NTNU
Norwegian University of
Science and Technology

Energy Aware RTOS for EFM32

Angelo Spalluto

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Lasse Natvig, IDI
Co-supervisor: Marius Grannæs, Energy Micro

Norwegian University of Science and Technology
Department of Computer and Information Science

NTNU
Norwegian University of
Science and Technology

Energy Aware RTOS for EFM32

Angelo Spalluto

Master of Science in Computer Science
Submission date: 18 June
Supervisor: Lasse Natvig
Co-supervisor: Marius Grannæs, Energy Micro

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This thesis is a continuation of Martin Tverdal's master thesis. Energy Micro is a Norwegian semiconductor company, located in Oslo, which focuses on 32-bit microcontrollers with ultra low energy consumption. The EFM32 microcontroller family is based on the ARM Cortex-M3. FreeRTOS is a small and free open source OS targeted for embedded devices.

Martin's thesis ported FreeRTOS to the EFM32 and implemented rudimentary support for energymodes and using the RTC(Real Time Counter) for tickless idle. This thesis aims to improve on this work making the approach more robust.

Central subtasks are:
* Including general hooks in FreeRTOS for:
  - Controlling sleep modes in a generic way;
  - Extending the task control structures for sleep mode control;
  - Using other systems than systick for keeping time in FreeRTOS;
  - Implement support for Tickless kernel;

* Using these hooks in a EFM32 specific port of FreeRTOS, which has the following functionality
  - Using the RTC for keeping track of time;
  - Enabling the use of energy modes through an intuitive API;

Assignment given: 15 January 2011
Supervisor: Lasse Natvig,IDI
Co-supervisor: Marius Grannæs, Energy Micro

Abstract

Power consumption is a major concern for portable or battery-operated devices. Recently, new low power consumption techniques have been used to achieve acceptable autonomy battery-powered systems. FreeRTOS is a real-time kernel designed especially for embedded low-power MCUs. Energy Micro develops and sells energy friendly microcontrollers based on the industry leading ARM Cortex-M3 32-bit architecture. The aim of this thesis is to propose a new *FreeRTOS Tickless Framework* solution that exploits the power modes provided by EFM32. Three different solutions have been proposed, such as *FreeRTOS RTC*, *FreeRTOS Tickless with prescaling* and *FreeRTOS Tickless without prescaling.* The simulations showed that the Tickless Framework saves energy from *15x* to *44x* more than Original version of FreeRTOS. Using a self-made benchmark the battery (1500 mAh) lifetime has been increased from 11 days to 487 days.

Acknowledgements

First of all, I want to dedicate this thesis to my *parents*, my sister *Antonella* and my girlfriend *Veronica*. Without their support, I would have been truly lost.

A special eternal thanks to *Veronica*, for her love and patience to wait me. Moreover, I would also to thank all of *my relatives* and *Emanuele*.

I would express gratitude to my supevisor, professor *Lasse Natvig* at the Department of Computer and Information Science at NTNU, for his guidance and to gave me this opportunity. I would also thank *Marius Grannæs* from Energy Micro for his help and support. A special thank to my office mate *Stefano Nichele*, for making the time at NTNU a so pleasant.

Contents

List of Tables

List of Figures

Listings

Part I

Introduction and Background

Chapter 1

Introduction

"PCs and monitors account for 39% of the information and telecommunications industry's carbon emissions, which is equal to a full year of CO_2 emissions from approximately 43.9 million cars."

- **Climate Savers Computing**[1]

"We want to take an industry-leading approach to energy conservation. The technology is now available to make significant improvements in conservations, and we set out to deploy technology to both conserve energy and cut costs."
- **Jay Taylor**, Regulatory Engineer Strategist at Dell

1.1 Motivation

In the last five years, IT systems have dramatically increased the amount of energy used in computing centers. Nowadays, energy conservation is getting the major concern factor for companies that manufacture hardware components and design software algorithms. According to a research carried out by Climate Savers Computing [1], another source that cause a waste of energy, is the carelessness of people to leave PCs on all nights. The results show, that, if a company left 10.000 PCs on overnight, it wastes 1.5 million KWh with losses close to 285.000 €, and carbon emissions around 887 tons of CO_2. According to the same research, U.S. government predict that the electricity prices will increase up to 35% in the 2030. Thus, the grows of power consumption is involving both governments and IT industry to take immediate countermeasures against this phenomenon.
In the following sections I have outlined the reasons why it is useful to carry out research in these fields. Especially, the main goal of my Master thesis is to give a *green contribution* in those services and products, touched every day from a huge amount of people.

1.1.1 Green Computing

People in last two decades, associate the term *Green Computing* or *green IT* to indicate those techniques that use computers and their resources in an environmentally responsible manner. Green Computing develops and studies new possible techniques to reduce the impact of energy consumption in new technological devices. Recently, companies, organization and individuals, increasingly rely a new strategic energy solutions that allow them to save a considerable amount of money. Moreover, new companies are emerging due to the importance of manufacturing low power components such as multicore processors, LED display, new storage devices and low power MCU. Equally important, is the impact of energy algorithms that allow to manage the usage of resources much more efficiently, reducing the correlated energy consumption. Currently, many new algorithms are applied in a wide range of contexts such as scheduling algorithm, resource allocation, encoding algorithm and peripheral management. Energy aware systems cutback energy up to 30% using energy efficient coding. Furthermore, these new green-IT solutions reduce the emissions of greenhouse gasses in the world, involving a good aid to the entire environment.

