met). We note the difference between hard real-time (HRT) constraints and soft real-time (SRT) constraints. While for the SRT applications a time constraint violation has an impact only in the degradation of the outcome quality (e.g. the loss of a number of frames in a video processing application leads to image quality degradation), for the HRT applications, the violation of a time constraint has catastrophic impacts (e.g. an overcome of the maximum allowed response time of an airbag system can cause loss of human lives).

Hard real-time tasks have a number of requirements, from which we mention, beside the timing constraints, the predictability, the deterministic behavior and a special care regarding resource sharing. Moreover, regarding how "hard" (critical) are the time constraints, the scheduling policy becomes more restrictive. A representative example could be a set of tasks which perform data acquisition and require a perfectly periodical execution were no jitter is allowed. Another examples of tasks with HRT constraints, part of a predictable data communication protocol are presented in [1]. As revealed in [2] the jitter in the most common Hard RTOSs is: "*typically on the order on microseconds to a few tens of microseconds*" and in Soft RTOSs it "*ranges from milliseconds to seconds*"! In the same article, it is shown the impact of this jitter on a stepper motor control: an increase jitter can cause the loss of position steps. As a result, these tasks cannot be scheduled with numerous scheduling mechanisms (e.g. the mechanisms based on queues are not recommended). On the other hand, a restrictive static, scheduling mechanism neither would allow the schedulability of an increase number of tasks nor the dynamic addition of tasks. As a result, if we look at the tasks in a pessimistic manner and consider all of them to be HRT tasks, we may well fail to schedule a set of tasks, which are in fact feasible if they are scheduled by other means.

In practice, on a real time system, we have both tasks with hard real time constraints and tasks with soft real time constraints, thus it is important for a RTOS to manage them both.

In this paper we propose and describe a hard real-time execution environment extension to a popular RTOS called FreeRTOS. In section II we make a very brief presentation of other approaches that we have accounted and studied. Section III briefly describes FreeRTOS' funtionality and section IV describes the execution environment, HARETICK, which will represent the HRT extension of the operating system. In section V, the integration of the two components: FreeRTOS and HARETIC is presented. The implementation is analyzed and evaluated in section VI and we conclude the paper in section VII.

## II. Related Work

In order to provide time guarantees and predictability to HRT tasks there have been proposed a number of solutions based on hybrid scheduling methods in which HRT tasks are scheduled by static offline scheduling techniques, which provide usually non-preemptive execution, and SRT tasks are scheduled by dynamic on-line scheduling techniques.

One attempt to combine the offline scheduling techniques with the online ones is to "*translate the off-line schedules into task attributes for fixed priority scheduling*" [3]. This method translates the task's attributes into the ones requested by the fixed priority scheduling, and assigns fixed priorities to the tasks. This method has the disadvantage of increasing the number of tasks by splitting them into instances of a number of tasks, which can generate offset or priority assignment conflicts. The method uses time target windows, which reduces the granularity for task execution and task launching.

Another hybrid scheduling technique is proposed in [4] where the tasks are divided into sets according to some common features. Tasks are then scheduled within a set (each set can have its own algorithm). There is also a global scheduler for the sets. In the article, this approach is presented only as a principle, without providing any concrete scheduling mix.

A more popular solution is the so called Hierarchical Scheduling [5, 6]. The main idea of this method is to temporally isolate the hard real-time tasks from each other and/or the rest of the tasks by using servers, so the tasks execute within their associated server. In this case, the behavior of the system is determined by two factors: the main scheduling and execution mechanism which manages the server and the behavior of the task within the server. Such a scheduling support was implemented and described in [7] and [8] on the base of the *FreeRTOS* operating system [9].

## III. FreeRTOS Overview

### A. General Features

FreeRTOS, provided by Real Time Engineers Ltd., is an open source, small, scalable and well documented real-time operating system, which has been supported on more than 30 different hardware architectures. All these features make FreeRTOS a popular real-time operating system. Moreover, its sources are written mostly in C with very few extra assembly routines, which make the system portable and easy to understand. Another noticeable feature is its kernel small size: it requires about 5 to 10 kB of ROM, depending on the compiler, architecture and kernel configuration. On the other hand it takes from up to a few kB of RAM to all the available RAM, depending on the number of tasks, the number of queues created and the memory allocation schemes [10]. Each task has its own allocated stack (the minimum default stack size being 140 words for the ARM Cortex M3 implementation). FreeRTOS also provides three modes for implementing the heap. The simplest mode implements the memory allocation, but no memory de-allocation or memory reuse while the most complex mode implements both allocation and de-allocation [11].

