Comparative Performance Evaluation of CMSIS-RTOS

Douglas P. B. Renaux[1,2,3]
1 - Graduate School in Applied Computing (*PPGCA*)
2 - Laboratory of Embedded Systems Innovation and Technology (LIT-CITEC)
3 - Department of Electronic Engineering – (DAELN)
Federal University of Technology - Paraná (UTFPR) - Curitiba - PR, Brazil
Curitiba - Paraná
douglasrenaux@utfpr.edu.br

Abstract— **CMSIS-RTOS is an RTOS standard recently defined by ARM to improve portability among microcontroller applications. Compliance to this standard can be achieved by design (the case for new RTOSes) or by an adaptation layer on top of an existing RTOS.**

CMSIS-RTOS has received criticism to its performance; yet, there is no published data comparing its performance to other RTOS. A comparative performance evaluation is conducted against Commercial and FOSS RTOS, resulting in the first published quantitative performance evaluation of CMSIS-RTOS. Contrary to the criticism, the evaluated implementation of CMSIS-RTOS presented no performance penalty when compared to two other classes of RTOS: Commercial and FOSS.

Keywords: CMSIS, RTOS, performance evaluation, Cortex-M, ARM microcontrolller, embedded systems.

I. INTRODUCTION

CMSIS-RTOS is a standard published by ARM in 2012 [1] that defines an RTOS API. Although the standard aims at Cortex microcontrollers (CMSIS stands for Cortex Microcontroller Software Interface Standard) it has been argued [2] to be suitable for general small microcontroller architectures, i.e. for microcontrollers without an MMU and thus implementing a multithreading model of concurrency.

To accommodate an existing RTOS into this new standard an adaptation layer is required that could impact system's performance. This scenario has been used to justify criticism to CMSIS-RTOS [3]. However, to this date, no published data is available comparing CMSIS-RTOS to other RTOSes, indicating that the criticism lacks any quantitative data to support it.

The aim of the research described here is to conduct a quantitative performance evaluation of CMSIS-RTOS to two other classes of RTOS: Commercial and FOSS (Free and Open Source Software). The X Real-Time Kernel [4] as a representative of a commercial RTOS and a FOSS RTOS.

II. RTOS PERFORMANCE EVALUATION

Recent publications on performance evaluation of RTOS include the "Survey and Performance Evaluation" by Anh and Tan [5], in IEEE Micro 2009 and the Master Thesis of Boger (2013) [6]. Both used benchmarks based on the Rhealstone.

The Rhealstone benchmark for RTOS was proposed in 1990 [7] as a composition of six time measurements of basic services provided by an RTOS: task switching time, preemption time, interrupt latency, semaphore shuffling time, deadlock breaking time, and inter-task messaging latency. The benchmark result is the number of rhealstones/second calculated from inverse of the (possibly weighted) average of these six measurements. It is a simplistic benchmark that certainly can be improved but is has the benefits of concentrating on the most basic services and has been the basis for many RTOS performance evaluations.

Anh and Tan [5] surveyed the features of 15 RTOS (both commercial and FOSS). Four of these RTOS were selected for performance bechmarking on a given hardware platform (Renesas M16C/62P, with a 16-bit microcontroller at 24MHz). In some of the measurements reported, including task switching time, the difference in performance among the four RTOS was larger than ten times. Also noticeable is that the RTOS that performed best in one measurement, often performed worst in another.

The results presented by Anh and Tan leads us to question the lack of performance data available for comparing RTOS performance.

Boger [6] evaluated the FreeRTOS [8] on a dual-core Cortex-A9 running at 667 MHz. The benchmark code used was the Rhealstone. The results were not compared to any other processor or RTOS.

Aroca [9] measured the latency, jitter and worst case response time of PCs (Pentium II 400 MHz) when running Windows XP, Windows CE, QNX, uC/OS, Linux, and VxWorks. The focus was on generating interrupts at high rates to measure latency due to interrupts disabled by the kernel.