1.1.2 Embedded System

The impact of power consumption does not only leverage the field of desktop computers or data centers, but even portable devices commonly known as *Embedded Systems*. Embedded systems are small computers designed to perform dedicated tasks. Basically, these systems are employed in real time environments with strict time deadlines. Lately, many Embedded systems are applied in some areas such as *automotive*, *medical devices*, *defence* and *maritime systems*. The main aspect concern embedded systems is power consumption. This is particularly important for those applications that really need to save power during their life time.
Some examples of energy devices are *energy metering systems* (Power, water, gas and smart meters), *home and building automation* (remote, light and climate control), *alarm and security systems* (motion sensors, burglary alarms). Replacing a battery in some devices might be very expensive. This means, the system should be conscious of the amount of power it is using, taking appropriate steps to conserve battery life. Therefore, in order to extend their battery life, aggressive energy conservation techniques are needed to save energy in I/O devices. Due to reasons just explained, Embedded systems contain low power processor architecture called RISC. These architectures, provide high level performance and limited amount of energy consumption. Some examples of RISC processors are ARM, ATMEL AVR, Power PC, MIPS and SPARC.
There are several other methods to conserve power in embedded systems, such as *clock control*, *power sensitive processors*, *low-voltage ICs*, and *circuit shutdown*. Some of these techniques must be addressed by the hardware designer during the selection among different ICs systems.

1.1.3 Real Time Operating System (RTOS)

As already mentioned above, the style of algorithm implementation in these devices affect significantly the final power consumption. In fact, writing a stand-alone application (endless loop) is much more energy expensive than using a specific operating system called *Real Time Operating System (RTOS)*.
The roles of RTOS is to keep active a processor only when the system has to compute some operations. On the contrary, if an idle state occurs the RTOS pushes the whole system in an idle task. The OS should know in advance when to wake up the core unit and when to move it to idle state. On the other hand, a RTOS might provide to the user an internal framework that handles the power state of a MCU in efficient way. Although this feature represents a useful solution to cope energy consumption in portable devices, it is still quite hard to find modern RTOSes equipped with a well designed Power Manager (PM). Nevertheless, some vendors of RTOSes have already included features to handle sleep states. The support in this area is still poor, especially for open source companies that provide RTOSes. This plays up for those customers that are get used to work with open source platform. In last years, the amount of firms and private people that are making use of open source RTOSes is rising. In fact, adopting a proprietary solution a customer invests a considerable amount of money around 1-5K €for normal RTOSes and more than 5K €for complex platforms. Needless to say, the motivation of using an open source platform is due to the large amount of money that a customer can save in a final product

1.1.4 Government support

Recently, US governments announced a funding initiative equal to $52 billion to support energy friendly companies [2]. The investment are partitioned as follow, $3.4 billion for smart grids, $2.4 billion for batteries and e-cars, $43 billion for energy technology, $3.2 billion in power efficiency grants, a broadband internet initiative and lighting efficiency standard.
Furthermore, despite the disastrous impacts the 2008 worldwide financial crash, the demand of developing new energy solutions and products never slumped. This means that the market of small devices does not risk a collapse of sales. Therefore, governmental institutions are more involved to fund new initiatives.

1.2 Previous work

The contribution of my thesis is based on previous work done by Martin Tverdal. The aim of his thesis [3] was to provide a porting for FreeRTOS in EFM32 microcontrollers. Basically, Martin covered two main points, *Tickless system* and *management of energy modes*. Nevertheless, his contribution still need an addition work, in order to be considered as a valuable porting for FreeRTOS.
My Master thesis involves an additional work in those points that needed a major improvement. Martin provided a Tickless support (see section 4.3) for EFM32, based on using only a Real Time Counter (RTC) during idle state of system. As

he observed in his report, this kind of approach saves a huge amount of power consumption. He implemented the same approach used in Linux (version 2.6.34) before it was changed in the final solution currently used, *Totally tickless system.* One of the improvement of my master thesis is to use in FreeRTOS the last solution adoped in Linux. Furthermore, Martin provided only a specific porting of Energy Modes for FreeRTOS. He did not develop a *Generic Framework* even able to deal with other platforms. The work done by Martin was not totally approved by FreeRTOS. Also in this case, my thesis presents a different way to handle energy modes both in EFM32 and other platform compatible in FreeRTOS.
Besides, more information about work did by Martin are discussed and analyzed in methodology section.

1.3 Goals

The goals of my thesis allow to split up the work and define the correlated milestones. In this chapter it is reported a brief description for each of them.

1. Implement a generic energy management framework for FreeRTOS. Implement a specific porting for EFM32.

The purpose of this section is to propose a Generic Power Management Framework capable to deal with the power modes provided by most Microcontrollers. This framework must be able to handle the requests arisen from tasks and interrupts.

2. Introduce support to keep time in FreeRTOS using RTC timer instead of Systick timer. Implement a specific porting for EFM32

FreeRTOS keeps track of time using a register inside the SYSTICK module of CORTEX-M3. The unit core raises an interrupt every time the register is updated. Upon interrupt, FreeRTOS updates its internal variable called xTickCount. The new support changes the resource of timer from CORTEX-M3 to a timer generated by RTC. At the completation of this goal is expected to reduce the overhead introduced in Martin's solution.

3. Implement FreeRTOS support for Tickless kernel on EFM32

This support avoids to introduce inside the system a periodic interrupt generated from CORTEX-M3. The system provides a dynamic timer without using a period interrupt. This requires to redesign the scheduler of FreeRTOS. When an idle task occurs, it have to wake the core up at the upcoming task or if an asynchronous event occurs. Also in this case we use the RTC timer as main timer of the system.

