Porting SLOTH System to FreeRTOS running on ARM Cortex-M3

S. Pinto, J. Pereira, D. Oliveira, F. Alves, E. Qaralleh, M. Ekpanyapong, J. Cabral and A. Tavares
Centro Algoritmi - University of Minho
{sandro.pinto, jorge.pereira, daniel.oliveira, f.alves, jorge.cabral, adriano.tavares}@algoritmi.uminho.pt
qaralleh@psut.edu.jo, mongkol@ait.ac.th

***Abstract*—Traditionally, operating system (OSes) suffers from a bifid priority space dictated by the co-existence of threads managed by kernel scheduler and asynchronous interrupt handlers scheduled by hardware. On real-time systems, where reliability and determinism plays a critical role, this approach presents a noteworthy lack, as any interrupt handler can interrupt an execution thread, regardless of its priority.**

This paper presents the implementation of an unified priority space approach (SLOTH), handling each thread as an interrupt. A light-weight version of FreeRTOS was internally redesigned, to replace the software scheduler by a hardware one, which exploits a Commercial Off-The-Shelf (COTS) hardware interrupt controller, provided by ARM Cortex-M3. The results showed that our implementation solves the priority inversion problem, and simultaneously improves the system performance, reduces the memory footprint and simplifies maintainability.

***Index Terms*—Unified Priority Space, Threads as Interrupts, FreeRTOS, Real-time Systems, ARM Cortex-M3, NVIC.**

I. Introduction

One of the most important core task of an operating system kernel is to manage the execution control flow of a computing system [1]. This encompasses handling of synchronous execution threads and asynchronous interrupt service routines (ISRs), simultaneously. Threads are managed by software, which means that they are scheduled and dispatched by the OS software scheduler. ISRs, in contrast, are managed by hardware, i.e. they are scheduled and dispatched by mechanisms provided with the hardware interrupt controller, usually triggered by hardware events associated with peripheral devices.

This division into thread priorities and interrupt priorities forms a dual priority space, with ISRs having a higher privilege of execution. This issue induces a well-identified problem on scientific community, designed as rate-monotonic priority inversion [2], [3]. On embedded real-time systems, where time and determinism plays a critical role, this means that the execution of real-time threads can be delayed by interrupts that have semantically lower priority. On applications fitting safety-critical systems, this problem can cause catastrophic consequences to the human life.

To overcome the aforementioned problem, a light-weight version of the FreeRTOS targeting the ARM Cortex-M3 architecture was refactored to support the SLOTH [4] concept. The thread-related application programming interface (API) were redesigned based on the programmable interrupt subsystem services, while still keep them syntactically intact to avoid the porting effort for legacy applications. A suspending feature is also presented to overcome the run-to-completion drawback of implementing threads as pure interrupts. The strategy has a significant impact on the entire system, presenting advantages on performance, memory footprint and code management.

II. Related Work

The problem of a bifid priority space is not new as the first proposal dates from 1995 [5]. Since there, other solutions were proposed, and guided by different concepts. Kleyman and Eykholt [5] presented an effective way to unify the execution flow of an operating system by handling interrupts as threads. However, although they justify the mapping of interrupts into threads using a low overhead technique, their approach leads interrupts to the same level of performance overhead and indeterminism as software threads. The same methodology was followed by Lohamn et al. [6] as a configurable property of the CiAO embedded operating system.

Luis Leyva-del-Foyo et al. [7] address the priority inversion problem by integrating interrupts and tasks handling on a low level interrupt handling component of an operating system, and, therefore, eliminating the overhead of a dual priority space. Their approach is portable to several hardware platforms, adaptable for various operating system implementations and well suitable for real-time systems.

SLOTH [4] and SLEEPY SLOTH [8] are the only projects targeting commodity hardware (e.g., Infineon TriCore hardware interrupt subsystem) by first treating threads as pure interrupt handlers [4] on a redesigned OSEK-OS specication. However, SLOTH presented a significant limitation as it dictates a control flow based on run-to-completion semantics, which later was solved by abolishing the distinction between threads and ISRs [8]. Thus, threads could be dispatched as efficiently as interrupts, and interrupts could be scheduled as flexibly as threads.