Regarding timing in FreeRTOS, we can speak about the time representation in the system and the time overhead introduced by the context switch. System time is represented by a core counter that provides a periodical interrupt which increments the system time (and has also task dispatcher functionality in the case of a preemptive operation). In the ARM Cortex M3 implementation, the System Tick Counter is used for this purpose. The frequency at which the RTOS tick operates is defined by the `configTICK_RATE_HZ` directive. A higher frequency gives a better time resolution, but the frequency is upper bounded, on one hand by the timer hardware support resolution and, on the other hand, by the system's capability to handle frequent interrupt service routine calls. Another time parameter taken into consideration is the context switch time of the FreeRTOS, which is dependent on the architecture, compiler and configuration. A context switch time of 84 CPU clock cycles has been obtained on ARM-Cortex M3, for the Keil compiler, optimized for speed [10], plus an interrupt entry penalty which is typically 12 CPU clock cycles, on this architecture.

### B. Scheduling Mechanism

FreeRTOS is a preemptive or cooperative system, which provides dynamic task scheduling (highest priority first, for tasks with different priorities and round robin among tasks with the same priority).

In both preemptive and cooperative mode, the scheduler operates as a periodical timer interrupt service routine. In the cooperative mode, the timer interrupt service routine (ISR) has only the purpose to increment the system time. In the preemptive mode, the timer ISR also acts as a task dispatcher: it saves the current execution context on the stack and, if a context switch is required, it checks if a higher priority task has been unblocked. In such a case, the context is switched to this higher priority task. After the task execution terminates, the context is restored and the processor exits the ISR [11].

In FreeRTOS, task scheduling is based on priorities. The scheduler chooses the highest priority task from the ones that are in the ready state. The scheduling policy of tasks with the same priority is round robin. The operating system also creates an *idle* task, which has the lowest priority and is responsible for auxiliary duties (i.e. memory management). The scheduling management implementation is based on lists. Each possible priority has its own list of tasks, where they can be created and added dynamically. This kind of approach lacks determinism, being a source of jitter in task execution, which makes the operating system neither suitable for hard real-time tasks, nor for control systems.

## IV. HARETICK: a Hard Real-Time Execution Environment

HARETICK (<u>Ha</u>rd <u>Re</u>al-<u>Ti</u>me <u>C</u>ompact <u>K</u>ernel) [12, 13] is a hard real-time operating kernel in continuous development, since 2000, by the DSPLabs team (Digital Signal Processing Laboratories), at the Politehnica University of Timisoara. It is currently prototyped for embedded platforms such as the ARM7-based Philips LPC2xxx and ARM Cortex-M3-based EFM32-G8xx microcontrollers.

HARETICK is designed as a single-user, multitasking, hybrid real-time operating kernel for embedded, DSP or small microcontroller based platforms, able to provide maximum predictability for hard real-time applications. It is a hybrid real-time operating kernel, providing support for two concurrent task execution environments: the HRT context, for the execution of hard real-time tasks in a non-preemptive manner, and the SRT context, for the execution of soft real-time (or regular) tasks in a classical, preemptive and priority-based manner. Its main components are presented in [12] and the key features used in our implementation will be discussed in the following sections. Until now, the focus was on the development of the HRT execution context with different non-preemptive scheduling policies [12, 13], from which we have been especially interested in FENP.

The FENP algorithm is a non-preemptive time-triggered, cyclic scheduling algorithm, used for scheduling hard real-time, perfectly periodic tasks. At the offline analysis step, it statically assigns a start time to each task, within its period. This start time can be seen as a constant offset (phase), calculated from the first release instant. All the next execution instances (jobs) of a FENP task can be further calculated by adding the task period to the start time of the previous instance.