4. Develop a Benchmark to perform simulations

This goal is used to demonstrate the behavior of the system using the new FreeRTOS solution. The benchamrks is used to perform some simulations with different versions of FreeRTOS.

5. Establish a contact with FreeRTOS maintainer

Establish a contact with FreeRTOS's mantainer to understand which are the limitations of the new proposed FreeRTOSes. Achieve the possibility to integrate these changes officially in FreeRTOS would enhance the contribution of my work for Energy Micro.

1.4 Thesis organization

The report has been divided in the following five parts: *Introduction*, *Methodology*, *Results*, *Conclusion* and *Appendix*. Below are described the chapters included in each of these parts.

Chapter 1 Provides motivations and goals of the Master thesis.

Chapter 2 Provides a description of Energy Micro company and a further explanation about Development Kit used in this work.

Chapter 3 Provides a general overview about RTOSes and some basic notions of the Time management field. This part is useful to understand the changes in FreeRTOS.

Chapter 4 Provides a general overview about FreeRTOS and it gives an explanation about its main data structures.

Chapter 5 Provides an explanation how to use the Energy Management Framework developed for FreeRTOS.

Chapter 6 Provides an explanation of the Framework used to keep track of time in FreeRTOS using a RTC timer.

Chapter 7 Provides an explanation of the Framework used to implement the Tickless support fo FreeRTOS kernel.

Chapter 8 Provides the results of the simulations performed for three different types of FreeRTOS. The *Original FreeRTOS 7.0.1*, *FreeRTOS RTC 7.0.1*, *FreeRTOS Tickless 7.0.1 with prescaler* and *FreeRTOS Tickless without precaler*.

Chapter 9 Remarks the results and the main points of the whole thesis.

Chapter 2

EFM32 Energy friendly microcontroller

"Energy Micro's mission is to make the world's most energy friendly electronics."

- **Geir Førre**, President and CEO of Energy Micro

"I have tried it tonight, and have to say your energy profiler rocks so incredible hard!! I love it. It is so fast and smooth, and that it warps to the exact line of code that generated a particular amount of current is amazing."
- Happy Energy Micro's Customer

2.1 Energy friendliness applications

Recently, the explosion of battery sensitive electronics devices, caused a necessity to design and develop low power devices. Energy Micro has addressed this issue by introducing in the market their *Energy friendly* products 32-bit EFM32 MCU. The 32-bit EFM32 MCU is the world's most energy friendly microcontroller for use in energy sensitive applications. In the next sections (2.2) is described which are the important technology factors that allow EFM32 to be defined as most energy microcontroller. In this chapter the main important features useful are discussed to understand the related work of thesis. More information are available on Energy Micro website. Figure (2.1) shows all of the possible EFM32 target applications.

Figure (2.2) shows the block diagram of EM32. Figure (2.3) shows the indicators color for each energy mode (see section 2.3.1) where it works for.

Figure 2.1: EFM32 Target Applications [4]

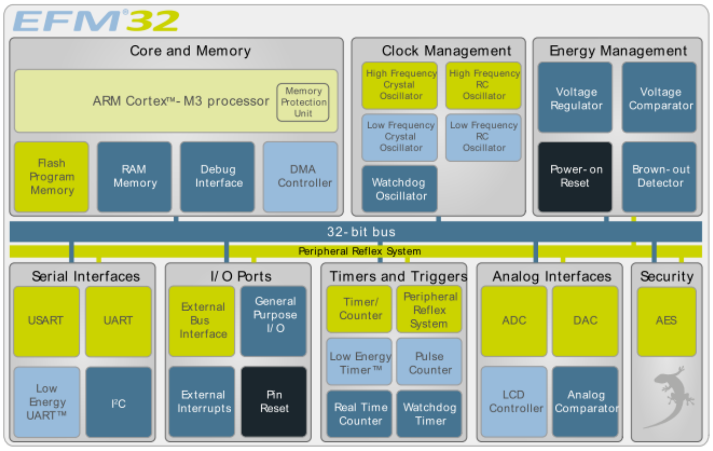


Figure 2.2: Block diagram of EFM32[5]

2.2 Ten important technology factors for EFM32

At the designing step of EFM32 family, were defined ten important factors that make EFM32 different from other microcontrollers. Below, there is a brief description for each of them.

1 Very low active power consumption

One of the most important factor in EFM32, is to have a very low active power when the processor is running. EFM32 tries to keep the power as low as possible. Using a clock of 32Mhz and running code in Flash memory, EFM32 only consumes

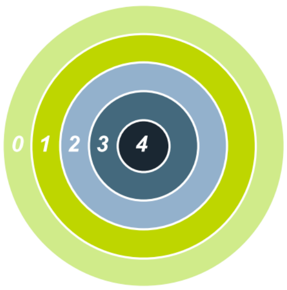


Figure 2.3: Energy Mode indicator[5]

180 µA/MHz.

2 Reduced processing time

Processing time is another factor that strongly affects the energy consumption in a microcontroller. In fact, the processor must spend as much time as possible in deep state. The core unit used by Energy Micro to meet these requirement is ARM Cortex-M3 [6]. The performance of Cortex-M3 allows EFM32 to solve complex problems in few clock cycles.

3 Very fast wake-up time

EFM32 keeps as low as possible the wake-up time of MCU when it moves from deep sleep modes to active mode. In fact, before starting execution of code, MCU waits the oscillators' stabilization. The energy spent to wake-up the oscillators is wasted, because no processing can be done in this period. Hence, reducing the wake-up time is important to reduce the overall energy consumption. EFM32 has reduced the wake up time for deep sleep up to 2 µs.

