

# Lightweight IO Virtualization On MPU Enabled Microcontrollers

Francesco Paci  
University of Bologna  
Bologna, Italy  
f.paci@unibo.it

Davide Brunelli  
University of Trento  
Trento, Italy  
davide.brunelli@unitn.it

Luca Benini  
University of Bologna  
Bologna, Italy  
ETHZ  
Zürich, Switzerland  
luca.benini@unibo.it  
luca.benini@iis.ee.ethz.ch

## ABSTRACT

In the era of the Internet of Things (IoT), millions of devices and embedded platforms based on low-cost and limited resources microcontroller units (MCUs) will be used in continuous operation. Even if over-the-air firmware update is today a common feature, many applications might require not to reboot or to support hardware resource sharing. In such a context *stop*, *update* and *reboot* the platform is unpractical and dynamic loading of new user code is required. This in turn requires mechanisms to protect the MCU hardware resources and the continuously executing system tasks from uncontrolled perturbation caused by new user code being dynamically loaded. In this paper, we present a framework which provides a lightweight virtualization of the IO and platform peripherals and permits the dynamic loading of new user code. The aim of this work is to support critical isolation features typical of virtualization-ready CPUs on low-cost low-power microcontrollers with no MMU (Memory Management Unit), IOMMU or dedicated instruction extensions. Our approach only leverages the Memory Protection Unit (MPU), which is generally available in all ARM Cortex-M3 and Cortex-M4 microcontrollers. Experimental evaluations demonstrate not only the feasibility, but also a satisfactory level of performance of the proposed framework in terms of memory requirements and overhead.

## Keywords

Virtualization, MPU, Microcontrollers, Dynamic Linking

## 1. INTRODUCTION

Many IoT applications envision the deployment of large numbers of microcontroller-based smart sensor nodes in hard-to-reach locations [1, 2]. This not only means that they are supposed to operate unattended, without direct maintenance, and likely with the same battery for many years; but also that the software could be updated (if necessary) only remotely; and in many scenarios it is expected that bug fixes, functional improvements, reconfiguration will be necessary over the time. Clearly the old fashion style for reprogramming embedded systems based on stopping the device, updating the firmware and restart, become unfeasible when millions of low cost devices are spread all over and are expected to be updated with new functionality many times over their life span.

In addition, IoT devices are expected to provide more and

more services on the same hardware. The possibility to have multiple “application tasks” running on the same hardware, possibly coming from different developers, introduces the challenge of protecting the resources from misuses and to guarantee adequate computing bandwidth to all the tasks or to prevent over-allocation of resources that would lead to collective starvation.

In such a scenario, well-known virtualization technologies already used in computing servers, gateways and other high-end computing systems become fundamental also in low-end and ultra-low cost programmable end-nodes for IoT. First, the virtualization of the hardware resources becomes necessary to execute securely multi-function software and different applications with well-controlled interference. Then, the capability to remotely download new parts of code, to link dynamically the binary and to execute runtime within the main application, avoids on-site maintenance or periodic down-time and reboot.

These two requirements highlight the importance of IO virtualization and dynamic linking on low-cost, low-power microcontrollers. However, if this technology is well known and available in operating systems for high-end embedded systems (e.g. Linux on ARM Cortex-A microprocessors), providing mechanisms for dynamic linking in low-resource microcontroller based embedded platforms, such as ARM Cortex-M class, is still a challenge, and only few and limited solutions have been proposed so far.

The dynamic linking proposed in this work executes on the FreeRTOS [3] operating system and it is based on the framework presented in [4] which addressed the capability to download new functions remotely. The main contributions of this paper are:

- a Lightweight Virtualization layer which separates the user space from the kernel space, therefore now all the physical peripherals are *virtualized*. Such a virtualization is a protection towards system tamper and ready to be extended to protect possible conflicts on the hardware assignments;
- our solution is integrated with FreeRTOS and exploits standard communication API provided by the operating system. Thus, it can be easily ported also on other microcontrollers.
- we provide the capability to have the *dynamic linking* of new user code, managing its life cycle as well as its orderly shutdown in case of attempted violations of protected memory regions;

The paper is organized as follows. Section 2 gives an overview of works related to our contribution, Section 3 describes in depth the framework architecture and provides all technical details of this solution, Section 4 details our per-

formance and memory footprint, while Section 5 concludes the paper.

