2.2

Report on the identified OS-independent abstractions for micro-ROS

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	2.2
Deliverable name	Report on the identified OS-independent abstractions for micro-ROS
Date	December 2018
Dissemination level	public
Workpackage and task	2.2
Author	Acutronic Robotics
Contributors	
Keywords	micro-ROS, robotics, ROS, microcontrollers, hardware abstraction
Abstract	This document provides an overview of the RTOS abstractions analyzed and implemented, linked to the Task 2.2, *Platform lower level firmware and libraries*.

Contents

1 Summary

In this document, we analyze the RTOS abstraction layers (AL) we have created for the modular stack of micro-ROS. Firstly, we have analyzed the CMSIS Cortex microcontroller for abstracting RTOS calls. In order to build these ALs, we have analyzed the interfaces of NuttX, the project's default RTOS, and FreeRTOS. This has allowed us to compare and establish ALs for such a different RTOSes.

The ALs eases the exchange of the used RTOS under the micro-ROS layered stack. As the ALs provides upper layers abstractions the rest of software requires, this will allow reusing code in different RTOSes, reducing the invested time at the development stage.

The adapters are the implementation of unified APIs made in each RTOS, coded taking into account its singularities and coding style. We have tried to find a common ground where a simple API is offered to the developer. Making a common abstractions implementation was not an easy task.

Note that this does not implies that the use of the ALs is mandatory, it is just a tool that helps unifying the use of different RTOSes in micro-ROS. In order to understand the performance sacrifice the AL inquires, performance tests needs to be performed.

2 Acronyms and keywords

Term	Definition
RTOS	Real-Time Operating System
AL	Abstraction Layer
XRCE-DDS	Extremely Resource Constrained Environments
UART	Universal Asynchronous Receiver Transmiter
SPI	Serial Peripheral Interface
I2C	Inter-Integrated Circuit
CAN	Control Area Network
CMSIS	Cortex Microcontroller Software Interface Standard

3 Introduction

OFERA consortium aims to provide, a modular stack of micro-ROS. To offer this feature, consortium members have analyzed how the project output fits into ROS 2 architecture for embedded-systems and how this parts could be modularized.

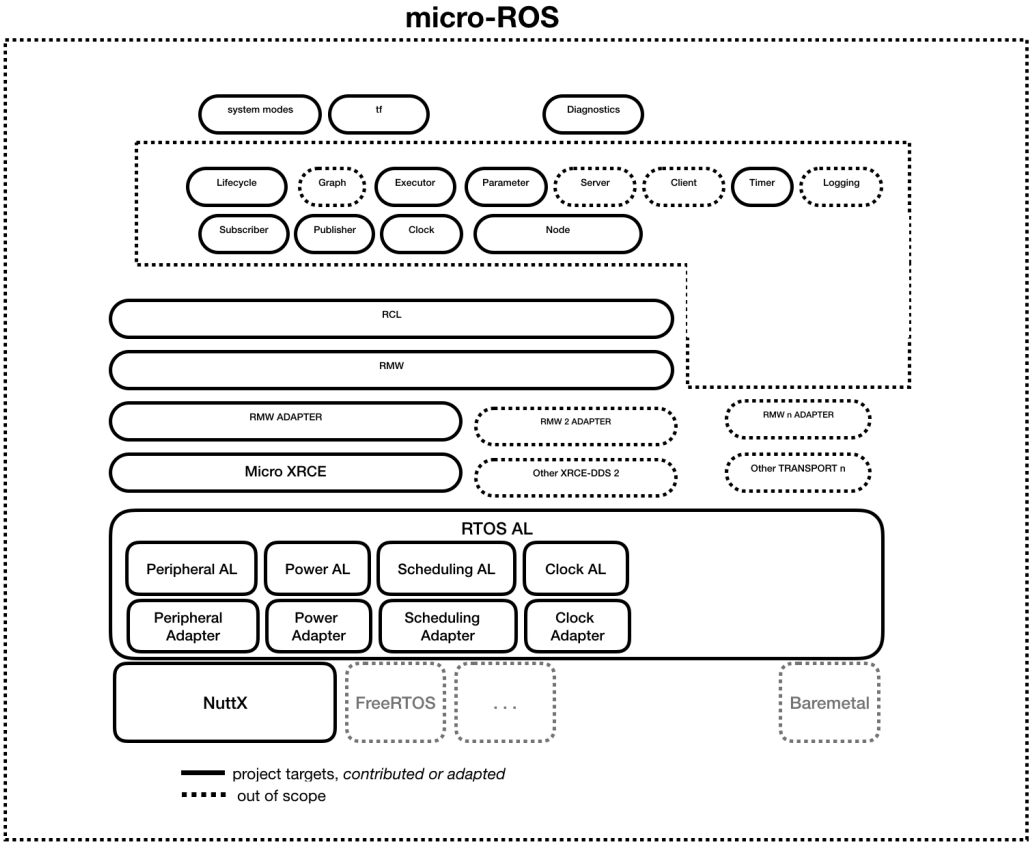
This modularization will allow the exchange of parts in convenience, adding or deleting non-required parts by user or exchanging parts which ones that add more value due to its characteristics (i.e. switching among RTOSes or middleware implementations). This impacts directly, among other elements, to the RTOS. This is due to it provides the basic functionalities to the rest of the system such as, peripheral access, clock or timing resources to the top layers. These mentioned resources are consumed typically by upper layers in the software stack, such us the middleware