4 Ultra-low standby current

Energy wasted in standby state is another important factor to take into account for reducing the energy consumption of MCU. In this state, EFM32 reduce the leakage current as much as possible.
The Deep Sleep mode includes RAM and CPU retention, Power-on Reset and Brown-out Detection safety features, and a Real Time Counter while only using 900 nA. In Shutoff mode the consumption is only 20 nA.

5 Autonomous peripheral operations

The peripherals in EFM32 are designed to operate with minimum intervention of CPU. In fact, EFM32 peripherals can run without using the CPU, therefore reducing considerably power consumption.

6 PRS Peripheral Reflex System

PRS allows the peripherals to communicate directly with each other without involving the CPU. In the meanwhile the CPU is sleeping, a peripheral can produce a signal and it triggers the reaction of another peripherals.

7 Energy modes

EFM32 provides the possibility to use different energy modes during its life time. More information are given in next section (2.3.1).

8 Energy efficient peripherals

EFM32 provides a set of low power peripherals that contribute with other factors to reduce energy consumption. Peripherals are:

- LCD controller driving 4x40 segments at only 0.55 μA;
- Low Energy UART, full communication at 32 kHz while consuming only 100 nA;
- 12-bit ADC performing 1 million samples/sec at only 200 μA;
- Analog Comparator using as little as 150 nA;
- Hardware accelerator for 128/256-bit AES encryption and decryption in only 54/75 cycles;

9 AEM Advanced Energy Monitoring

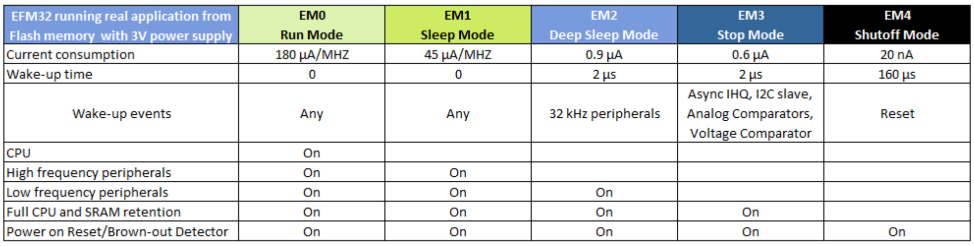
EFM32 provides an advanced energy monitoring system (AEM) which gives an instant feedback on the power consumption of the prototype application. This tool also integrates a full J-Link from Segger for easy debugging and programming.

10 EnergyAware Software

The energyAware software suite includes energy code examples, CMSIS libraries and a Profiler that reads the kits Advanced Energy Monitoring (AEM) system data and enables simple graphical visualization and optimization of application.

2.3 Energy modes in EFM32

As already explained above, one of the important feature of EFM32 is the possibility to use different Energy Modes (EM0-EM4). The Figure (2.4) shows an high overview of Energy modes in EFM32. Some parts of this section is taken from my Specialization project[7].



EFM32 running real application from Flash memory with 3V power supply	EM0 Run Mode	EM1 Sleep Mode	EM2 Deep Sleep Mode	EM3 Stop Mode	EM4 Shutoff Mode
Current consumption	180 µA/MHZ	45 µA/MHZ	0.9 µA	0.6 µA	20 nA
Wake-up time	0	0	2 µs	2 µs	160 µs
Wake-up events	Any	Any	32 kHz peripherals	Async IHQ, I2C slave, Analog Comparators, Voltage Comparator	Reset
CPU	On				
High frequency peripherals	On	On			
Low frequency peripherals	On	On	On		
Full CPU and SRAM retention	On	On	On	On	
Power on Reset/Brown-out Detector	On	On	On	On	On

Figure 2.4: Energy Modes [5]

2.3.1 Energy Modes

Energy Mode 0

In EM0, When the CPU is running code from flash does not consume less than 180 µA/MHz. All peripherals can also be activated.

Energy Mode 1

In EM1, the CPU is sleeping and the power consumption is only 45 µA/MHz. The peripherals including, DMA, PRS and memory system is still available.

Energy Mode 2

In EM2 the high frequency oscillator is turned on, but with the 32 kHz oscillator running. Low energy peripherals (LCD, RTC, LETIMER, PCNT,WDOG, LEUART, I2C, ACMP) are still available, giving a high degree of autonomous operation with a current consumption of 0.9 µA.

Energy Mode 3

In EM3 the low-frequency oscillator is disabled, but there is still full CPU and RAM retention, as well as Power-on Reset and Brown-out Detector, with a consumption of only 0.6 µA. Even in this mode, the wake-up time is in the range of few microseconds.

Energy Mode 4

In EM4, the current is down to 20 nA and all chips functionality is turned on except the pin reset and the power on reset. All pins are put into their reset state.

2.3.2 Transition among sleep modes

The unit that manages the transitions among the energy modes, is called Energy Management Unit (EMU). As I described previously, the system has five energy modes and the transition among these modes can occur differently. Some parts of this section is taken from my Specialization project[7].

- A transition from EM0 to a low energy mode can only be triggered by software;
- A transition from EM1-EM3 to EM0 can be triggered by an enabled interrupt or event;
- A transition from EM4 can only be triggered by a pin reset or power-on reset;

An energy mode is selected by first configuring some control registers (EM4CTRL, EMVREG,EM2BLOCK) in *EMU_ CTRL*. If SLEEPONEXIT bit is 1, the transition into a low energy mode can optionally be delayed until the end of lowest priority Interrupt Service Routine (ISR). Figure (2.5) shows the transitions among energy modes.