III. CMSIS-RTOS

CMSIS v1.0 [10] was defined by ARM in 2008 to provide a standard device driver API to embedded software developers. CMSIS version 1 objectives were:

- Standard access by middleware libraries (USB stack, TCP/IP stack, file system, ...) to the integrated peripherals of Cortex microcontrollers [11].
- A vendor-independent HAL that would allow a standard way of accessing the peripheral hardware improving portability

among Cortex microcontrollers of different silicon vendors [10].

In 2012 [12], CMSIS version 3.0 (Fig. 1) was published adding CMSIS-RTOS and CMSIS-SVD. The first is a standardized RTOS API while the second is a standardized XML description of the registers of integrated peripherals.

The current release of CMSIS is version 4, dated Feb 28, 2014. It includes CMSIS-Driver and CMSIS-Pack. The first is a standard API for the device driver functions to be used by middleware components. The second is a description of a software component delivery mechanism.

ARM´s market share in the 32-bit microcontroller market puts ARM in a leadership position to define such standards. Market acceptability is likely to turn CMSIS into a *de-facto* standard.

The thrust behind CMSIS-RTOS is [11]:

- to stimulate the development of middleware that is now built on top of standardized HAL and RTOS API´s;
- to reduce the learning curve for embedded software developers of concurrent applications;
- to improve the portability of concurrent embedded software applications.

CMSIS also includes (Fig. 1): (a) CMSIS-DSP: a DSP library optimized for Cortex-M4; and (b) CMSIS-DAP: a standard debugging interface.

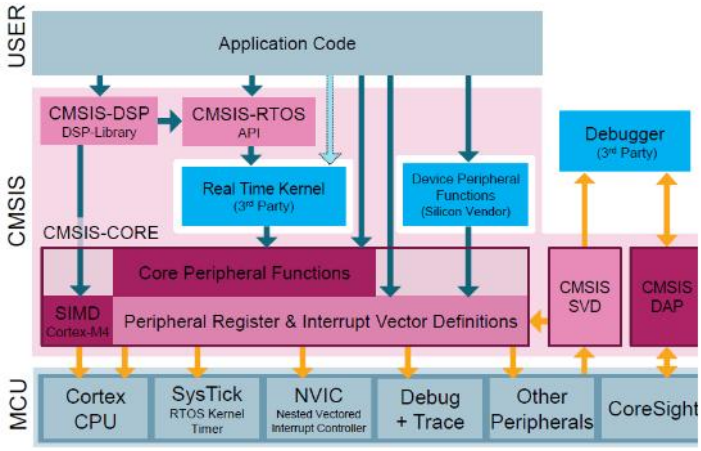

Fig 1 - CMSIS v 3 Structure (source ARM [11]).

IV. CMSIS-RTOS FUNCTIONALITY

This section summarizes the functionality defined in the CMSIS-RTOS API.

The state diagram of CMSIS-RTOS threads is presented in Fig. 2, while Fig. 3 presents an overview of the CMSIS-RTOS services. Functions marked with a $ sign are optional.

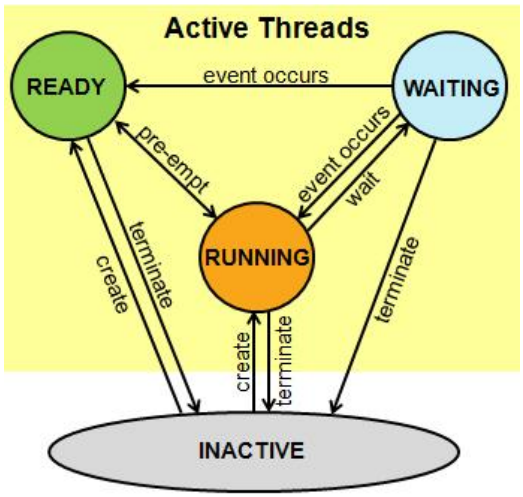

Fig 2 – Thread state diagram in CMSIS-RTOS (source ARM [13]).