transport layer, power sleeps determined potentially by user-code or timer primitives used by ROS 2 code.

This document discusses the benefit of using the RTOS abstraction layers and explains its first implementation using two different RTOSes as base: NuttX and FreeRTOS.

4 Abstraction Layer in General Architecture

The next image shows the modularized micro-ROS stack, where all the layers are presented:



This images shows all the outcomes project is going to offer, starting from low-level implementation where default RTOS support is provided and the mentioned ALs are implemented. At the top of it, we find the middleware implementation, where the provided abstraction layers are used.

The selection of the abstraction layers has been depending different factors: common RTOS functionalities shared between each others, desired micro-ROS features and the resource-usage upper layers have, which are dependent of the RTOS and the hardware, for example, hardware clocks.

The idea to underline is that the abstraction layer use is not mandatory, but provides interoperability feature to the code that uses it. As explanatory example, a sensor driver coded using the ALs, will be possible to exchange it across different RTOSes. It also has impact in the transport layer of the XRCE-DDS Client, where, if in the transport layer is used, the exchange of the RTOS is possible not modifying the DDS the code.

As rule, we have set the objective of having a minimalistic AL that would help developer to easily use each AL.

As it can be seen, we have split the ALs into four groups:

- Peripheral: it provides abstraction for different communication buses. The adapters provides, to each communication bus and RTOS, the abstraction implementation.
- Power: it provides abstraction for using power saving routines.
- Scheduling: it provides abstraction for scheduling the different tasks micro-ROS application uses.
- Clock: it provides abstraction for clock routines. This clocking is typically used under DDS and ROS 2 layer.

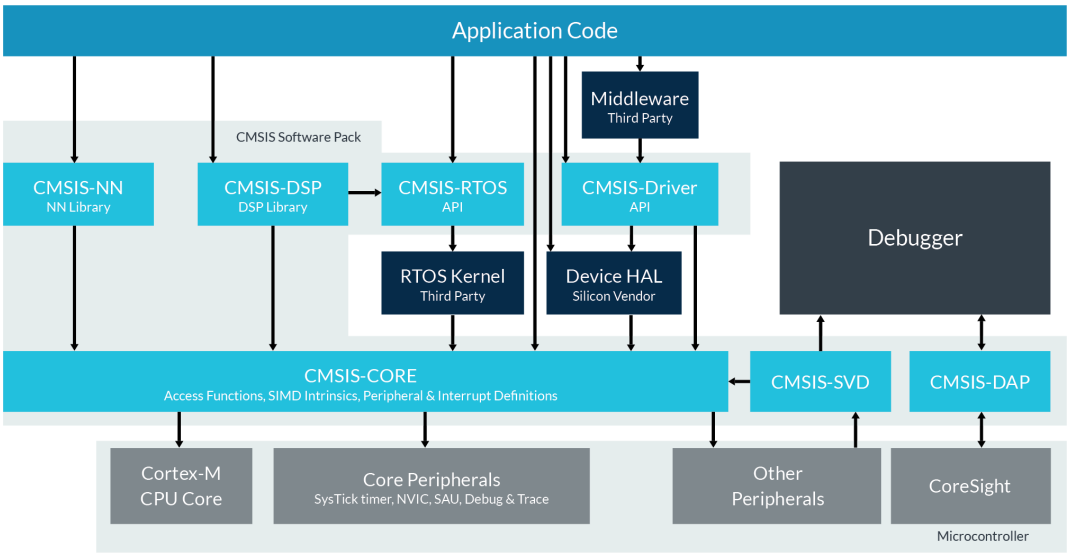
As it can be seen, these abstraction layers provides the common hardware resources micro-ROS software stack requires from the RTOS.