A fruitful unification of the priority space requires hardware support that is not strictly provided by commodity processors. However, some existent commercial processors, like ARM Cortex-M Series and Infineon TriCore, integrate programmable hardware interrupt controllers which properly exploited will tackle such an issue.

III. Design

This section briefly presents an overview of SLOOTH and SLEPPY SLOTH approaches, explaining how we plan to use

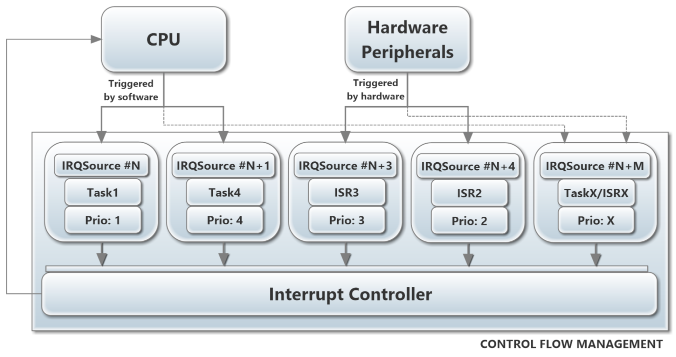

Fig. 1: ISR Vector Table

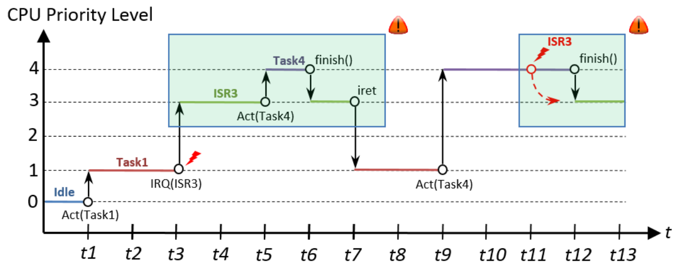

Fig. 2: Unified Priority Space: Example of an Execution Flow

them as the backbone of our implementation.

A. *Threads as Interrupts*

The central design idea of this work is having threads run as interrupt handlers, unifying a priority space in order to solve the priority inversion problem. An overview of our design is presented in Figure 1. As it can be seen, tasks and ISRs are represented by abstract interrupt sources, that will be configured with appropriate priorities. For tasks, the interrupt source request is triggered synchronously by software (CPU), while for ISRs it is triggered asynchronously by hardware (peripherals). This way, the scheduling is done completely in hardware, and so, the interrupt controller subsystem is responsible to always dispatch the interrupt source (task/ISR) with the highest priority. Therefore, if ISR2 is running and the interrupt source associated to Task1 is triggered, the interrupt controller stops the execution of ISR2 and starts running Task1.

In the example depicted in Figure 2 an unified execution flow of a system composed by two tasks (Task1 and Task4) and one ISR (ISR3) is presented. Task1, Task4 and ISR3 are associated to interrupt sources with priorities 1, 4 and 3, respectively. Analysing the picture, Task1 is activated at t_1, and runs until an asynchronous hardware event happens at t_3. At this time, since this event is linked to an ISR that has higher priority than Task1, the latter is interrupted and ISR3 is dispatched. During the execution of ISR3, another high priority task (Task4) is activated at t_5. On a system with a bifid priority space, the typical behaviour leads ISR3 to execute until the end, and only after return-from-interrupt instruction (`IRET`) Task4 would be dispatched. However, by unifying the priority space since Task4 has higher priority than ISR3, the task will be dispatched at t_5, and only when it terminates at t_6 ISR3 will be recovered. Task1 is re-scheduled again after the end of ISR3 at t_7. At t_9 the task with the highest priority is activated again, which means that Task1 is interrupted and Task4 dispatched. During the execution of Task4 an asynchronous event triggers, once again, ISR3. But, since Task4 has the highest priority, it is not interrupted, and only when it finishes (t_{12}) ISR3 will be dispatched.

So, the analysis of the previous example clearly shows that the unified priority space approach solves completely the priority inversion problem of a system with a dual priority space. However, treating threads as pure interrupts has an inherent limitation: they can not be suspended, which means that it only supports run-to-completion control flow. To tackle suck drawback, the next subsection describes how to add a suspending feature, which extends in some way the usual preemptive mechanism.