## 2. RELATED WORKS

Virtualization support for embedded systems based on high-end CPUs, such as the ARM Cortex-A series, has been extensively explored in the academic literature and has reached industrial maturity [5]. This class of devices exploits the hardware acceleration extensions to provide hardware abstraction and protection to critical resources. Recent Cortex-A CPUs feature native virtualization support like MMU and IOMMU address translation, interrupt virtualization, TrustZones [6, 7], etc. Cortex-M MCUs do not come with any of those hardware extensions. Furthermore, available memory and computational resources are much more limited. Our work and the related works surveyed below deal with Cortex-M3 and Cortex-M4 class of devices, where virtualization is not a mature technology and several compromises with respect to full hardware-supported virtualization have to be made.

### *Abstract Virtual Machines and Interpreters*

One of the most common approaches for virtualization on MCUs is based on interpreter-based virtual machines, which have been originally conceived with the main purpose of creating high-level easy-to-use languages and run-times at a higher abstraction level than the traditional C language. Python [8, 9], Java [10, 11], Javascript [12], Lua [13] are all lightweight multi-paradigm scripting languages employed in Virtual Machines for embedded systems. Their main benefit is the cross-platform support. They are interpreted by a native virtual machine loaded on the microcontroller, thus they introduce high overhead in term of latency of access to the resources in comparison to virtualization layers written in native code, but they are designed for easy software application development and to meet the increasing demand of fast run time customization, without the need of complex or dedicated compiling toolchains. Such a kind of virtualization, usually, is focused on improving portability, extensibility, ease-of-use in development and protection but lacks of performance, multiple user level accesses and low-level hardware control. Only the exposed high level resources can be leveraged by the user.

Bogliolo *et al.* [14] presented *Virtual Sense*, a sensor node which executes java-compatible virtual machine called *Darjeeling VM* [11] on top of Contiki OS [15]. This work is close to ours in the emphasis on supporting resource allocation and protection for multiple independent user tasks on the MCU. However this solution, besides the overhead introduced by the interpreter, is oriented to share only network stack between Darjeeling VM tasks, while our work is general to all peripherals.

### *Just In Time/Ahead of Time Compilation*

A well-explored approach to reduce the run-time overhead of VM interpreters is Just in Time or Ahead of Time Compilation. Micropython [8] developers, for example, introduced in their platform the concept of *decorator* to emit ARM native *opcode* and to use native C types, but not all native C types are supported and the implementation of this optimization is platform dependent. A solution can be to extend with C wrapped functions called from python, but there are drawbacks: marshaling and unmarshaling of data is very expensive in terms of computational resources and with this solution the programmer loses the low level abstraction. In comparison, using our solution, the developer implements C functions which will be executed in user level tasks. In general these approaches require a higher memory footprint to host the just-in-time or ahead-of-time compile

process and do not achieve the performance of native code execution. Furthermore, they are difficult to use in contexts where real-time constraints cannot tolerate the jitter introduced by on-line compilation.

### *Native Implementations*

Native virtualization is the closest to hardware and extremely desirable for resource and performance-limited devices. This technique usually relies on the use of MPU that is the only hardware unit available for security in low-end systems.

Bhatti *et al.* [3] presented a complete operating system designed for WSN (Wireless Sensor Network) and optimized to simultaneous execution of threads which can be loaded dynamically. Their work relies on *Mantis OS*, a custom operating system. They target Atmel and their solution is highly customized, thus is not general, while our work relies on FreeRTOS therefore it is highly extensible and portable to other platforms. Moreover they do not explicitly address security and protection.