5 Abstraction Layer Analysis and Implementation

Firstly, we have a analyzed CMSIS abstractions ARM offers. Cortex Microcontroller Software Interface Standard (CMSIS) is a set of hardware abstraction layer (HAL) for Cortex-M processors. The CMSIS HAL is divided in multiple chunks, depending its functionality. This HAL could be use for DSPs, debugger interfacing or core-startup tasks, among many other features.

Regarding micro-ROS, as the project is using boards supported by the RTOS, the most interesting HAL to reuse is the one related to peripherals. Using a HAL that abstracts peripherals would help reusing drivers and code related with the sensor/actuators micro-ROS desires to control and the communication means to use for talking to the micro-ROS network.

The next picture shows CMSIS structure diagram:



CMSIS-RTOS implements threading and scheduling tools and CMSIS-Core provides some core functionalities, such as some peripheral access, system timer or debugging access. The problem with this abstraction layers is that they are at very low level and they are not at the same level of what RTOSes provides.

For instance, STM32Cube software provides FreeRTOS system calls for STM32 Cortex devices. This software sets-up all the software files you require for having FreeRTOS working. At its lower level, some of CMSIS abstraction primitives are used, but they are afterwards used by FreeRTOS. As NuttX uses also a different approach of setting abstractions from the hardware, offering a POSIX like interface, CMSIS and NuttX are also not complementary. Due to this, we consider that the use of CMSIS is not beneficial.

Once we have discarded CMSIS to build abstraction layers, we have taken the two RTOSes we are analyzing in the project to make a comparison between their interfaces. One of the main hurdles found joining NuttX and FreeRTOS in a common AL is that NuttX tries to follow as much as possible POSIX standard. On the other hand, FreeRTOS uses much lower level calls, being a RTOS that is much more near from the hardware level than NuttX.

As each RTOS has its own peculiarities, in some cases, it was necessary to do some jumps between the different layers of the RTOS to offer the unified APIs we present at this document.

5.1 Peripheral AL

This abstraction layer provides means of accessing to the hardware resources of the microcontroller in order to establish communications. This typically includes buses such us UART, I2C or SPI. In order to have more robust communication means that are able to operate in harsh environments, such us industrial factories or facilities, support for CAN or RS-485 communication buses would be also desired.

5.1.1 I2C AL

I2C is a communication bus which is characterized by being synchronous, multi-slave and multi-master, using a serial type of communication. As this bus is intended for connecting slave devices to microcontrollers, this implies that the abstraction layer requires of reading and writing capabilities and bus handling properties.

In any system, the I2C peripheral has normally two basic operations: read a register or write a register from any of the devices that is attached to the bus. Each register could have a different size in bytes. This case needs to be handled and, in some RTOSes, they separate implementations of one-byte operations to multiple byte operations.

In this AL, we have tried to implement an API that simplifies the use of this peripheral, but without giving up to any of the options the standard API of each RTOS offers. The implemented API allows having very similar read and write functions in both RTOSes. Also, some extra functionalities were implemented, to make the development of drivers based on this API easier, such as:

- When reading a register from the slave device, the function internally does the request, writing at the bus and reading the answer from the slave.

- When you request a several byte-size register data read, the response order of the data is in inverse order. We have set it in the right position when the API returns the value.