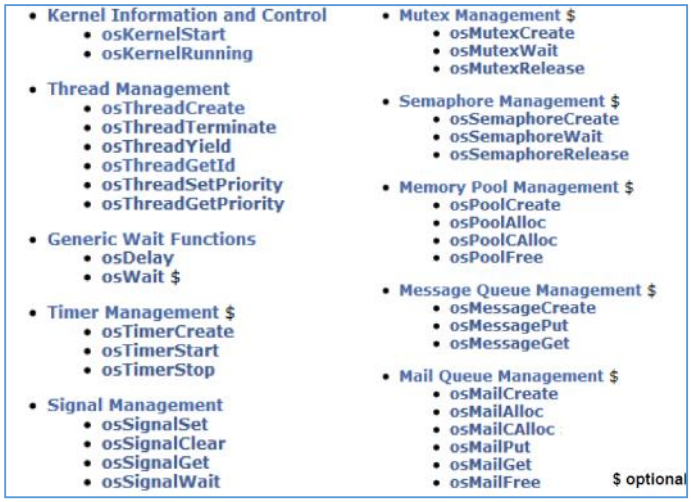

Fig 3 - Overview of CMSIS-RTOS API (source ARM [12]).

A. *Mutex (mutual exclusion semaphore)*

A Mutex is used to implement mutual exclusion among a set of taks, hence, it cannot be used by ISRs. The thread that locked (*osMutexWait*) the mutex has ownership and no other thread may release it (*osMutexRelease).* An *osMutexWait* may specify a timeout.

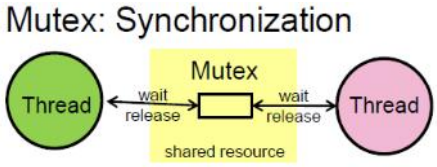

Fig 4 - CMSIS-RTOS Mutex (source ARM [13]).

A given implementation of Mutex may include priority inheritance, in this case, a thread that owns a mutex inherits the highest priority among the threads that are blocked on that mutex.

B. Semaphore

Semaphores have three significant differences to mutexes:

a) on creation they are assigned a value (1 or more);

b) can be released by ISRs;

c) have no notion of ownership or priority inheritance.

As an ISR cannot be blocked, a call to *osSemaphoreWait* returns an error code immediately if the current value of the semaphore is 0.

Semaphores (Fig. 5) may be used for synchronization between ISRs and threads: a thread blocks on a semaphore (*osSemaphoreWait*) and is released by an ISR (*osSemaphoreRelease*). Semaphores may also be used to control the access to several shared resources of the same type, in this situation the semaphore initial value corresponds to the number of shared resources.

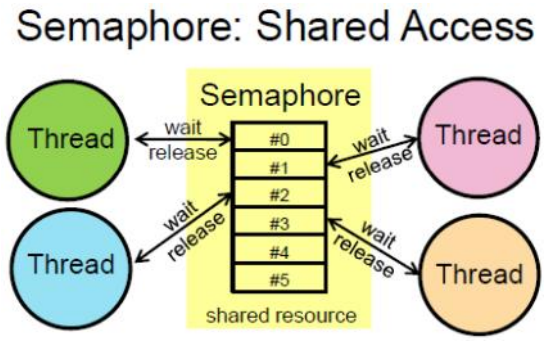

Fig 5 - CMSIS-RTOS Semaphores (source ARM [13]).

C. Message Queue

A Message Queue (Fig. 6) stores copies of integers (or pointers). The size of the queue is determined when it is created. Values are inserted with *osMessagePut* and removed with *osMessageGet.* Threads are blocked (with optional timeout) when attempting to insert to a queue that is full or to remove from a queue that is empty. ISRs have access to message queues but instead of blocking they get an error code.