To the best of our knowledge we find only one very recent work that addresses the problem in a broad and general sense, similarly to our solution. Andersen *et al.* [16] presented an embedded platform that relies on TinyOS. They use a mixed paradigm that permits to have Lua VM but the computational intensive part of code can be written in native C. To address security they use a task receiving event based system calls, to separate kernel to user space tasks. Our work differentiates from the latter by permitting to have both system call support and Event based peripheral virtualization. Moreover Andersen *et al.* do not provide any information on the performance of the event based system call paradigm.

## 3. SOFTWARE ARCHITECTURE

In this section we present all the software layers in our runtime system, focusing on software protection. Figure 1 shows the layer stacking from three viewpoints, first from a hardware point of view, then from address space access, divided in IO and Flash/RAM. We divided core hardware from peripherals in two different stacks to underline that the OS can expose system calls to access to the core hardware resources, while the Virtual IO Layer is designed to access to the peripherals. The last stack shows that the access to memories is direct for privileged tasks, while the access from user mode tasks is strictly regulated by MPU. Two different kinds of tasks are defined: privileged tasks and user mode tasks, which will be discussed in next section.

Another important layer depicted in Figure 1 is FreeRTOS [17], a well known Real Time Operating System for a broad range of Embedded Systems from 8 to 32bit, including low power and ultra-low power MCUs. We implemented our framework on an STM32F4 based platform, and even if some details in the following description are related to this specific microcontroller, our framework can be easily extended to be platform independent.

In Sections 3.1 and 3.2 we focus on the first and third stack, namely on exploiting the MPU and providing Safety Extensions, while in Section 3.4 we discuss the second stack.

### 3.1 Real Time OS

The main reason for using FreeRTOS is its versatility: many MCUs are supported and the code is maintained and upgraded often by Real Time Engineers Ltd. Moreover it is modular and there are some extensions available (e.g. MPU extension), which can be added to the core release. The open source nature makes possible to extend it. It has more over a small memory footprint and sources consist of a small number of files. The scheduler supports real-time operation,

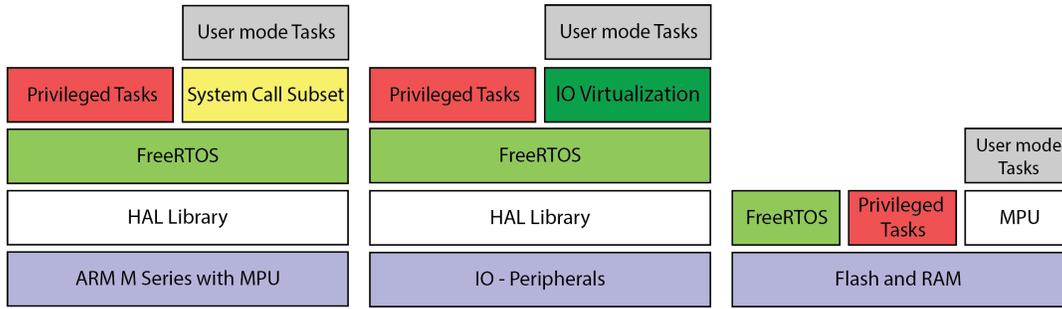


Figure 1: Hardware, IO and Memories layers.

both time-triggered by a configurable system tick and with support for priorities with preemption.

### 3.2 FreeRTOS Additions

To strengthen the security of the system, the FreeRTOS MPU module has been integrated to enable the usage of the Memory Protection Unit implemented on the microcontroller and to activate the two levels of privileges for the tasks execution. However, the original module is an experimental release, because of some limitations that we addressed in our work:

1. It does not have a proper way to access system resources. It provides only one system call. This system call raises the privileges of the caller from user mode to privileged, executes the call and then sets the privileges back to user space. This behavior has sufficient protection in an environment where a single developer wants to keep separation between tasks, i.e. the case where a single company develops all the firmware. While in the case we want to give to a third-party user the capability to develop his own code, the knowledge of the existence of this backdoor is really dangerous for protection.
2. The exploitation of the MPU is static. The protection sections of the MPU are not reconfigurable at run-time by privileged tasks.
3. The task termination is not correctly handled. When a user mode task raises an MPU trap the exception ends the system execution. Hence it would be extremely easy to create denial of service attacks.