These are the calls I2C AL API offers:

```
void uros_i2c_write(uint8_t device_addr,uint8_t regaddr, int8_t regval,uint8_t len )

uint32_t uros_i2c_read(uint8_t device_addr,uint8_t regaddr,uint8_t len )

void uros_i2c_init()
```

Nuttx and FreeRTOS have I2C calls which are very alike, excluding the fact that in NuttX the bus initialization is made at boot-up, where the device to control is mapped to the file system. The FreeRTOS implementation has been straightforward, but the implementation of the API in NuttX has been more complex.

In NuttX, we have added the chance of using in two different manners the same API. The first one, that functionally resembles more how FreeRTOS works, bypasses or fills-up some requirements POSIX interface has. This implies that the bus bring-up is not longer done at boot-up but when the driver call is done. The second one, wraps-up the NuttX's POSIX interfaces, abstracting this interface to the user of the AL APIs. So this means that, when using I2C AL in NuttX, developer could stick to the use of POSIX interface or uses the implementation that resembles more to FreeRTOS.

5.1.1.1 Limitations In this first alpha version of I2C AL, we achieved a basic functionality, consisting the read and write of data to a slave. But the AL has some limitations. It doesn't allow you to use advanced features of the peripheral:

- It is only possible to use one of the I2C buses available in the development boards.
- We can't modify the configuration of the peripheral. Parameters such as frequency are set by default in the configuration.
- It's not possible to change the role of the device, being master by default.
- In FreeRTOS, the read/write operations, are blocking operations.

5.1.2 SPI AL

SPI is a synchronous communication bus. It's formed by one master device and one or multiples slaves devices attached to the bus. This communication bus is full duplex and faster than I2C in terms of data transfer. But on the other hand, it requires a bigger amount of signals: Master Output Slave Input (MOSI), Master Input Slave Input (MISO), Clock (CLK) and Chip Select (CS) for selecting the slave the master talks to. The use of this peripheral it's very similar to the I2C bus, where read and write of slave registers is performed.

The implemented API and its functionality tries to simplify the bus usage to the developer. We also have implemented some improvements that makes easier to work with this peripheral:

- When reading a register, the request and reading of the data is implemented.

- As the retrieved data from the reading is inverted, the ordering is done, so the user does not need to do it.

This are the calls SPI AL API offers:

```
/**************************************************************************
 * Description: This function writes a define value to a register address.
 * Arguments: regaddr ->  Register address where we want to write a value,
 * regval -> Value to write, len -> Length of the data that we want to write
 * Return: Nothing
 * Notes:
 *************************************************************************/
void uros_spi_write(uint8_t regaddr, int8_t regval,uint8_t len )


/**************************************************************************
 * Description: Read the value of a register.
 * Arguments: regaddr -> address to be read,
 * len -> Number of bytes that will be returned
 * Return: The returned data.
 * Notes:
 *************************************************************************/
uint32_t uros_spi_read(uint8_t regaddr,uint8_t len )

/**************************************************************************
 * Description: Function to set the SPI configuration in freeRTOS.
 * Arguments:Nothing
 * Return:Nothing
 * Notes:
 *************************************************************************/
void uros_spi_config()
```

Regarding the AL implementation in each RTOS, the main difference resides in the chip select handling. In FreeRTOS, the handling of the CS is done using a standard GPIO writing call. But in NuttX, due to, again, the POSIX interface, a workaround was required. In order to ease it, we have added a GPIO call abstraction that enables directly the call of the GPIO from the application, like it is done in FreeRTOS.

As happens in I2C, using this AL in NuttX, the developer has the chance of using POSIX-like calls or using the direct GPIO call to control the CS.

5.1.2.1 Limitations SPI is more complex peripheral that the rest we had abstracted. We have achieved a proof of concept where read and write of multiple bytes over SPI bus is possible. These are the limitations the implementation has:

- The AL uses a default SPI bus which it's not possible to change to any of the available buses in the MCU.

- It's not possible to modify the default configuration of the bus.
- In this first version, advanced features are not implemented, such as DMA use.
- In FreeRTOS, the read/write operations, are blocking operations.