B. *Suspending Feature*

The previous subsection ended identifying the main SLOTH's drawback, that was later fixed with SLEEPY SLOTH [8]: tasks which are implemented as interrupts can not be suspended. This subsection will briefly present how to tackle the above problem, by basically modeling a task as consisting of three parts: prologue, body and epilogue [9].

1) Prologue: The prologue is executed wherever a high priority task is scheduled by the interrupt controller. It extends the standard behaviour of the hardware interrupt controller (i.e., saves automatically some registers of the CPU context) by saving the remaining context of the current task, and restoring the context of the new task. The only exception wherein the prologue is not executed is when the scheduling point is between two pure interrupts.

2) Epilogue: The epilogue is executed wherever a task is suspended or finished. If the task was suspended, it saves the task's context and then restores the state of the new dispatched. Otherwise the task was finished and it only restores the context of the new task. The only exception wherein the epilogue is not executed is when the scheduling point is between two pure interrupts.

Figure 3 shows the execution flow of a more complex application. The application is composed by three ISRs (ISR0, ISR1 and ISR5) and two tasks (Task2 and Task4). ISR0, ISR1 and ISR5 have priority 0, 1 and 5 respectively. Task2 has priority 2 and Task4 has priority 4. At t_1, an asynchronous event trigger ISR1, preempting ISR0. Since they are both pure interrupts the interrupt controller will perform the context switch automatically. Subsequently, ISR1 activates Task2 (t_2) which is prefixed by its prologue. When Task2 is created its own stack is initialized with the right registers values that should be load when the task is dispatched. This loading is done during the Task2 prologue. During the execution of Task2 another high priority task (Task4) is activated and then dispatched by the interrupt controller (t_4). Its associated

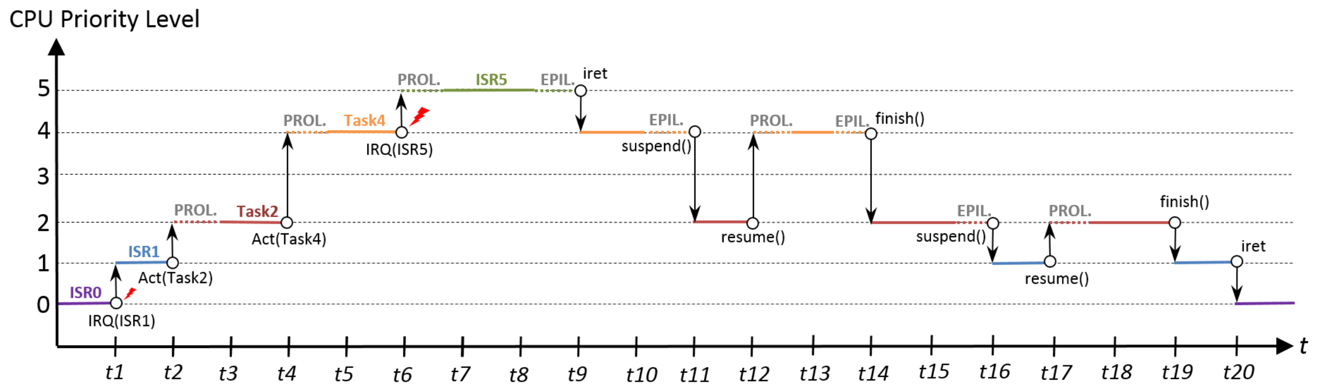

Fig. 3: Suspend Feature: Example of an Execution Flow

prologue will save the state of the preempted Task2 and restore the context of Task4. At t_6 Task4 is interrupted by a higher priority interrupt (ISR5) and at t_9 it will resume the execution. Therefore its state, when interrupted, must be saved (ISR5 prologue) and after must be restored (ISR4 epilogue). Task4 will suspend itself, and at that time its state must be saved in its own stack (i.e., a field of the task's TCB). When the suspension takes place, besides saving the task state is also restored the state of the next task. Task2 then resumes Task4, activating its prologue once again (t_{12}). This task finishes at t_{14} and at that point a epilogue is performed to restore the state of next task to be dispatched (Task2). ISR1 will be re-scheduled again when Task2 performs a suspension at t_{16}. This suspension triggers an epilogue that will save the context of Task2. Then, ISR1 resumes again Task2, that will execute its prologue to restore its context. When Task2 finishes (t_{19}), ISR1 is re-schedule normally as well as ISR0 at t_{20}.