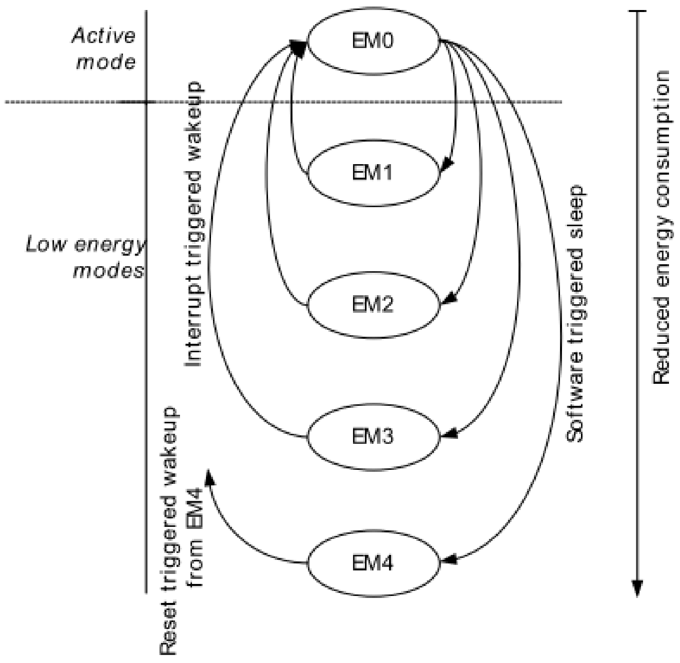

Active mode
Low energy modes
EM0
EM1
EM2
EM3
EM4
Interrupt triggered wakeup
Reset triggered wakeup from EM4
Software triggered sleep
Reduced energy consumption

Figure 2.5: EFM32 transitions among energy modes [5]

CORTEX-M3 processor is suitable for low-power designs and it has power-saving mode support (SLEEPING and SLEEPDEEP). The processor can enter in sleep mode using *Wait for Interrupt (WFI)* or *Wait for Event (WFE)* instructions. Essentially, CORTEX-M3 has separated clocks for essential blocks, so clocking

circuits for most parts of the processor can be stopped during sleep. The two sleep states (*SLEEPING* and *SLEEPDEEP*) provided by CORTEX-M3 are chosen setting SLEEPDEEP bit. This bit is situated in System Control Block(SCB) of Cortex-M3 System Control Register. If SLEEPDEEP bit is set to 0, then EFM32 chooses EM1 (SLEEPING), otherwise if it is set to 1 EM2 (SLEEPDEEP) is chosen.
In EFM32, EM3 is different from EM2 only because the low frequency clocks are stopped, but the CORTEX core is still in SLEEPDEEP.

WFI (Wait For Interrupt) To wake up the processor from WFI mode, the interrupt will have to be higher priority than the current priority level and higher than the level set by BASEPRI register or mask registers (PRIMASK and FAULTMASK). After the core is woken-up by WFI instruction, if the Priority Mask Register (PMR) is not set, the execution proceeds in ISR. Otherwise if PMR is set, the execution proceeds at the instruction after WFI[8].

WFE (Wait For Event) If the interrupt triggered during sleep has lower or equal priority than the mask registers or BASEPRI registers and if the SEVONPEND is set, it could still wake the processor from sleep. When the core is woken up by WFE instruction, execution proceeds at the next instructions of WFE. If the pending interrupt has high priority, then execution proceeds in ISR[8].

2.4 Timing in EFM32

2.4.1 Oscillators and peripheral clock

Most of the content of this section is taken from my Specialization project[7]. EFM32 supports the following oscillators:

- 1-28 MHz High Frequency RC Oscillator (HFRCO);
- 4-32 MHz High Frequency Crystal Oscillator (HFXO);
- 32 kHz Low Frequency RC Oscillator (LFRCO);
- 32.768 kHz Low Frequency Crystal Oscillator (LFXO);
- 14 MHz auxiliary RC oscillator (AUXHFRCO) used for flash programming and debug trace;
- 1 kHz separate RC oscillator ULFRCO (Ultra Low Frequency RC Oscillator). This oscillator is often used by WDOG Timer, it runs in EM3.

The system has different prescaler for *High Frequency Core Clocks* (HFCORECLK) and *Peripheral Clocks*(HFPERCLK). The core clock of the system is generated by HFXO or HFRCO. The differences are in terms of efficiency of wake-up time. The wake-up time of HFXO is around 400 µs whereas HFRCO only 0.6 µs. EFM32 always uses HFRCO. This clock is also used when it wants to wake up from EM2 or EM3 and when a reset occurs.

2.4.2 Real Time Counter (RTC)

Real Time Counter is used to keep track of time in low energy modes. In EFM32, RTC uses a 32 Khz oscillator that can run in EM2 consuming at most 0.9 μA. The features of RTC [5] are:

24-bit counter: The register where it is stored the value of counter is: RTC_CNT. The counter is incremented every clock cycle. When it exceeds the maximum value of RTC_CNT, it starts to counting again from 0.

Two compare registers (COMP0, COMP1): When in RTC_CTRL register the bit COMPT0TOP is set to 1. Then, the RTC count up to COMP0. If the bit COMP0 in RTC_IEN is set, then an interrupt is raised when the counter reaches COMP0. Furthermore, COMP0 and COMP1 can also be used as compare match. When the counter reaches their value, a device can be woken-up.