5.1.3 UART AL

UART is an asynchronous serial communication protocol. This protocol works point to point and each device needs to have exactly the same configuration. Normally, there are three basic operations in this communication bus: read messages from the bus, write messages to the bus and the detection of new data ready to read in the bus.

In devices that uses this serial communications, typically, the APIs offers sending/receiving data and the opportunity to configure the communication characteristics, such us baud rate speed, flow control or parity bit usage. The AL implemented inspires in this usage and extends the bus handling.

As the nature of both RTOSes is different, at the time of implementing the abstraction layer, there are some differences to notice. At write function, the API integrates the length reading of the message to send. At FreeRTOS, the writing is done using low-level functions, meanwhile, in NuttX, POSIX system calls are used.

The API offers `open` and `poll` functions which, at implementation level, there are some differences in between RTOSes:

- Open function: In NuttX is straightforward thanks to the POSIX system. This is because the configuration is already done in the configuration menu. On the other hand, in FreeRTOS, it is required to configure the parameters of the connection manually.
- Poll function: NuttX implements the `poll` function to know if there is any data available in the bus. But in FreeRTOS there isn't any tool available with a similar behavior. So, we developed a function that reads the state flags of the bus. As outcome, this workaround gives us a similar behavior of the AL in both RTOSes.

The last remarkable difference is that the `close` function it is implemented under NuttX using the POSIX system but, under FreeRTOS, no action is taken.

This is the API of the implemented UART AL:

```
/**************************************************************************
 * Description: This function configure the UART port.
 * Arguments: Nothing
 * Return: Nothing
 * Notes:
 **************************************************************************/
void uros_uart_open()

/**************************************************************************
 * Description: This function close the auxiliary UART port
 * Arguments: Nothing
```

```
 * Return: Nothing
 * Notes:
 ***************************************************************************/
void uros_uart_close()

/***************************************************************************
 * Description: Send over the UART the data that you want.
 * Arguments: Char array to be send over the UART
 * Return: Return the number of bytes send or the 0 if it doesn't send anything
 * Notes:
 ***************************************************************************/
uint8_t uros_uart_write(char *buffer)

/***************************************************************************
 * Description: This function receive the data available in the UART.
 * Arguments: buffer_out -> Array to be saved the data received.
 * Return: The number of bytes received.
 * Notes:
 ***************************************************************************/
uint8_t uros_uart_read(char *buffer_out)

/***************************************************************************
 * Description: Check if there is any data available in the UART.
 * Arguments: Nothing
 * Return: True if there is any available data and false if no data are available
 * Notes:
 ***************************************************************************/
bool uros_uart_poll()

/***************************************************************************
 * Description: Configure properly the UART
 * Arguments: Nothing
 * Return: Nothing
 * Notes:
 ***************************************************************************/
void uros_uart_config()
```

5.1.3.1 Limitations In the future, would be interesting to implement interruption-based bus status check, instead of polling, due to the impact this procedure could cause in the application performance. Also, it's necessary to implement a configuration routine, that allows the user to change the configuration of the peripheral without touching the `menuconfig` of the NuttX RTOS.

5.1.4 GPIO AL

The GPIO is a peripheral that allows you to make different operations with the external pins. The operations that are normally are possible are digital read or writes, analog read or writes and inter-

ruption mode.

Regarding the implementation of the AL, the aim is to simplify the use of the GPIO, allowing to use in both RTOS in a bare-metal way in all the layers of the RTOS. This is how the AL API calls have been set:

```
/*****************************************************************************
 * Description: This function configure the working mode of each GPIO pin.
 * Arguments: Pin to setup (0~13) and mode(INPUT,OUTPUT,INTERRUPTION)
 * Return: Nothing
 * Notes: Interruption is not working yet.
 *****************************************************************************/
void uros_gpio_config(uint8_t pin, uint8_t mode)

/*****************************************************************************
 * Description: This function write a value to the selected pin.
 * Arguments: Pin to write(0~13) and value (HIGH, LOW).
 * Return: Nothing.
 * Notes:
 *****************************************************************************/
void uros_gpio_write(uint8_t pin, uint8_t value)

/*****************************************************************************
 * Description: This function reads the value of a pin.
 * Arguments: Pin(0~13).
 * Return: The value of the pin (0 or 1).
 * Notes:
 *****************************************************************************/
bool uros_gpio_read(uint8_t pin)
```