C. *Synchronization*

Synchronization mechanisms are used to protect critical sections, which means that they avoid multiple tasks or threads from access shared resources simultaneously. Mutual exclusion (mutex objects) is one of the most used approaches to implement it. Commodity hardware usually provides a special CPU register that defines the minimum priority for interrupt processing. Basically when this register is set to a given priority value, it prevents the activation of all interrupts with the same or lower priority level. So, ensure the exclusivity of a resource means raising the current minimum CPU priority to the highest priority level of all tasks that share the resource. By contrast, release it means re-setting the register to the original value.

D. *Requirements on the Hardware Interrupt System*

The implementation of the unified priority space model is only feasible if the hardware interrupt controller fulfills some specific requirements: (i) the hardware interrupt controller must be programmable and provide different configurable priority levels, which allows changing the priority level of an interrupt source; (ii) the interrupt subsystem shall support manual triggering of interrupts as well as by software, allowing threads activation synchronously; (iii) the number of interrupt sources should be enough to cover all the threads and interrupt handlers desired for the system, or some kind of multiplexing/sharing of interrupt sources should be supported by the runtime system.

IV. Implementation

A. *Operating System and Architecture Overview*

1) FreeRTOS: FreeRTOS [10], [11] is a real-time operating system (RTOS) designed to be deployed on embedded systems with scarce resources. It is characterized by a very simple and small kernel core, written mostly in C, presenting a software architecture divided into two main layers: the "hardware independent" and the "portable" layer. The former is responsible for performing processor independent functions and is maintained intact for all architectures, while the second implements some architecture-specific routines (e.g. context-switching).

Among the extensive list of existing RTOSs, FreeRTOS was chosen because: (i) is open-source, which allows an internal redesign, crucial to this work's purpose; (ii) the kernel core is simple and small, which allows to perform the necessary changes without a huge engineering effort; (iii) is widely used and a low-end embedded market leading RTOS, due to the large number of supported architectures (ported to thirty-four different platforms until June 2013 [10]).

2) ARM Cortex-M3: The ARM Cortex-M3 processor was the first of the Cortex generation release by ARM in 2004 targeting a wide range of applications in the embedded systems domain. The interrupt subsystem present in the Cortex-M3 platform - NVIC (Nested Vectored Interrupt Controller) - fulfils the requirements described in Section III-D. By default this interrupt controller provides 16 sources of exception and 1-240 sources of external interrupts (IRQ).

B. *Threads as Interrupts*

Running threads as interrupts handlers is very straightforward on the Cortex-M3 platform. As sketched is section III, this involves the refactoring of the system vector table. This table is presented in the startup code for the target platform. Looking closely to the vector table layout, only entries above the 17th entry point (i.e., those dedicated to external interrupt sources) are available for the model implementation, as the first 16 entry points are dedicated to system exceptions handlers. However, even among those available entry points some of the last ones should be freed for special devices management.

Listing 1 presents the patches applied over the vector table. As it can be seen, each task will be linked to an interrupt source, based on its priority. The assignment is done during the task creation.

```
__Vectors:
        /* System Exceptions */
        DCD __initial_sp ; Top of Stack
        DCD Reset_Handler ; Reset Handler
        DCD NMI_Handler ; NMI Handler
        ... /* External Interrupts */
        DCD  FLASH_IRQHandler ; Flash
        ...
        DCD Task0_IRQHandler ; Task w/ prio. 0
        DCD Task1_IRQHandler ; Task w/ prio. 1
        ...
```

Listing 1: Vector Table Refactoring

Setting up a task IRQ channel during its creation encompasses the initialization of a structure that contains the configuration information of it (task priority, IRQ channel number and status). The variable `TaskIRQ_s` will be filled with the task attributes, and used to initialize the NVIC (Listing 2). At last, the interrupt source pending bit (linked to task) is set. If the configured task has higher priority than the one currently owning the executing path, the scheduler will dispatch it. However, the scheduler must be previously set by enabling all interrupts through the `PRIMASK` register.

```
void xPortAssignTask(unsigned portBASE_TYPE uxPriority)
{
        /* Task IRQ structure declaration */
        NVIC_InitTypeDef TaskIRQ_s;
        /* Priority assignment */
        TaskIRQ_s.NVIC_IRQChannelPreemptionPriority = uxPriority;
        /* Task assignment to the respective IRQ */
        TaskIRQ_s.NVIC_IRQChannel = LAST_IRQ + uxPriority;
        /* Task IRQ status */
        TaskIRQ_s.NVIC_IRQChannelCmd = ENABLE;
        /* NVIC initialization according to NVIC_InitStruct */
        NVIC_Init(&TaskIRQ_s);
        /* Sets the Task IRQ pending bit to 1 */
        NVIC_SetIRQChannelPendingBit(LAST_IRQ + uxPriority);
}
```