In next sub-sections we describe our proposed solutions to these limitations.

#### 3.2.1 MPU Extension

As already stated, this module permits to grant different access privileges on a task-by-task basis. For each task the MPU settings are stored in the task descriptor, called Task Control Block (TCB) in FreeRTOS. When a task is created, it can be started with one out of two levels of privileges:

1. Privileged Tasks (similar to Linux Kernel Mode execution). The task executes with permission granted to access all system resources, memories and peripherals.
2. User mode Tasks (similar to Linux Kernel User Mode, also called unprivileged tasks). The task is executed in more restrictive environment and has access only to a limited subset of memory and IO addresses.

STM32 Cortex-M4 has eight configurable MPU regions. When activated the protection policy is white-list based: to access to a specific position in the address space, the task should have a grant on one MPU region. The privileges on an MPU region can be: NONE, READONLY AND READ-WRITE. In FreeRTOS these MPU regions are configured as

follows:

- Region 0 *FLASH protection*  
Protects whole FLASH providing read-only privileges to both privileged and user mode tasks.
- Region 1 *OS FLASH protection*  
Protects from accesses by user mode tasks to the OS code in FLASH
- Region 2 *OS RAM access*  
Provides permission to privileged task to access the OS structures stored in RAM
- Region 3 *Peripheral access*  
Used to enable or disable the access to peripherals.
- Region 4 *Task Stack access*  
Used to give access to tasks own stack.
- Region 5-7 *Not used*  
These three regions are not used by FreeRTOS MPU module, thus they are open to developer purposes.

In Table 1, we show a list of MPU configurations used in our solution. As the reader can notice, there is no access to peripherals granted to user mode tasks. This access can be only allowed through the IO Virtualization Architecture.

One of the main constraints of the FreeRTOS MPU module is that it permits to configure the last regions (from 5 to 7) at compile time only. Thus, we implemented a specific software module to reconfigure these regions at run-time for each task. This is done for the following reasons:

1. Access to Virtual IO Layer (deeply explained in Sub-section 3.4) can be restricted by an MPU Region and must be asked by a task. This makes the Virtual IO Layer aware about the number of tasks that are using it.
2. Moreover access to heap or other memory regions can be granted at run-time. This is open to several future applications.

#### 3.2.2 Safety Extensions

As previously stated, the single system call paradigm is not safe. The *raise\_privilege* system call has been removed and replaced by more specific system calls for required cases. For example to grant access to FreeRTOS Queues and Direct Task Notification, the following list of system calls are added:

- MPU\_xTaskGenericNotify: Direct task notification Notify function
- MPU\_QueueReceive: Receive a message on a queue
- MPU\_xGetCurrentTaskHandle: Get the current task handle
- IO\_Layer\_REGISTER: Registration to Virtual IO Layer

**Table 1: Default MPU region setting in FreeRTOS**

Privileged Perm.	User Mode Perm.	Region Desc.
READ ONLY	READ ONLY	all Flash Protection
READ ONLY	NONE	OS Code Segment in FLASH
READ WRITE	NONE	OS RAM Protection
READ WRITE	NONE	Peripherals
READ WRITE	READ WRITE	Task Stack
NOT USED	NOT USED	User configurable
NOT USED	NOT USED	User configurable
NOT USED	NOT USED	User configurable