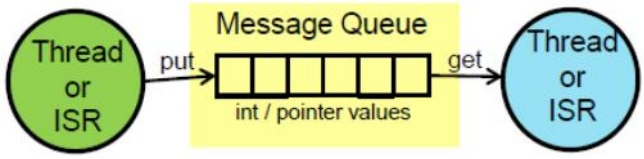

Fig 6 – Message Queue (source ARM)[13].

D. Mail Queue

A Mail Queue (Fig. 7) holds messages of any size that are actually stored in a memory pool. Hence, the queue holds only pointers to the memory blocks in the pool. A call to *osMailCreate* reserves memory for the pointer queue and for the memory blocks. *osMailAlloc* finds a free memory block that is then filled by the sending thread before sending (*osMailPut*). The receiver thread performs an *osMailGet*, accesses the memory block and then releases it (*osMailFree*). Threads may be blocked (with optional timeout) by *osMailAlloc*, *osMailPut* or *osMailGet.* ISRs cannot be blocked and instead receive error codes.

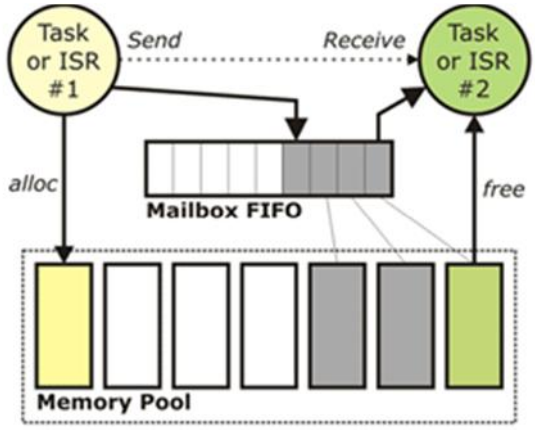

Fig 7 - CMSIS-RTOS Mail Queue (source ARM [14]).

E. Memory Pool

A Memory Pool is composed of a set of memory blocks of fixed size and managed by the RTOS. The size of the pool is defined when it is created (*osPoolCreate*) and the required memory is reserved, as well as the number of memory blocks and their sizes. Threads request memory blocks with *osPollAlloc* and return these blocks to the pool with *osPollFree*. There is no blocking, if no memory block is available osPollAlloc simply returns an error code.

F. Timming Services

Timming services include delays and Timers:

- *osDelay*(T) – suspends the execution of a thread for T milliseconds.
- *osWait*(TO) – suspends the execution of a thread until either a signal, message or mail is received. A timeout (TO) may be specified.
- *osTimerCreate* – creates a software timer including the definition of a callback function, its argument, and the type of timer (single-shot or periodic).
- *osTimerStar(T)* – starts the countdown with a value of T milliseconds.
- *osTimerStop* – stops the timer. It can be restarted later.

G. Signals

A signal is a binary flag that may be set/reset by other threads or ISRs, hence, it is the simplest form of communication/synchronization among threads and ISRs. A thread may own as many as 31 flags. A thread may suspend its execution (*osSignalWait*) waiting on a specific flag or on a set of them.

V. EVALUATION OF CMSIS-RTOS

The author has previously conducted a detailed study, reported in [15], comparing the functionality presented in the CMSIS-RTOS standard to POSIX and to three RTOS (VxWorks, FreeRTOS and X Real-Time Kernel). The conclusions of this study are summarized as:

- The functionality defined in CMSIS-RTOS is representative of its class of RTOS for microcontrollers without MMU. RTOSes in this class implement multi-threading.
- In a class providing more functionality are the POSIX-compliant RTOSes that are targeted at microprocessors with MMU. These RTOS support multiple processes.
- The wide acceptability of a standard for multi-threaded RTOS would bring several beneficial consequences, including: availability of interchangeable software components, particularly middleware; reduction of development cost and time, improving competitiveness due to developer's efficiency and reducing time-to-market; and reduction of the learning curve for concurrent real-time software development.
- Although the CMSIS-RTOS was defined for the ARM-Cortex architecture, it is applicable to other architectures. Hence, if widely adopted it would fill a gap due to a lack of standard for this class of RTOS.
- Combined with CMSIS-CORE that defines an API for access to peripherals (HAL), CMSIS becomes a complete Low Level Platform API providing a significant benefit to layers above.