Even that we have achieved the same behavior for both RTOSes, the internal implementation is quite different. In NuttX, POSIX is used in its implementation. To achieve this, we did some modifications. As in NuttX is not allowed to use the GPIO as in a bare-metal system, accessing directly to the pin, you need to register the GPIO before you make use of it. Fortunately, there is a way to do this and access the GPIOs like in a bare-metal application. For that purpose, we have created a `config` folder of the board within NuttX. Thanks to this, we can use GPIO write/read functions like in bare-metal applications, respecting the NuttX structure and allowing us to access the GPIO from any part of the code.

To implement the AL in FreeRTOS, it was necessary to add a thinner layer to the standard call the RTOS offers and adapt some peculiarities. This singularity is related about how the arguments are used under FreeRTOS calls.

5.1.4.0.1 Limitations In this first version of the AL, a determined number of GPIO pins for read and write purposes are set. This module of the AL it's only implemented in the Olimex STM32-E407 board, where the Arduino compatible pins are in function. The AL has the next limitations:

- It's only allowed to use the Arduino expansion pins of the board. Actually, it's totally possible to use any pin, but a pin mapping extension needs to be done.
- This AL it is not implemented in the L1 board.
- It's not possible to use advanced configuration features in the configuration of the GPIO. The pins could only be used for reading or writing. For example, it's not possible to set a pull-up resistor in the GPIOs of the MCU.
- Interruption functionality is not implemented in NuttX, due to a problem cleaning the interruption flag.

5.2 Power AL

The power management is the ability of controlling the different energy saving modes the MCU can enter. This is done normally setting MCU specific registers to certain values. Usually, these different power modes are classified in run, stand-by, sleep or stop modes. This implies that when the microcontroller enters into a specific power mode, its functionality changes. For example, it normally involves a reduction of the clock frequency, or a reduction of the voltage of the MCU, or disabling some of the peripherals, or even disable everything except a few registers and setting the clock to a very low-frequency. This is MCU power mode dependent.

Once the MCU enters into a power saving mode, there are normally three ways to set it back into operational mode:

- Using a wake-up timer, that can wake up periodically or restart the MCU.
- Using a wake-up pin. When the MCU detects an interruption in this pin, the MCU will wake up.
- Modifying the value of registers of the MCU, normally in non-deep sleeping modes.

For the deepest power saving modes, MCU restart is normally required, loosing all the data stored in the RAM.

It is important to notice that this feature is heavily related to the power saving the MCU architecture offers and not only to the RTOS.

The AL that we want to develop for the power management of the micro-ROS devices, consists on offering an API where the user can select to enter and wake up from any power saving modes that are available. This would allow easily to give the user the control over, at which parts of the application the MCU, should enter in power saving modes and when it should wake up again.

In order to implement the proposed API, we need to understand better how each RTOS power saving methods works. The approach NuttX uses is more complex than in FreeRTOS. This is because it implements a *power scheduler* that decides automatically which power mode is the best one to operate at. This approach gives us less freedom to decided which mode do we want to set the MCU at the application code. This implies that user can select in which power modes to enter, but cannot command to wake it again up, because that is the power scheduler task.

FreeRTOS uses a simpler approach. Its power saving mechanisms are much more simpler that NuttX, but it also offers to the user a bigger control. It allows going into different power modes manually. To wake again up, you can set a timer or configure a wake-up pin and, in some modes, even modify the power-mode registers to come back to run mode without losing data.

This big differences makes difficult to find a meeting point between both RTOSes when implementing the power AL.

5.2.0.1 Limitations Right now, power AL tests have been performed, allowing to enter in any power mode from both RTOSes. FreeRTOS power AL module is fully functional (It can go to any power mode and wake up) but in NuttX, we are encountering problems when trying to wake up the MCU from the sleeping modes we have set. The reasons are two:

- In NuttX we have available a power scheduler that decides which power mode should run. The problem is that the wake up functions are implemented to work with that tool, so it's difficult to work outside this function.
- When it goes to a sleep mode, the frequency of the clock is reduced and some peripherals are disabled, but when you try to wake up manually, it doesn't set the previous configuration properly.