Clock Source: RTC clock source and its prescaler are defined in Clock Management Unit (CMU). The clock depends by prescaler and the RTC Frequency equation is

$$f_{RTC} = \frac{f_{LFACLK}}{2^{RTC\_PRESC}}$$

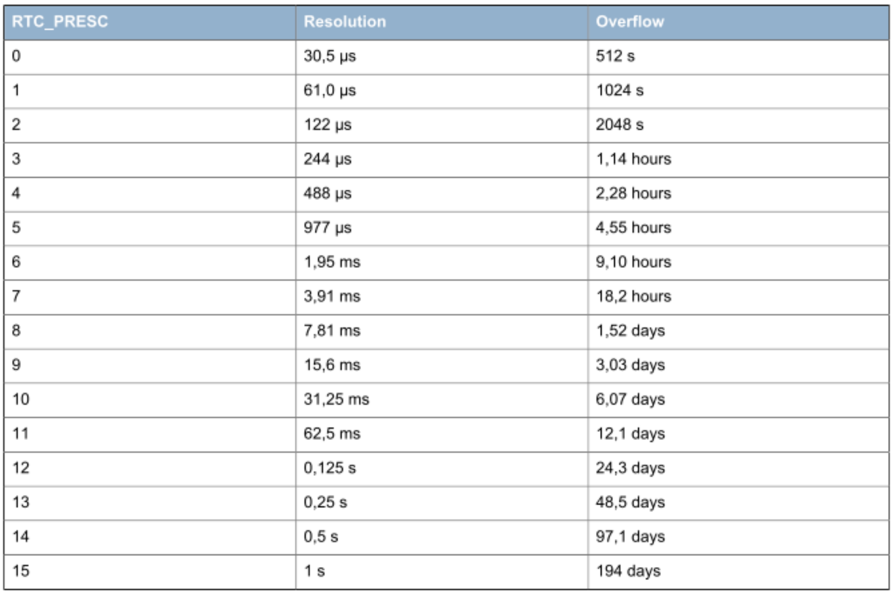
RTC_PRESC is a 4 bit value, while LFACLK has a frequency of 32khz. The resolution of RTC depends by the result of equation. The Figure 2.6 shows RTC resolution vs Overflow. Furthermore, in many cases, using prescaling can cause an increase of power consumption. If the clock is prescaled by four there is a major consumption of energy because the prescaler uses more power than RTC [5].

2.4.3 System Timer Cortex-M3

CORTEX-M3 has own unit to keep track of time. The counter of core (24-bit) is stored in *Current Value Register* and since in this case the counter is a basic countdown timer, then the value of register is decremented every clock cycle.
When the counter reaches 0, a *Reload Register* set again the value of *Current Value of Register*. The core clock can be generated either by HFXO or HFRCO (see section 2.4). The *System Tick* (SYSTICK) is generated according to the resolution (value stored in an internal register) of the core. The core generates a periodic interrupt (even when the system is in sleep mode) used by OS to keep track of time.

2.5 EFM32 Development kit

In this project I have used the EFM32 Development Kit provided by Energy Micro. THE DVK comes with a separate MCU board and prototyping board which are plugged into the motherboard. EFM32 DVK supports a USB connector that can be used to power the board and provide a live debug session to IDEs such as IDE



RTC_PRESC	Resolution	Overflow
0	30,5 µs	512 s
1	61,0 µs	1024 s
2	122 µs	2048 s
3	244 µs	1,14 hours
4	488 µs	2,28 hours
5	977 µs	4,55 hours
6	1,95 ms	9,10 hours
7	3,91 ms	18,2 hours
8	7,81 ms	1,52 days
9	15,6 ms	3,03 days
10	31,25 ms	6,07 days
11	62,5 ms	12,1 days
12	0,125 s	24,3 days
13	0,25 s	48,5 days
14	0,5 s	97,1 days
15	1 s	194 days

Figure 2.6: RTC resolution vs Overflow [5]

Embedded WorkBench. The Figure 2.7 shows MCU board, while Figure 2.8 shows the mainboard hardware layout of EFM32 DVK.

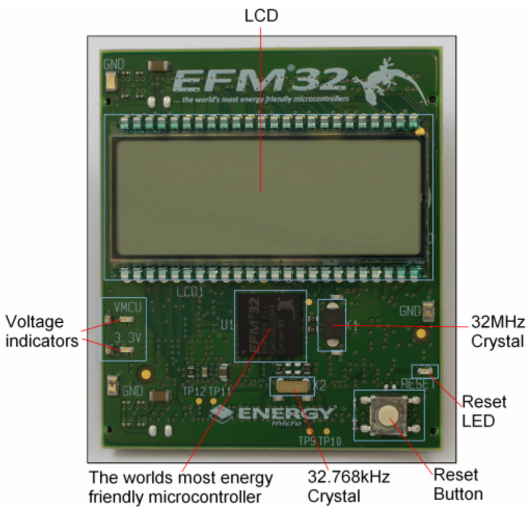


Figure 2.7: MCU board [9]