VI. PERFORMANCE EVALUATIONS

Rather than creating a new RTOS benchmark, the benchmark used is the same as in Anh and Tan [5] that is based on the Rhealstone [7] [6]. It evaluates six performance aspects of an RTOS basic functionality: task switch time, semaphore passing time, inter-task message transfer time, memory allocation and release, task activation from an ISR, and interrupt latency.

The hardware platform used in this benchmark is a Cortex-M3 microcontroller, with 32K of Flash, 8K of RAM and a 72 MHz clock. The compiler used was the IAR EWARM v7.2. The CMSIS-RTOS RTK version v4.7 (made freely available by ARM) was compared to the X Real-Time Kernel v2.1 and a FOSS.

Time measurements were performed with an oscilloscope that monitors two I/O pins. The same code was used in all tests to set/reset the I/O pins; the execution time of each of these operations is 30ns. The only difference in the tested codes was to accommodate the differences in the APIs of the RTOS, as depicted in Table I below.

Among the relevant characteristics of an RTOS is its determinism [16]. Hence, performance evalutation of RTOS is not performed in the same way as General Purpose Operating Systems would be evaluated. By repeating the test, exactly the same result is obtained (within the resolution of the oscilloscope based measurement). The way that the Rhealstone benchmark is conceived, with simple tests with only one or two threads, collaborates to this determinism.

TABLE I. RTOS API USED IN EVALUATION

CMSIS-RTOS	X Real-Time Kernel	FOSS
osThreadYield	os.Yield	YIELD
osSemaphoreWait	sem.Wait	SemaphoreTake
osSemaphoreRelease	sem.Signal	SemaphoreGive
osMessagePut	os.Put	QueueSend
osMessageGet	os.Receive	QueueReceive
osPoolAlloc	os.Alloc	Malloc
osPoolFree	os.Free	Free
osSignalSet	os.ResumeThread	Resume
osSignalWait	os.SuspendThread	Suspend

A. *Task switching time*

Measures the time to transfer control of one task to another at the same priority level after a call to *Yield()*, hence, the measured time includes the execution time of the Yield( ) plus the context switch time.

Task1	Task2	
Pulse pin 1 Yield()	 Pulse pin 2 Yield()	Measure time from pulse 1 to pulse 2

B. *Semaphore passing time*

Two time measurements: the time to transfer control of one task to another due to a semaphore wait and back to the first task after a semaphore release and Yield. Both tasks at the same priority level. The first measurement includes the semaphore wait and the context switch. The second includes the semaphore release and the context switch.

Task1	Task2	
Pulse pin 1 SemWait Pulse pin 1	 Pulse pin 2 SemRelease Yield()	Measure time from pulse 1 to pulse 2 and then to next pulse 1.

C. *Inter-task message transfer time*

The receiver tasks has higher priority than the transmitter task. Measure execution time of Receive and context switch (due to empty message box) and time to send message and context switch (due to higher priority task receive a message).

Task1 - Rx (higher prio)	Task2 - Tx	
Pulse pin 1 Receive Pulse pin 1	 Pulse pin 2 Put	Measure time from pulse 1 to pulse 2 and then to next pulse 1.

D. Memory Allocation

A single task allocates and releases a memory block.

Task1	
Pulse pin 1 Alloc Pulse pin 2 Free Pulse pin 1	Measure time from pulse 1 to pulse 2 and then to next pulse 1.

E. Task activation from an ISR

An ISR wakes a suspended task. Measure the time from the resume command in the ISR to the start of execution of the waken task. An idle task is used to measure the time to execute Suspend.