5.3 Scheduling AL

Schedulers are software entities which handle process scheduling using different techniques or objectives. Their main task is to decide which process to execute at each time in the processing unit. In real-time environments, such as embedded systems -for example robotics- the scheduler also must ensure that processes can meet deadlines. The meeting of the deadlines is required in some use-cases, where safety functionality of the system is in stake. These processes could be of different nature, such us communication, data processing or control tasks. In order to obtain more information about scheduling, please refer to deliverable 2.10, *Report on RTOS Scheduling*.

At this stage, a basic scheduling mechanism for micro-ROS is desired. This starting point will allow as defining a scheduling system where the different tasks are appropriately managed, making it independent of the RTOS in use.

In order to make the implementation, each RTOS features have been analyzed. The scheduler has some implementation differences and behavior in each RTOS.

In a nutshell, the implementation of the function to create tasks are pretty similar in both RTOSes. The main difference is that in FreeRTOS you need to define and then create tasks and that, in NuttX, you only need to create them out. The delete function is different in each RTOS, because FreeRTOS has a function that automatically deletes the task. In NuttX this works differently, first we need to stop the task with a signal, and then we need to kill it. The kill and stop are wrapped in the function kill() of NuttX.

The task `delay`, `yield`, and `getid` have a very similar implementation. The function to get and set priority is implemented and works nicely in FreeRTOS, but in NuttX we're not allowed to take or modify that data in the library layer. Finally, there is a last function that gives some conflicts, called `task_periodically`. This function it's a native function of FreeRTOS. It sets in pause mode the task for a determined time and, when wakes up again, gives the maximum priority. But in NuttX it's difficult to do something equivalent, due to the problems to set the priority.

This is how the AL API calls have been set:

```
/*****************************************************************************
 * Description:This function creates a task attaching to a function,
 *             giving a priority and a stack size.
 * Arguments: name-> A human understandable name, fun -> function to attach
 *            priority -> priority of the function,
 *            stack_size-> Stack size of the task.
 * Return: Nothing
 * Notes:
 *****************************************************************************/

void uros_task_create(char *name, void *fun, uint8_t priority, uint16_t stack_size)

/*****************************************************************************
 * Description: This function delete the task with the ID given.
 * Arguments: thread_id -> ID of the task to delete.
 * Return: Nothing
 * Notes:
 *****************************************************************************/

void uros_task_delete(void *thread_id)

/*****************************************************************************
 * Description: This function "sleep" the task for a number of ms.
 * Arguments: time_ms -> number of miliseconds to "sleep" the task
 * Return:Nothing
 * Notes:
  *****************************************************************************/

void uros_task_delay(uint16_t time_ms)

/*****************************************************************************
 * Description:Change the priority of a task to the max priority
 *             periodically and then come back to the default priority.
 * Arguments: time_ms ->  Period of changing the priority of the task.
 * Return: Nothing
 * Notes:
  *****************************************************************************/

void uros_task_periodic(uint16_t time_ms)

/*****************************************************************************
 * Description: Returns the priority of the task.
 * Arguments: thread_id -> ID of the Task
 * Return: Priority of the task.
 * Notes:
 *****************************************************************************/
```

```
int8_t uros_task_getpriority(void *thread_id)

/*********************************************************************
 * Description: Set a different priority to the task.
 * Arguments: thread_id -> ID of the task, priority -> New priority of the task
 * Return: Nothing
 * Notes:
 *********************************************************************/

void uros_task_setpriority(void *thread_id,int8_t priority)

/*********************************************************************
 * Description: Get the ID of the Task
 * Arguments: Nothing
 * Return: The ID of the Task.
 * Notes:
 *********************************************************************/

void uros_task_getid()
```

5.3.0.1 Limitations The basic functionality of a scheduler has been achieve: Create and delete tasks, pause the execution and modify the priority. But in both RTOS we have the problem when changing the scheduler algorithm. In FreeRTOS, is possible to change it, but in boot time. In NuttX it changes automatically, but at this moment it's not possible to choose manually which scheduling algorithm we want to use.