### 3.2.3 Graceful Task Termination - Killer Task

FreeRTOS does not provide task termination. Thus, when an unprivileged task tries to access a memory address without permission a *trap* is generated from the MPU and the OS ends its execution in an endless loop. This is not acceptable if we want to keep all other tasks and OS in execution. The desired behavior is that the task causing the *trap*, is aborted while the system continues its execution. Thus a memory trap handler and a specific task, called *Killer Task*, have been created to manage the termination of the task that raised the *trap*. The *Killer Task* is a privileged task created at boot time and it is in *sleep* state, when the MCU is in normal usage. When a *trap* occurs the task is activated. The *Killer Task* gets the task handles of the task that generated the *trap* and removes it from the scheduler execution queue. Then it resumes the scheduler execution and goes back into sleep, waiting for the next *trap*.

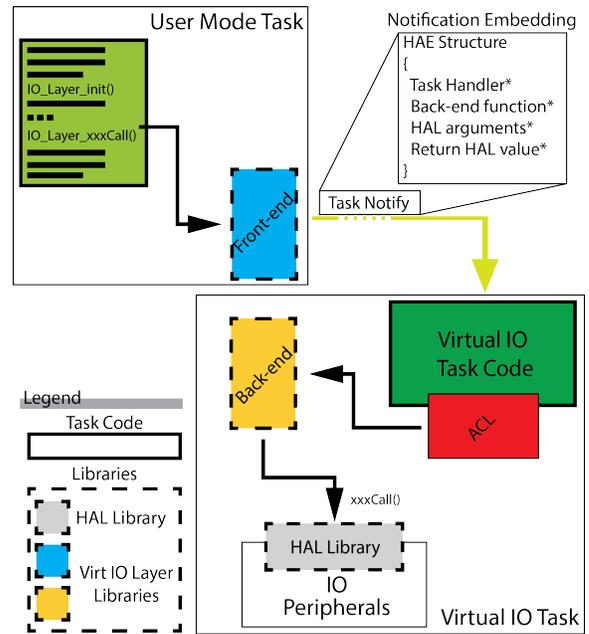
## 3.3 Software Protection

In a software protection perspective, the MPU enables the OS to keep the control on the user mode tasks. Thus, with the MPU all user mode tasks cannot tamper the whole system. On the other hand, if we want to enable a third party software developer to access only a small subset of peripherals, a fine grain control on address space must be implemented. Usually in a MCU all peripherals addresses are grouped from a starting to an ending address. However, if we want to provide fine grain access to a subset of them, three free MPU regions are really limiting. Moreover there are other two limitations: one is that the minimum area for an MPU regions is usually 32 Bytes (i.e. on STM32f4) that is usually larger than the register pool of a peripheral. The other is that register set of several peripherals consists of both control registers, and reading/writing ports, at subsequent memory positions. Thus it is not possible to grant the access to a read-only register and denying the permission to a contiguous configuration register.

## 3.4 IO Virtualization Architecture

The Virtual IO Layer architecture consists of two main parts: (1) a task called *Virtual IO Task* that invokes the callbacks to access to IO and to peripherals through the hardware abstraction layer (HAL); (2) a library named *Virtual IO Library* that contains the front-end calls forwarded transparently to the *Virtual IO Task* and the back-end calls invoked by the *Virtual IO Task* to access the HAL Library. The *Virtual IO Task* is a FreeRTOS task that handles all the IO calls from the user mode tasks to the peripherals. As shown in Figure 2: this task acts as a task-in-the-middle that receives all calls from user mode tasks that attempt to access to the peripherals, checks the permissions and forwards the requests through the HAL library.

### 3.4.1 Virtual IO Library



**Figure 2: IO Virtualization High Level Architecture**

The library consists of two subsets: a front-end functions subset and the relative back-end functions subset.