Task1	Timer ISR	
request timer IRQ Pulse pin 1 Suspend Pulse pin 1 …	 Set pin 2 Resume Reset pin 2	Measure Suspend and CS (pulse 1 to start of idle task). Then measure pin2 pulse width and time from negative edge of pulse 2 to pulse 1.

F. Interrupt Latency

A task generates an interrupt request via an I/O pin. Measure the time from the request to the start of execution of the ISR. None of the three RTOS evaluated had any interference in this measurement, hence all measures were the same. The reason being that interrupts with a priority higher than the kernel's interrupt are never disabled.

Task1	ISR	
pulse IRQ	 Pulse pin 2 RETI	Measure time from IRQ to pulse 2.

VII. MEASUREMENTS

By conducting the tests described in the previous section for the three RTOSes, the following measurements were obtained:

TABLE II. MEASURED TIMES (US)

Test	RTK	X RT Kernel	FOSS
1 Yield+CS	3.92	3.64	2.1
2.1 Wait+CS	4.56	4.3	14.9
2.2 Release+Yield+CS	4.68	4.62	10.8
2 Total	9.24	8.92	25.7
3.1 Get+CS	5.48	7.3	17.5
3.2 Put+preemption	5.72	6.4	9.7
3 Total	11.2	13.7	27.2
4.1 Alloc	0.92	1.4	3.8
4.2 Free	1.12	0.96	3.1
4 Total	2.04	2.36	6.9
5.1 Suspend+CS	5.5	4.4	5.1
5.2 Resume	2.5	2.48	3.8
5.3 CS to task	5.6	2.5	1.9
5 Total	8.1	4.98	5.7
6 IRQ Latency	0.32	0.32	0.32
k Rhealstones*/s	172.3	176.8	88.3

CS = Context Switch
Rhealstones* = Rhealstones variant described in Section II

Analyzing the results above, one notices that the largest performance differences of the final result are 2:1. Also, the largest difference in individual measurements is 3.45:1. Hence, the results obtained here have a much lower variance than those reported by [5] (described in Section II).

Concerning the aim of this research, the evaluation of CMSIS-RTOS, the results indicate that there was no performance penalty when an RTOS is designed to follow the API defined in the standard. This was the case for RTK.

But then another question arises: would there be a performance penalty if an adaptation layer was used with an existing RTOS ? To evaluate this scenario, an adaptation layer was written for the X Real-Time Kernel. This adaptation layer is partly implemented by macros (*#define*) and partly implemented by functions. Since macros may perform straightforward conversions of one API to another and thus they can often be resolved at compile time, they are preferred since this poses no runtime penalty.

For the services listed in Table II, only osSignalSet and osSignalWait required implementation by functions that convert calls to osSignalSet and osSignalWait into calls to os.SuspendThread os.ResumeThread. Hence, only test 5 had

differences in execution time. The new three measurements of test 5 are reported in Table III below. These values were used to calculate the number of Rhealstones/s resulting in the value 176,300, which is 0.35% below the previous value of 176,800.

TABLE III. MEASURED TIMES (US) IN TEST 5 FOR X REAL-TIME KERNEL WITH ADAPTATION LAYER FOR CMSIS-RTOS

Test	X RT Kernel with adaptation layer
5.1 Suspend+CS	4.46
5.2 Resume	2.60
5.3 CS to task	2.5
5 Total	4.98
k Rhealstones*/s	176.3

The 0.35% overhead due to the adaptation layer for the X Real-Time Kernel cannot be taken as a general rule, as this value may vary depending on the characteristics of each specific RTOS. It shows, however, that CMSIS-RTOS compliance may be obtained with an adaptation layer at almost no performance cost.