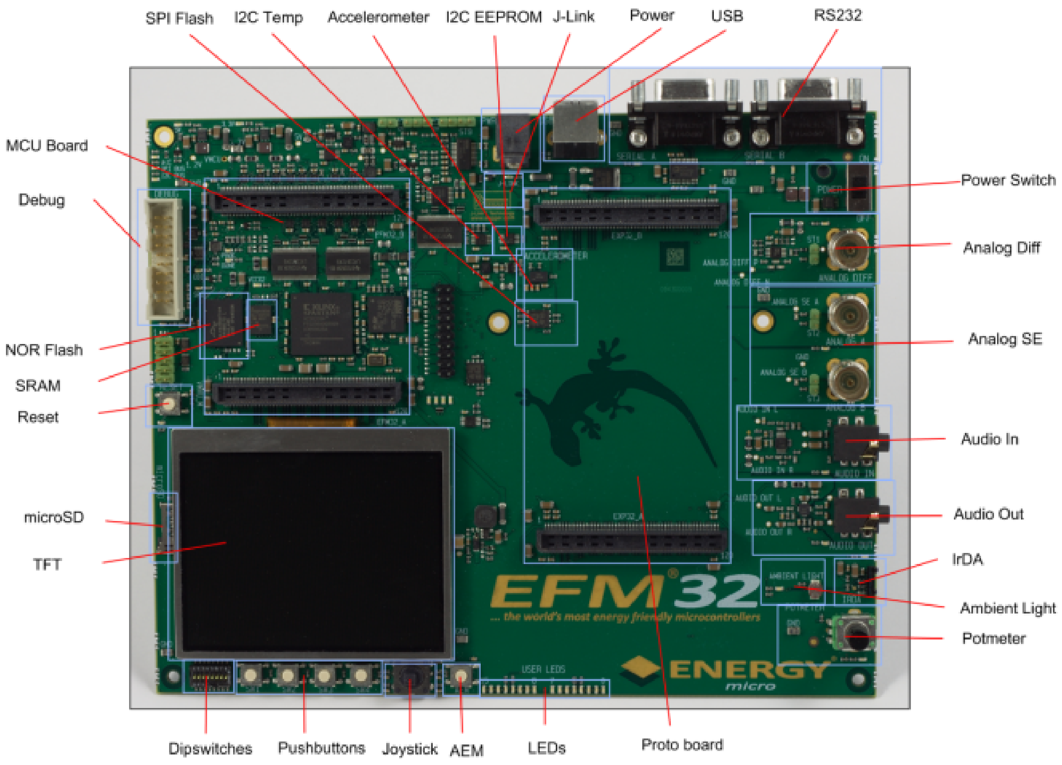


Figure 2.8: Mainboard Hardware layout [9]

2.6 About Energy Micro

Energy Micro is a Norwegian semiconductor company focusing on 32 bit microcontrollers with ultra low energy consumption. Energy Micro develops, markets and sells the world's most energy friendly microcontrollers, based on the industry leading ARM Cortex-M3 32-bit architecture. The company was founded in 2007 by experienced semiconductor professionals with previous expertise from Chipcon, Texas Instruments, Atmel and Nordic Semiconductor[4].

Chapter 3

FreeRTOS

"Last year, FreeRTOS was officially downloaded more than 77,500 times. This degree of popularity often surprises industry insiders, because FreeRTOS does not have a marketing budget and has not been featured in any industry surveys."

- **Richard Barry** as Guest editor for EETimes

3.1 About FreeRTOS

FreeRTOS is a free real-time operating system kernel for a small embedded systems. Since most of the code is written with C programming language (around 4000 lines of code), it has been ported to many different platforms.
FreeRTOS provides the following scheduling alorithm: *preemptive*, *cooperative* and *hybrid configuration options*. It supports the Cortex M3 Memory Protection Unit (MPU)and it is designed to be small (4K -9K), simple and easy to use. Further, it provides mechanisms for tasks to communicate and share data safely such as *queues*, *binary semaphores*, *counting semaphores*, *recursive semaphores* and *mutexes for communication* and *synchronization* between tasks. FreeRTOS supports both tasks and co-routine. In addition, FreeRTOS does not have restriction on the number of tasks that can be created no restrictions imposed on priority assignment.

3.2 FreeRTOS kernel

3.2.1 Tasks

In real time applications the workload is divided in many tasks. But since the processor can handle only one task per time, the system needs some policy to decide when use a task rather than another one. In this section is explained the main structure of a task and the list of tasks' state.

3.2.1.1 Task Control Block (TCB)

FreeRTOS provides for each task a data structures where are stored all the useful information about its context. Listing (3.1) shows the structure of a normal TCB. Nevertheless, upon creation of task, the system provides to create only those fields needed for a task (it depends on FreeRTOS' configuration) and discarding the useless ones.

```c
/*
 * Task control block.  A task control block (TCB) is allocated to each task,
 * and stores the context of the task.
 */
typedef struct tskTaskControlBlock
{
  volatile portSTACK_TYPE *pxTopOfStack;    /*< Points to the location of the last item placed on the tasks stack.  THIS MUST BE THE FIRST MEMBER OF THE STRUCT. */

  #if ( portUSING_MPU_WRAPPERS == 1 )
    xMPU_SETTINGS xMPUSettings;       /*< The MPU settings are defined as part of the port layer.  THIS MUST BE THE SECOND MEMBER OF THE STRUCT. */
  #endif

  xListItem       xGenericListItem; /*< List item used to place the TCB in ready and blocked queues. */
  xListItem       xEventListItem;   /*< List item used to place the TCB in event lists. */
  unsigned portBASE_TYPE  uxPriority;     /*< The priority of the task where 0 is the lowest priority. */
  portSTACK_TYPE      *pxStack;     /*< Points to the start of the stack. */
  signed char       pcTaskName[ configMAX_TASK_NAME_LEN ];/*< Descriptive name given to the task when created.  Facilitates debugging only. */

  #if ( portSTACK_GROWTH > 0 )
    portSTACK_TYPE *pxEndOfStack;     /*< Used for stack overflow checking on architectures where the stack grows up from low memory. */
  #endif

  #if ( portCRITICAL_NESTING_IN_TCB == 1 )
    unsigned portBASE_TYPE uxCriticalNesting;
  #endif

  #if ( configUSE_TRACE_FACILITY == 1)
    unsigned portBASE_TYPE  uxTCBNumber;  /*< This is used for tracing the scheduler and making debugging easier only. */
  #endif

  #if ( configUSE_MUTEXES == 1 )
    unsigned portBASE_TYPE uxBasePriority;  /*< The priority last assigned to the task - used by the priority inheritance mechanism. */
  #endif

  #if ( configUSE_APPLICATION_TASK_TAG == 1 )
    pdTASK_HOOK_CODE pxTaskTag;
  #endif

  #if ( configGENERATE_RUN_TIME_STATS == 1 )
    unsigned long ulRunTimeCounter;   /*< Used for calculating how much CPU time each task is utilising. */
  #endif
} tskTCB;
```