When a user mode task wants to access peripherals, it needs to subscribe to the Virtual IO Layer, using one special front-end function. Registration is required for two purposes:

1. The user mode task must have read only access to the *Virtual IO task* handle. This is needed to use the OS event notifications to notify the *Virtual IO task*. Therefore, one of the MPU regions of the task must be run-time configured to read-only access to *Virtual IO task* handle.
2. User mode tasks are not authorized to use interrupt handlers, because interrupt handler code is executed in privileged mode. We used a queue system to communicate from interrupt handlers to user mode tasks. Hence the registration routine creates a new queue and saves the queue handler in a structure. This will be used afterwards if the task will request access to one peripheral in interrupt mode.

The registration is done through a system call that was previously mentioned in subsection 3.2.2, hidden by a front-end call. The system call is needed to configure an MPU region described in the former purpose. The registration procedure works as follows: (1) The user mode task invokes the *IO\_Layer\_init()* routine, which through (2) the *IO\_Layer\_REGISTER* system call (3) sets an MPU region of the caller task to access to *Virtual IO Task* descriptor in read-only mode. This is needed to send Notifications. Then the framework create and initializes a System Queue (4) for using the DMA (the procedure is described in Back End Subset subsection). Before returning, if the procedure was successful, the task is added to the list of Virtual IO subscribed tasks.

### Front End Subset

The Front End subset is intended to be called from the user mode tasks. These calls have the same signature of the original HAL library calls, beside the function name, which is extended with a prefix to make the programmer aware that

is using the Virtual IO Layer and, obviously, to avoid a name space conflict. Thus for each HAL library function that we want to expose to the third party developer a function must be written. Each function declares a structure that contains:

1. The user-mode task handler.
2. A pointer to the relative back-end function to be called by the *Virtual IO Task*
3. A pointer for each original HAL Library function argument.
4. If the original HAL function returns a non-void value, a field to store it.

We refer to this structure with the name HAL Library Argument Embedding Structure (HAE Structure). Then HAE structure is instantiated in the function, on the stack, and all structure's fields are assigned with their values. A notification is sent to the Virtual IO Layer Task with a pointer to this structure. At the end optionally the HAL Library return value is returned if the function is non-void. A recap of the embedding of this function is shown in right top corner of Figure 2.

### Back End Subset

The back end (or call back functions) is the part of the library meant to be called by the *Virtual IO Task*. For each front-end function, there is one corresponding back-end one that takes in input a single argument, a void pointer. Its body contains a declaration of the HAE structure written for the corresponding front-end function. The void pointer is then cast in this structure, arguments are then used to call the original HAL function. When the HAL Library call ends up, the return argument is written in the structure, that still resides in the user-mode stack then control return to the *Virtual IO Task*. Then the *Virtual IO Task* suspends its execution waiting for the next call.

This architecture has two advantages: (1) the ease of use, the programmer does not need to learn a new interface to use the HAL. (2) All front-end calls and back-end calls have the same format, so they can be written by a programmer or generated by an automatic tool.

To Handle DMA asynchronous calls and to get notified when a DMA transfer is completed, we use the Queue returned when the user mode task subscribes the Virtual IO Layer. For security it is important that all the interrupt service routines (ISR) are implemented by the system. Moreover inside each service routine there is a Queue Send operation used to notify the task that wants to use the DMA that the routine is called. To correctly notify the corresponding user mode queue a reference table is used. This reference table is set by the back-end, when the user mode task invokes one of the DMA HAL Library functions.

### 3.4.2 Virtual IO Task

The *Virtual IO Task* is a privileged task that handles the communication from user mode tasks to peripherals. It starts when the Virtual IO layer is initialized, typically at system boot time. The communication is handled via *Direct Task Notification*. When started this task hangs in suspended state waiting for a call from one of the user mode registered tasks through the front-end.

The priority of this task is higher than all user mode tasks. Thus, when the notification is thrown from the front-end, the user mode task waits that the *Virtual IO task* ends its execution. Therefore even if task notifications are asynchronous, the call to HAL Library is blocking because in FreeRTOS the preemption of the scheduler is priority based.