VIII. CONCLUSION

By analyzing the results of tests described in the previous sections for the three evaluated RTOSes, the following conclusions can be drawn:

- Contrary to what was expected in paper [3], the CMSIS-RTOS standard does not impose a significant performance overhead. Not even when an adaptation layer is used.
- The complexity of an adaptation layer, and consequently its performance penalty, is strongly dependent on the differences among the API of CMSIS-RTOS and the API of the kernel in use. When the same type of services are available in both then the adaptation layer tends to be a replacement of function names and parameters that can be resolved at compile time with little to none performance overhead. However, if the type of services made available by the kernel is significantly different then the adaptation layer may impose a higher performance overhead.
- The analysis reported in [15] shows that CMSIS-RTOS defines the same sort of services that are available in the 4 other analyzed APIs.

For future work in this research line, many other RTOSes may have their performance evaluated. The measuring infra-strucuture is readily available in a lab and easily implemented in code. The structure of each test is well defined in Section VI of this paper and can be repeated by anyone interested in repeating these tests or comparing other RTOSes to the ones presented here.

REFERENCES

[1] ARM. (2012, February 27). *ARM Extends CMSIS with RTOS API and System View Description* [Press release]. Available at: http://www.arm.com/about/newsroom/arm-extends-cmsis-with-rtos-api-and-system-view-description.php

[2] D. Renaux and F. Pottker, "Applicability of the CMSIS-RTOS Standard to the Internet of Things", 10th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems - SEUS/ISORC 2014, Reno-USA, June 2014.

[3] R. Barry, "Who wins when Cortex-M adds RTOS?". March 2012, www.embedded.com.

[4] eSysTech , "X Real-Time Kernel, Programmer´s Manual v2.1", 2008

[5] T. N. B. Anh and S. L. Tan, "Survey and performance evaluation of realtime operating systems (RTOS) for small microcontrollers," IEEE Micro, Aug 2009. DOI: 10.1109/MM.2009.56.

[6] T. J. Boger, "Rhealstone benchmarking of FreeRTOS and the Xilinx Zynq extensible processing platform", Master Thesis, Temple University, May 2013.

[7] R. P. Kar, "Implementing the Rhealstone Real-Time Benchmark" Dr. Dobb´s Journal, April 01, 1990. Available at: http://www.drdobbs.com/cpp/implementing-the-rhealstone-real-time-be/184408332

[8] R. Barry, "FreeRTOS Reference Manual". 2012. freertos.org.

[9] R.V. Aroca and G Cauri, "A real time operating systems (RTOS) comparison", S Carlos–SP–Brasil. Workshop de Sistemas Operacionais (Operating Systems)(WSO'2009).

[10] ARM. (2008, November 12). *ARM Introduces Software Interface Standard for Cortex Processor-Based Microcontroller* [Press release]. Available at: http://www.arm.com/about/newsroom/23722.php

[11] M. Gouda, "CMSIS-RTOS an API interface standard for real-time operating systems," ARM Technology Symposia, China, 2012.

[12] ARM, "CMSIS version 3 adds CMSIS-RTOS – an API interface standard for 3rd party real-time operating systems". STMicroelectronics MCU Technical Seminars, April 2012. Available at: http://www.anglia.com/events/st_seminars/10/pdf/CMSIS_V3_MS_Apr_2012.pdf

[13] CMSIS-RTOS API: Generic RTOS interface for Cortex-M processor-based devices. Available at: http://www.keil.com/pack/doc/arm/cmsis/cmsis/documentation/RTOS/html/index.html. April, 2013.

[14] ARM, "Mail Queue of CMSIS-RTOS". Available at: http://www.arm.com/products/tools/software-tools/mdk-arm/middleware-libraries/rtx-real-time-operating-system.php

[15] D. Renaux, F. Pöttker, "Avaliação do Padrão CMSIS-RTOS para uso em Sistemas Embarcados" – SBESC 2013 – Nov, 2013.

[16] D. Kalinsky, "Basic Concepts of Real-Time Operating Systems". Linuxdevices.com. Nov 2003.