## V. Integration of the HARETICK and FreeRTOS

### A. Tasks Model and Functionality

In the proposed extended operating system we use two sets of tasks: $\mathbf{S}^{FENP}$ as the set of hard real-time tasks (ModXs [12]) and the set **S** of simple real-time tasks. All the ModX are scheduled by the FENP algorithm, which can briefly be described as follows: the execution (start time) of any two consecutive jobs of a ModX $\mu_i$ in $\mathbf{S}^{FENP}$ is separated by a time interval equal to the task period ($T_i$).

$$s_{i,k+1} = s_{i,k} + T_i \ . \ \ 1 \le i \le m \qquad (1)$$

As a consequence of (1), the start time of any execution instance (job) of $\mu_i$ can be statically determined in a direct manner:

$$s_{i,k+1} = s_{i,1} + kT_i = \varphi_i + kT_i \qquad (2)$$

where $\varphi_i$ is the start time of the first job of $\mu_i$ (also called the "phase" of $\mu_i$). Equations (1) and (2) formally describe the scheduling principle of this algorithm: the start time of the next instance *(k + 1)* of a FENP task $\mu_i$ is computed by adding the task period $T_i$ to the start time of the current instance, $s_{i,k}$.

Another particular feature introduced by the FENP method to the task model presented in the previous section is that the relative deadlines are equal to the corresponding periods:

$$D_i = T_i \ , \ \ \forall \mu_i \in \boldsymbol{S}^{FENP} \qquad (3)$$

As a result, the timing behavior of a task $\mu_i$ in the $\boldsymbol{S}^{FENP}$ subset can be fully determined by the tuple:

$$\mu_i \equiv (\varphi_i, C_i, T_i) \ . \ \ 1 \le i \le m \qquad (4)$$

where $C_i$ is the execution time of the task.

The simple real-time tasks are determined by:

$$\theta_i \equiv (C_i, T_i, P_i) \ . \ \ 1 \le i \le m \qquad (5)$$

where $P_i$ is the priority of task $\theta_i$.

These tasks can be scheduled by a priority based real-time scheduling mechanism.

While the ModXs cannot be preempted by any other tasks (including other ModXs), the simple real-time tasks can be preempted, as can be seen in Fig. 1. Here, $M_i$ are ModXs and $T_i$ are tasks. The oblique lines mark the preemption points within a task (note: $T_1$ has higher priority than $T_2$).

Next we will present a system where the simple tasks can be scheduled and executed by the FreeRTOS, while the ModXs are scheduled and executed by an extension of the FreeRTOS based on the HARETICK kernel.

### B. Proposed Solution

We have started our implementation using the FreeRTOS version v7.5.2, on the ARM Cortex–M3 architecture [14]. As hardware platform we used the EFM32-G8xx microcontroller, because of its increased popularity, accessible price and high performance interrupt capabilities.

The main idea is to make use of a timer as the source of the system time and the corresponding ISR as the execution context of the HARETICK kernel, while preserving unaltered the system structure of the FreeRTOS. In the referenced implementation of the FreeRTOS, the system time is supported by the System Tick Timer, which is also the main source of interrupts, followed by other software sources of interrupts, such as the SVC and PendSV. The extension of a new execution environment to the existing FreeRTOS version is possible due to the complex management of the NVIC (Nested Vector Interrupt Controller) in ARM-Cortex architecture, which can isolate the interrupt sources and nest them based on priorities.

The main HARETICK modules, which are used in this implementation, are: BOOT, SYSINIT, LOADER, HSCD and HDIS. The SSCD module is replaced by the FreeRTOS scheduling mechanisms.

The BOOT component of the HARETICK system is responsible for loading and starting the system. This module also initializes the system processor and the basic access to the system memory. SYSINIT is the system initialization module, lauched by the BOOT task after finishing its own functions. The main role of the SYSINIT task is the initialization of the main HARETICK parameters like global variables, system tables, system pointers etc. The task responsible for loading the user applications into the system is the LOADER component. The first step of the LOADER task is to add a new record into the PDT (Process Descriptor Table) for every hard real-time task or executable module (ModX) in the system. The crucial fields needed to be set in a PDT record are: the process ID (PID), the time specific fields (the execution period, the worst case execution time – WCET, etc.), the address of the