Listing 2: Task Assignment to an IRQ Channel

C. Suspending Feature

Supporting the named suspending feature in SLOTH-based environment is very challenging. The approach as described in the above design section led to the creation of a prologue and epilogue code segment, executed wherever a task is dispatched by the hardware. With these two segments it is possible to suspend tasks, preserving properly their extended contexts and restoring them later when related tasks are resumed.

When a task is dispatched by the interrupt controller the handler associated with that interrupt source is executed. Each handler defined in the start-up file (Listing 3) is responsible for identify the IRQ channel associated with the triggered task (through `R2` register) and only then call the `xPortSwitchContext` routine to execute the respective task prologue and epilogue.

```
Task0_IRQHandler:
                PROC
                IMPORT xPortSwitchContext
                MOV R2,#0
                B xPortSwitchContext
```

Listing 3: IRQ Handler Refactoring

The prologue code segment (Listing 4) is executed when a higher priority task is scheduled, starting by saving the context of the previous task in its own stack, pointed by `pxTopOfStack` in the TCB structure. Then, this pointer should be updated with the new top of the stack location. Furthermore, the system stack is recycled by removing the contents of certain registers automatically saved by the interrupt controller. That set of registers will be saved in the task's stack to prevent the possibility of implosion of the main stack in the presence of multiple chained tasks. Then the context of the dispatched task is restored, ending with a jump to the task function body.

```
Prologue:
        ...
        LDR  R0,  =pxCurrentTask
        LDR  R1,  [R0]
        ...
        MOV  R3,  #4
        MUL  R0,  R1,R3
        /* R1 = pxTopOfStack */
        LDR  R3,  =pxCurrentTCB
        LDR  R3,  [R3,R0]
        LDR  R1,  [R3]
        /* Saves the extended task context */
        STMIA R1!, {R4−R10}
        ...
```

Listing 4: Partial Prologue Code Segment

In turn, the epilogue code segment (Listing 5), is executed when a task is suspended or finished, starting by checking if the current task was suspended or terminated. In the former case, it is implicit that a `vTaskSuspend()` API call happened. In this case the tasks TCB is updated by placing the 'S' character on the task status. Then, the context is saved in its own TCB stack field, which occurs immediately after getting the address to its top. The next step will be executed regardless if the task was suspended or finished. This encompasses restoring the context of a low priority task, which involves the replacement of registers values as well as the main or execution stack.

```
Epilogue:
        ...
        LDR  R0,  =pxCurrentTask
        LDR  R3,  [R0]
        MOV  R1,  #4
        MUL  R2,  R3,R1
        /*R11 = pxTopOfStack*/
        LDR  R3,  =pxCurrentTCB
        LDR  R3,  [R3,R2]
LDR R11, [R3]
/*Restores extended task */
LDMIA R11!, {R4−R10} /*R4−R10*/
...
```

Listing 5: Partial Epilogue Code Segment

Finally, to resume a suspended task it is necessary to identify its priority, which is crucial to select the correct IRQ channel. After this identification, it will be verified if the task is indeed in the suspended state, and if so the resume is performed. This encompasses only setting the pending bit of the corresponding task channel.

D. Synchronization

The acquisition of a mutual exclusion object for protection of shared resources will be performed taking into account which task is currently running and desires to acquire it. Based on the priority of the task that wants to acquire the mutex, it will be verified which other tasks share the same resource. The resource shared between tasks must be statically identified during each task creation. As previously discussed in section III-C, to guarantee the exclusive use of that resource the current minimum CPU priority should be raised to the highest level among all tasks that share it. The `BASEPRI` register presented in the NVIC is used to perform this priority ceiling protocol, where interrupt handlers with a lower or equal priority to the task that acquires the mutex will not be dispatched.