5.4 Timer AL

Timers are hardware units that cohabits within the microcontrollers together with another blocks. They allow MCU measuring time and normally they are implemented using countdown or count-up cycle. Once the counting is complete, they interrupt the MCU to aware that the requested time has expired. Thanks to timers, you can set periodical events and measure the time in reference to an external or internal clock. Normally you can configure timers to have two functionality: periodical timers or one-shot timers. The difference among them is that, the first one, starts counting again and, the second one, stops working once it raises the interruption.

In micro-ROS, the timer use is made not only at RTOS level but also in higher layers, such as the transport layer of the Micro XRCE-DDS Client. This feature also would be desired by the end-user that codes micro-ROS application. For this reason, as it is an important and valuable resource in embedded applications, we have decided to abstract it. For that purpose, we built an abstraction layer that allows using MCU hardware timers each RTOS provides.

At this first version, periodic timer has been implemented, this is how its API looks like:

```
/*********************************************************************
 * Description: This function is function is to modify the period of the
 * timer when is already working.
```

```
 * Arguments:  timer_num-> Timer that you want to use (1~4), period ->
 *  is the period of the timer (0 ~4294967296) in ms
 * Return: Nothing
 * Notes:
 ***************************************************************************/
uint8_t uros_timer_period(uint8_t timer_num, uint32_t period)

/***************************************************************************
 * Description: This function configure the timer.
 * Arguments:  timer_num-> Timer that you want to use (1~4), period ->
 *  is the period of the timer (0 ~4294967296) in ms
 * Return:  1 or 0, if the configuration was a success.
 * Notes:
 ***************************************************************************/
uint8_t uros_timer_config(uint8_t timer_num, uint32_t period)

/***************************************************************************
 * Description: Start the specified timer.
 * Arguments: Timer_num -> Number of the timer to be use.
 * Return: If success 1 otherwise 0
 * Notes:
 ***************************************************************************/
uint8_t uros_timer_start(uint8_t timer_num)

/***************************************************************************
 * Description: Stop the specified timer.
 * Arguments: Timer_num -> Number of the timer to be use.
 * Return: If success 1 otherwise 0.
 * Notes:
 ***************************************************************************/
uint8_t uros_timer_stop(uint8_t timer_num)
```

As it can be seen, the AL provides functionalities to set up the timers, configure them and start or stop them. The actual implementation makes use of the timers 2 to 5 and are configured to be of 32 bits. The first timer is typically used by both RTOSes for operational purposes.

As what happens with the other ALs, the implementation made in each RTOS differs due to the nature of each one. We have worked with the POSIX interface under NuttX to abstract it. The advantage of NuttX is that user can configure which function the timer calls after is done counting. In contrast, FreeRTOS has always defined which function the timer calls once it triggers the interruption. Apart from this, FreeRTOS just makes calls to handle the register of the hardware timers.

5.4.0.1 Limitations The timer AL module can create a 32 bits periodic timer, start/stop the timer or modify the working period on running time. But it's not possible to split the 32bit timer into sub timers of 16 bits or 8 bits. Also we can't use one-shot timers. On the other hand, in NuttX you can set any function as callback function, but FreeRTOS only allows you to use the default callback function.

6 Outcome Analysis and future steps

As we have seen during the document, abstracting different nature RTOSes is not a trivial task, but we have probed that is possible. Even that, we have achieved creating a proof of concept of the most important features micro-ROS software stack requires from the RTOS level, the impact of these AL needs to be analyzed.

One of the topic to analyze is until which extend the AL could be developed. It is required to have a better implementation of them in overall. Right now, the configuration options of many ALs are limited, for example, serial configuration is not possible or bus selection is not allowed either. The AL should be an enhancement for reusing code and enable RTOS switch, but it shouldn't impact dramatically the RTOS features and performance.

Test-benches also should be run, in order to gather scientific data about the performance costs of the ALs. This would also help to bring light to see the suitability of the ALs. Consortium will keep analyzing if the AL are appropriate for its use in the project.

7 Source to AL proof of concept

The ALs proof of concept has been released under micro-ROS Abstraction Layer repository (commit 20234ba), where separate files are provided for FreeRTOS and NuttX AL implementation.