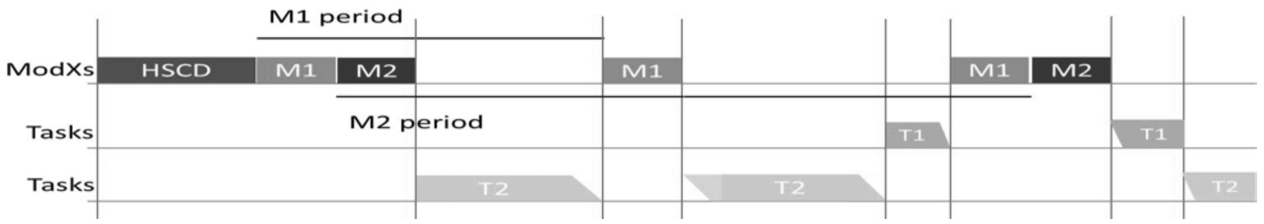


Fig. 1. Example of Task Scheduling and Execution

application code, the address of the memory zone for input/output parameters and global variables. These fields are used later by the HDIS executive and the Hard Real Time Scheduler (HSCD). The LOADER also allocates and initializes the amount of memory required by every ModX (for global variables and for input/output parameters). The Hard Real Time Scheduler (HSCD) is the only task with the ability to schedule the HRT context. Mainly, it manages the HDIS table (Hard Real Time Dispatcher Table) used by the HDIS executive to launch the appropriate ModX. The first invocation of the HSCD behaves slightly different and is represented by a distinct function, called PREHSCD. PREHSCD implements, beside the HSCD scheduling itself (*PREHSCD_scheduling* from Fig. 2), some initializations for the ModX (*Init_tasks*), the system initialization for the FreeRTOS (*FreeRTOS_Init*). The functionality attributes and function calls from FreeRTOS_Init are presented in Fig. 3: the initialization of all the global variables used in FreeRTOS, the creation of some demo tasks, for now, with the help of *xTaskCreateRestricted()* function. FreeRTOS_Init also calls a slightly modified version of the *vTaskStartScheduler()* from FreeRTOS, which does not call *prvStartFirstTask()* and after that, *FreeRTOS_Init* enables FreeRTOS interrupts. After the *FreeRTOS_Init* call, PREHSCD starts the HARETICK extension by setting the timer for the first interrupt and enabling the timer interrupt. In the end, the PREHSCD module starts the FreeRTOS by calling *prvStartFirstTask()* from FreeRTOS. All these are presented in Fig. 2, where regular rectangles represent the original HARETICK modules and the rounded rectangle depicts the modified version of PREHSCD. The HARETICK modules are marked with underlined capital letters, the HARETICK function calls are written with regular (non-italic) text and the new implemented functions are written with italic. In Fig. 3 are extended the function calls *FreeRTOS_Init()* and *StartFreeRTOS()* from Fig. 2.

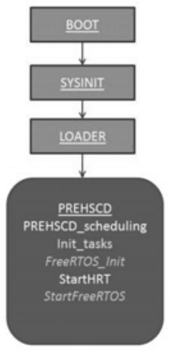


Fig. 2. System Initialization Modules

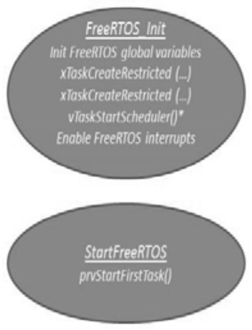

FreeRTOS_Init
Init FreeRTOS global variables
xTaskCreateRestricted (...)
xTaskCreateRestricted (...)
vTaskStartScheduler()*
Enable FreeRTOS interrupts
StartFreeRTOS
prvStartFirstTask()

Fig. 3. FreeRTOS Initialization Modules

The HARETICK extension of FreeRTOS uses a specific mechanism to launch the ModX into execution. This mechanism is called HDIS (Hard Real Time Dispatcher) and is an important component of the HARETICK system. The HDIS component is activated by the timer interrupt and may be considered an interrupt service routine. Once in execution, HDIS first saves the interrupted execution context (the FreeRTOS system or tasks). Further on, it identifies the current ModX to be executed, extracts from the HDIS table the next task to be executed as well as its start time (i.e. the system time value of the next occurrence of the timer interrupt) and programs the processor for this interrupt. The job of this component does not end with launching the currently scheduled task. It also has actions left to do after the task finishes execution. These actions consist of advancing the pointer of the HDIS table to the next task to be executed and restoring the execution context back to the FreeRTOS. Therefore, the HDIS executive is divided into two components: the HDIS Prefix (HDIS_PRE), which is executed by the timer interrupt service routine, and the HDIS Suffix (HDIS_SUF), which is executed after every ModX finishes its execution.