E. Limitations

Four main limitations can be identified on this implementation. The first one is related to the limited number of tasks: the refactoring of the vector table is restricted to the available number of interrupt sources provided by the interrupt controller. However, since this is a hardware limitation, it can be overcome choosing a microcontroller version which features a greater number of interrupt sources, or supporting interrupt sources multiplexing/sharing mechanism through the runtime system. Second, the suspending flexibility offered to the tasks can only be performed over the task that is currently executing. This means that is not possible to suspend another task inside the body of the task that has presently the control of the CPU. Third, the priority level of each task must be atomic and so, tasks (which are assigned to a dedicated interrupt source based on their priority) should not have the same priority. Finally, the implemented synchronization mechanism is not compatible with FreeRTOS original API, and thus it will be overcame in a near future.

V. Evaluation

Since our approach aims at making use of hardware features to implement some OS task management functionalities, this leads to a positive impact in system performance, memory footprint and maintainability. To corroborate this predictions, we tested two different versions of the FreeRTOS on a ET-STM32F103 evaluation board with an ARM Cortex-M3 running at 72MHz. Basically, a native light-weight (with a minimal core kernel APIs support) version of FreeRTOS were compared against the redesigned implementation. To assess the aforementioned metrics some software tools, like Keil μVision4 toolkit [12] and Understand [13], were used.

A. Performance

In order to assess the performance gain achieved by our implementation, we have performed several microbenchmarks on both OS versions. The selected scenarios encompass the modified thread-related system calls, which include:

1) Create a task with high priority to trigger its dispatching: execution time from the point before `xTaskCreate()` to the first instruction of the created/activated task;
2) Create a task with low priority which does not trigger its dispatching: execution time of `xTaskCreate()` system service;
3) Suspend the running task, which leads necessarily to the dispatching of another task: execution time from the point before `vTaskSuspend()` to the point after the task was dispatched;
4) Resume a task of higher priority, which triggers its dispatching: execution time from the point before `vTaskResume()` to the first instruction of the resumed task;
5) Resume a task of lower priority, which does not trigger its dispatching: execution time of `vTaskResume()` system service;
6) Acquire a resource: execution time of `xSemaphoreTake()` system service;
7) Release a resource with the dispatch of another task: execution time from the point before `xSemaphoreGive()` to the first instruction of the dispathced task;
8) Release a resource without the dispatch of another task: execution time of `xSemaphoreGive()` system service;

The performance results are depicted in Table I. On the first and second microbenchmark, the time spent on memory allocation was not taken into account for both implementations. As it can be seen, compared to the standard light-weight version of FreeRTOS, our implementation presents better results in all scenarios. Concretely, it reduces the execution time between 3.62% (*6 - Acquire a resource*) and 82.76% (*5 - Resume a task of lower priority*), which corresponds to a speedup between 1.04 and 5.80, respectively.

B. Memory Footprint and Code Managment

The memory footprint results reflects the code memory in bytes of both compiled kernel images (ROM size of compiled hex file). To measure the size of each kernel image, the MDK ARM toolchain (Keil uVision4) was used without any optimization. Table II presents the achieved results. It shows that the modified FreeRTOS has lower memory footprint than the original one. On the other hand, the boot memory is higher than the original, because this section suffered some significant changes at the software architecture-level to improve the

TABLE I: Performance Results

	Execution Time (clock cycles)							
	1)	**2)**	**3)**	**4)**	**5)**	**6)**	**7)**	**8)**
FreeRTOS (standard)	610	730	326	374	232	166	702	188
FreeRTOS (modified)	496	420	142	164	40	160	426	40
Comparison	**-18.69%**	**-42.47%**	**-56.45%**	**-56.15%**	**-82.76%**	**-3.62%**	**-39.32%**	**-78.73%**

TABLE II: Memory Footprint and Code Managment Results

	Code Memory (bytes)			Lines of Code (LOC)		
	kernel	**boot**	**Total**	**kernel**	**boot**	**Total**
FreeRTOS (standard)	3124	372	3496	1162	121	1283
FreeRTOS (modified)	952	460	1412	602	319	921
Comparison	**-69.52%**	**+23.56%**	**-59.61%**	**-48.19%**	**+163.63%**	**-28.20%**

unified priority space approach on the FreeRTOS. However, since the required boot memory code is considerable smaller than the kernel one, the total code memory was decreased 59.61%.