The body of this task, besides the *Task Notify Wait*, consists of an Access Control List (ACL), shown in Figure 2, that checks that the callee HAL Library function can be in-

voked by the caller. The pointer to HAE Structure is cast to a generic structure common for all HAE Structures (we always know that the first two fields are fixed: the user-mode task handler and the pointer to the call-back function), then the ACL permission check occurs. If the checking passed, the back-end function is invoked.

## 3.5 Dynamic Linking

The dynamic linking permits a task to be added to the run time tasks without rebooting the system. We implemented dynamic linking to demonstrate the usage of the whole system. Therefore, we implemented a privileged task in charge of dynamic linking other user mode tasks. Tasks are cross-compiled and unresolved dependencies to system library calls are run-time linked and the task is added to scheduler execution queue. The library in charge of dynamic linking usermode tasks is derived from the work of [4]. In Flash memory we reserved a section to store these new tasks binaries to be linked and then added to FreeRTOS scheduler ready task list.

## 4. EXPERIMENTAL RESULTS

In this section we present results in term of performance and memory footprint. All tests were conducted on an *STM32F411RE* NUCLEO-64 Board [18]. This is a platform by ST Microelectronics, it embeds an ARM<sup>®</sup> 32-bit Cortex<sup>®</sup>-M4 CPU running up to 100 MHz with FPU and MPU. It features 512 KB of Flash memory and 128 KB of RAM memory. In our software setup we use the new driver for accessing hardware peripherals provided by ST called Hardware Abstraction Layer Driver (HAL Driver) [19].

We identified two main use cases, i.e. ways to access peripherals in a Microcontroller unit, that must be considered separately:

1. Atomic Action:

This is the case in which we call a HAL Driver routine each time we access a peripheral. In other words, we just want to access once an IO address or we may access it in a loop, but call does not involve peripheral transfer after it. An example of this behavior is when we want to configure or read a GPIO PIN, or write something on the UART.

2. Continuous Action (or Tunneling Action):

In this second case we consider all the peripheral usages that involve the use of DMA. For example when we want to set Analog to Digital converter and read it at regular intervals by the DMA.

### 4.1 Virtual IO Layer Timing

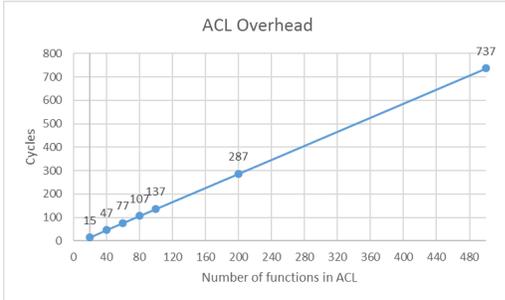
The time of accessing a peripheral using the Virtual IO Layer is reported in Table 2. The first row gives the cycles to get the task handle through a system call. The *MPU\_xTaskGenericNotify()* is the direct task notification system call. The third row reports the cycles required to notify the *Virtual IO Task*. The last row gives the number of cycles to return control, after the HAL Driver call back to the User mode task. The cycles measurement has been done with the *DWT\_CYCCNT* hardware cycle count register of the Cortex-M4 MCU.

It is worth mentioning that with this paradigm, continuous mode operations pay the overhead just once, when the setup of the peripheral or IO is performed. Thus when the DMA is working the only overhead is the queue used to synchronize the ISR with the user mode task.

The cycles overhead to check if the function that the user mode task wants to use is permitted by the ACL grows linearly with the number of checks that occurs. In Table are

Virtualization Step	VIO (Cycles)
getTaskHandle	97
MPU_xTaskGenericNotify	47
xTaskNotify + CS	490
Notify wait + CS back	293
<b>TOTAL</b>	<b>926</b>

**Table 2: Timing overhead of accessing the IO using the Virtual IO Layer in Cycles**



**Figure 3: Overhead of the control in the ACL.**

we report the the overhead As expected the number of cycles are proportional to the number of function addresses to verify.