During the process of extension of the FreeRTOS we focused on preserving as much as possible the original FreeRTOS code. The first type of modifications were made to the interrupt priorities (the main priority in the system is now the time support for the HRT context, represented in our case by a dedicated timer). The relation among priorities in FreeRTOS have been preserved. Another modification was the *vTaskStartScheduler()*, which no longer calls indirectly the *prvStartFirstTask()*. Instead, the *prvStartFirstTask()* is called from PREHSCD module as described above.

The RAM memory is divided in two. The first half is used by the HARETICK extension and its ModXs and the second half is used by the FreeRTOS and its tasks.

## VI. Analysis and Evaluation

To analyze the functionality and performance of the new developed system, we have used a hardware platform based on the EFM32-G8900-STK, running the proposed extended operating system and a logic analyzer LA1032.

As a case study, we have chosen two ModXs, with the following relation between their periods: $T_2 = 2T_1$. We have also chosen two real-time tasks, with different periods first with priority $P_1$ *(Task1)* and the other with priority $P_2$, where $P_1$ represents a higher priority than $P_2$.

Using the logic analyzer, the functionality of the system has been tested and visualized. Fig. 4 presents an example of task preemption. It can be seen how the *ModX* $M_1$ preempts *Task 2* (at time $t_1$ as marked on the figure), and how *Task 2* is also preempted by *Task 1* (at time $t_2$), due to the higher priority of the latter task. *Task 1* is preempted by *ModX* $M_2$ at time $t_3$.

In Fig. 5 we can see another important difference between the ModXs and the tasks which are scheduled and executed within FreeRTOS, namely the perfect periodicity of the ModXs. While in the execution of the regular tasks, a significant jitter can be easily noted, the execution of the ModXs is perfectly periodical: the dispatch frequency of tasks is constant, with a value of 40 Hz, as can been seen by the auto measurements in the lower left corner. On the other hand, the dispatch frequency of the regular tasks varies (e.g. 39.94 Hz and 40.06 Hz). The value of the jitter increases with the frequency (at a task execution frequency of 100 Hz we have differences of up to 2 Hz).

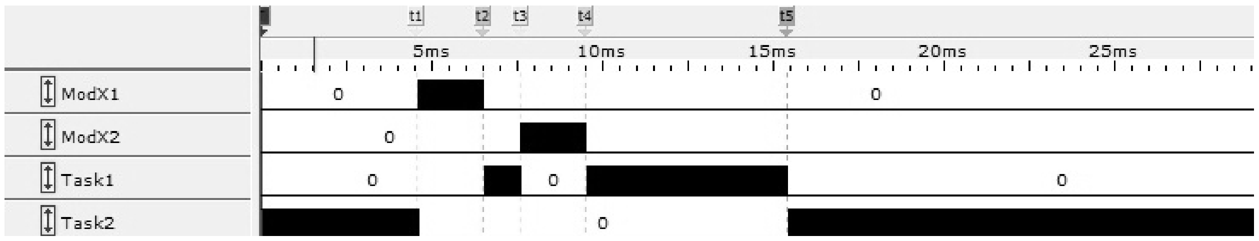
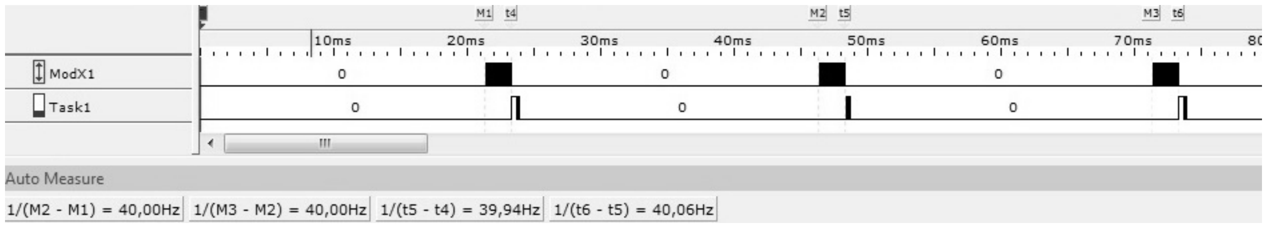