The code management results show the number of lines of code (LOC), excluding comments and blank lines, required to implement both minimalists versions of FreeRTOS. To assess code management metrics, the Understand software by Scientic Toolworks was used. As it can be seen in Table II, the number of LOC for the kernel was decreased 48.19%. On the other hand, on the boot code the number of LOC was increased 163.63%. However, once again, since the boot code is considerable smaller than the kernel one, the total number of LOC was decreased 28.29%. This means that with our implementation the maintainability effort was also slightly reduced.

VI. Conclusion and Future Work

This paper described how standard modern interrupt controllers can be used to implement a single system execution flow, unifying threads and interrupts. Instead of a pure software-based scheduler our approach uses the hardware interrupt subsystem to implement a unified priority space, avoiding the rate-monotonic priority inversion problem and ensuring a high level of reliability and determinism. To implement the followed approach (SLOTH), we have internally redesign a light-weight version of FreeRTOS targeting the ARM Cortex-M3 architecture. The evaluation results showed convincing improvements at multiple levels: on execution performance; on memory footprint; and in code management. For example, with respect to system performance our implementation outperforms the standard implementation, reducing the execution time by a factor of 4% to 83%.

In summary, the presented solution suppresses the existent distinction between threads and ISRs on FreeRTOS: threads can now be treated as interrupts, and interrupts can be suspended like threads. Developers can forget the artificial differences of each execution flow, and choose priorities homogeneously based only on the requirements and constraints of the applications.

Proposed as future work is the implementation on a full version of FreeRTOS, giving complete support for the remain APIs. Inclusively, the mutual exclusion feature will be also redesigned to be fully compatible with legacy applications. Additionally, considering that multiprocessing is becoming an emergent trend on todays embedded market, future research will be also focused on investigating ways to port this approach to multi-core platforms. Pandaboard [14] and ZedBoard [15] will be used as a reference boards since they both feature a dual-core ARM Cortex-A9 with a similar interrupt controller.

VII. Acknowledgements

This work is supported by FEDER through COMPETE and national funds through FCT Foundation for Science and Technology in the framework of the project FCOMP-01-0124-FEDER-022674.

References

[1] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. John Wiley and Sons, 2009.

[2] L. Leyva-del Foyo, P. Mejia-Alvarez, and D. Niz, “Predictable interrupt management for real time kernels over conventional PC hardware,” *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 14–23, 2006.

[3] F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat, and D. Lohmann, “Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system,” *International conference on Compilers, architecture, and synthesis for embedded systems - CASES*, pp. 167–174, 2009.

[4] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, “Sloth: Threads as interrupts,” *30th IEEE Real-Time Systems Symposium - RTSS*, pp. 204–213, 2009.

[5] S. Kleiman and J. Eykholt, “Interrupts as threads,” *ACM SIGOPS Operating Systems Review*, pp. 21–26, 1995.

[6] D. Lohmann and J. Streicher, “Interrupt synchronization in the CiAO operating system,” *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, 2007.

[7] L. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, “Predictable Interrupt Scheduling with Low Overhead for Real-Time Kernels,” *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications - RTCSA*, pp. 385–394, 2006.

[8] W. Hofer, D. Lohmann, and W. Schröder-Preikschat, “Sleepy Sloth: Threads as Interrupts as Threads,” *32nd IEEE Real-Time Systems Symposium - RTSS*, pp. 67–77, 2011.

[9] R. Müller, “Implementation of an Interrupt-Driven OSEK Operating System Kernel on an ARM Cortex-M3 Microcontroller,” 2011.

[10] “FreeRTOS homepage.” [Online]. Available: http://www.freertos.org/

[11] R. Barry, *Using the FreeRTOS Real Time Kernel*, 1st ed., 2010.

[12] “uVision4 User’s Guide.” [Online]. Available: http://www.keil.com/support/man/docs/uv4/

[13] “Understand Source Code Analysis and Metrics.” [Online]. Available: http://www.scitools.com/

[14] “PandaBoard homepage.” [Online]. Available: http://wwww.pandaboard.org/

[15] “ZedBoard homepage.” [Online]. Available: http://www.zedboard.org/