## 4.2 Virtual IO Layer Memory Footprint

The overhead in terms of memory footprint is described in Table 3. We show the code size of the library and of the Virtual IO Task separately, in case the compiler is invoked with the flag for performance (-O3) or space (-OS) optimization. The Size of the Virtual IO Library is measured with an average size of 50 functions (front end + back end). As we can notice from the results, the memory footprint is minimal, even if it scales with the number of driver functions that we want to provide to the user mode tasks.

Optimization	VIO Task	VIO Library
-O3	592 B	2876 B
-OS	464 B	2314 B

**Table 3: Virtualization Layer code size**

As a concluding note, it is important to stress the fact that the runtime of tasks when not interacting with the IOs is exactly the same as native FreeRTOS tasks, with no performance overhead for memory protection as the MPU is completely transparent from the performance viewpoint. This is very similar to what happens in virtual machine execution for high-end cores, and in sharp contrast with interpreted virtual machines or even JIT-based systems.

## 5. CONCLUSIONS

In this paper we have presented a virtualization layer for low-cost microcontrollers which creates a separation between kernel mode and user mode and protects the hardware resources from misuses when concurrent tasks or function are written by different developers. Moreover we demonstrated the effectiveness of a mechanism capable to execute new runtime code, without the need of system reboot. We have focused on small size of the framework and on lower overhead, because targeted for low-cost and limited computing capabilities microcontrollers such as the ones designed for IoT and WSN. Experimental results demonstrate that the overhead is limited and time delay is negligible considering

the typical application scenarios. Future works will extend dynamic linking toward multiple upload channels and will implement different permission policies to peripherals from different user mode tasks.

## 6. ACKNOWLEDGMENTS

This work was partially supported by EU Project EuroCPS H2020-ICT-2014 under Grant 644090 and in collaboration with STMicroelectronics.

## 7. REFERENCES

- [1] Lu Tan *et al.*. Future internet: The internet of things. In *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, volume 5, pages V5-376-V5-380, Aug 2010.
- [2] Ala Al-Fuqaha *et al.*. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347-2376, Fourthquarter 2015.
- [3] Shah Bhatti *et al.*. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563-579, August 2005.
- [4] Simon Holmbacka *et al.* Lightweight framework for runtime updating of c-based software in embedded systems. In *Presented as part of the 5th Workshop on Hot Topics in Software Upgrades*, Berkeley, CA, 2013. USENIX.
- [5] ARM Virtualization Extension. <https://www.arm.com/>.
- [6] *ARM Security Technology - Building a Secure System using TrustZone Technology*. Whitepaper, April 2009.
- [7] T. Alves and D. Felton. *Trustzone: Integrated hardware and software security-enabling trusted computing in embedded systems*. White paper, arm, july 2004.
- [8] Micropython website. <http://micropython.org/>.
- [9] PyMite. <https://wiki.python.org/moin/PyMite>.
- [10] Oracle Java ME Embedded. <http://www.oracle.com/>.
- [11] Niels Brouwers *et al.*. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys '09*, pages 169-182, New York, NY, USA, 2009. ACM.
- [12] Espruino Javascript Interpreter. <http://www.espruino.com/>.
- [13] Embedded power driven by Lua. <http://www.eluaproject.net/>.
- [14] Alessandro Bogliolo *et al.*. Virtalsense: A java-based open platform for ultra-low-power wireless sensor nodes. *International Journal of Distributed Sensor Networks*, 2012, 2012.
- [15] Contiki: The Open Source OS for the Internet of Things. <http://www.contiki-os.org/>.
- [16] Michael P. Andersen *et al.*. System design for a synergistic, low power mote/ble embedded platform. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks, IPSN '16*, pages 17:1-17:12, Piscataway, NJ, USA, 2016. IEEE Press.
- [17] FreeRTOS website. <http://www.freertos.org/>.
- [18] ST Microelectronics Nucleo Boards. <http://www.st.com/>.
- [19] ST Microelectronics Hardware Abstraction Layer Driver. <http://www.st.com/>.