Fig. 4. Example of Task Preemption



1/(M2 - M1) = 40,00Hz | 1/(M3 - M2) = 40,00Hz | 1/(t5 - t4) = 39,94Hz | 1/(t6 - t5) = 40,06Hz

Fig. 5. Example of Jitter in Task Execution

## VII. Conclusions

In this article we propose a relatively easy method to implement a hard real-time execution extension to the FreeRTOS, which is a popular open source real-time operating system. This hybrid system has both the advantages of FreeRTOS, e.g. ease of use, small size, dynamic task creation and scheduling capabilities, task prioritization, interrupt service, etc., as well as the advantages of a static non-preemptive hard real-time scheduling and execution support, needed by many real life applications with critical impact on the humans or the environment.

The proposed HRT execution environment extension can improve the functionality of the control systems by offering support for hard real time task execution with no jitter and by scheduling and executing tasks in a totally deterministic manner.

In our implementation, we have done minimal modifications to the FreeRTOS kernel, while also preserving the original API. We have implemented, tested and studied our system on a hardware platform, the EFM32-G8900-STK.

Our future plan is to further extend the system, by including new modules (e.g. power management modules).

## Acknowledgment

This work was partially supported by the strategic grant POSDRU/159/1.5/S/137070 (2014) of the Ministry of National Education, Romania, co-financed by the European Social Fund – Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013.

## References

[1] M. V. Micea, G. N. Carstoiu, L. Ungurean, D. Chiciudean, V.-I. Cretu, and V. Groza, "PARSECS: A predictable data communication system for smart sensors and hard real-time applications," *Instrumentation and Measurement, IEEE Transactions on*, vol. 59, no. 11, pp. 2968-2981, 2010.

[2] F. M. Proctor and W. P. Shackleford, "Real-time operating system timing jitter and its impact on motor control," *in Intelligent Systems and Advanced Manufacturing,* 2001, pp. 10-16.

[3] R. Dobrin, G. Fohler, and P. Puschner, "Translating Off-line Schedules into Task Attributes for Fixed Priority Scheduling," Proceedings of the 22nd IEEE Real-Time Systems Symposium, 2001.

[4] C. Tianzhou, H. Wei, X. Bin, and Y. Like, "A Real-Time Scheduling Algorithm for Embedded Systems with Various Resource Requirements," in International Workshop on Networking, Architecture and Storages, Shenyang, 2006.

[5] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," *in Real-Time Systems Symposium*, 2005. RTSS 2005. 26th IEEE International, 2005, pp. 10 pp.-398.

[6] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 257-269, 2005.

[7] R. Inam, J. Maki-Turja, M. Sjodin, S. M. Ashjaei, and S. Afshar, "Support for hierarchical scheduling in FreeRTOS," *in Emerging Technologies & Factory Automation (ETFA)*, 2011 IEEE 16th Conference on, 2011, pp. 1-10.

[8] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam, "Hard real-time support for hierarchical scheduling in FreeRTOS," i*n Proceedings of 7th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications* July 5th, 2011 in Porto, Portugal: in conjunction with *the23rd Euromicro Conference on Real-Time SystemsPortugal*, July 6-8, 201, 2011, pp. 51-60.

[9] R. Barry, "FreeRTOS Reference Manual," Real Time Engineers Ltd, no. 2011.

[10] Real Time Engineers Ltd. (2014). Available: http://www.freertos.org/

[11] R. Goyette, "An Analysis and Description of the Inner Workings of the FreeRTOS Kernel," Carleton University, no. 2007.

[12] M. V. Micea and V. I. Cretu, "Highly predictable execution support for critical applications with HARETICK kernel," *AEU - International Journal of Electronics and Communications*, vol. 59, no. 5, pp. 278-287, 2005.

[13] M. V. Micea, C. S. Stangaciu, V. Stangaciu, and V. I. Cretu, "Improving the efficiency of highly predictable wireless sensor platforms with hybrid scheduling," *in Robotic and Sensors Environments (ROSE), 2012 IEEE International Symposium on*, 2012, pp. 73-78.

[14] A. Cortex, "M3 Technical Reference Manual " June2007, no. pp. 187-206, 2007.