Listing 3.1: Task Control Block (TCB)

The main drawback of using tasks is due to the huge amount of space occupied in memory. In fact, for each task is allocated a TCB structure.

3.2.1.2 Task Lists

In FreeRTOS the tasks are event-driven. This means that a task process its work only when an event occurs like expiring of a timeout or synchronization events (e.g. queues, semaphores etc). For this reason, the kernel must guarantee for each task that is being scheduled to the right time and without any delay. Since no more than one task can be scheduled in the same time, the kernel needs to save the context of those tasks not running. Thus, the kernel needs to handle more than one state. The Figure 3.1 shows the interactions among these states.

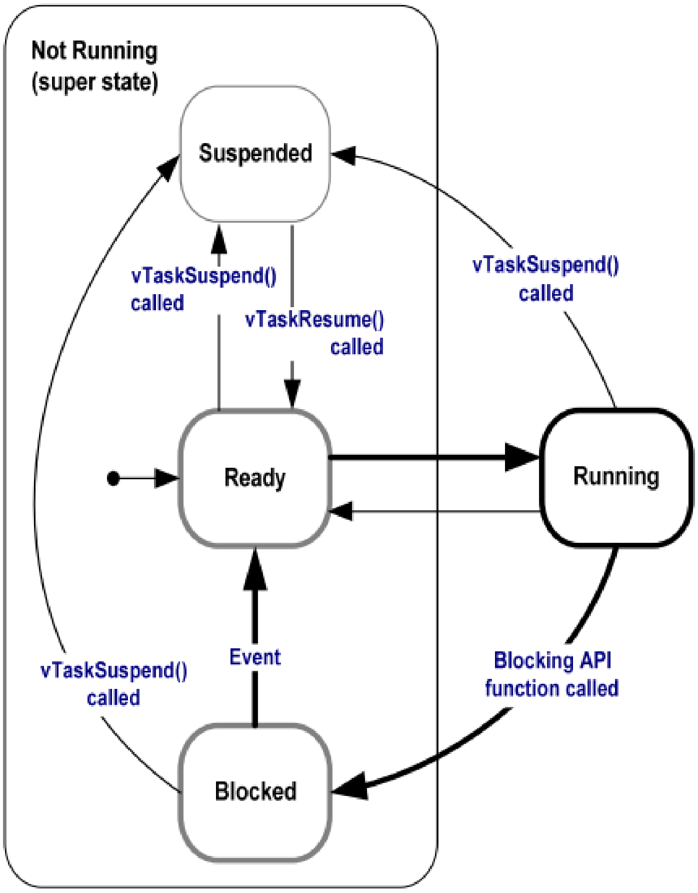


Figure 3.1: Task state transition [10]

FreeRTOS implements states using a list of items. Lists implemented in FreeRTOS are:

READY LIST: The tasks in this list are able and ready to run, but are not in running state because their priority is lower than priority of current task running. For every priority there is a list of tasks (more tasks can have same priority) ready to run *ReadyTasksList[n]*.

TERMINATING LIST: This list contains those tasks that have been deleted but their memory has not been freed, yet. Name of list is *TasksWaitingTermination*.

SUSPENDED LIST: When the kernel invokes the function to suspend task (or all tasks), the tasks are added in *SuspendedTaskList*. If the kernel wants to resume a task/s from a suspended list, resume function is called.

PENDING LIST: When the scheduler is suspended, some tasks can expire their timeout and they need to be moved in ready list, but since these tasks can not be moved in ready list are temporarily placed in *PendingReadyList*. They will be moved to the ready list when the scheduler is resumed.

DELAYED LIST: This list contains the tasks that are delayed for a while. The tasks in this list are sorted by wake up time. The name of list is *DelayedTaskList*.

OVERFLOW DELAYED LIST: This list contains those tasks that endure an overflow in wake up time. In fact, when a delay is added for a task, might occur that the sum of current time and delay time can involve an overflow. The name of list is *OverflowDelayedTaskList*. For example, if the system use a counter of 9 bits, then an overflow occurs every 512 ticks. If a task issues a request of delay of 20 ticks and the current tick number is 505, then the task is added in overflow list with a delay of 14 ticks. When the system raises an interrupt due to tick overflow, then the content of *OverflowDelayedTaskList* and *DelayedTaskList* is swapped.

3.2.1.3 Idle Task

The processors always needs something to execute, therefore must be at least one task to be scheduled. An idle task is automatically created by scheduler at the beginning of program, in this way the processor is held busy over all of time. The idle task has the lowest possible priority, in this way it never prevents a higher priority application task from entering the running state. Furthermore, the idle task is responsible for releasing resources not longer used by other tasks. The idle task has a role very important inside a RTOS. The usage of idle task is well explained in next chapters.

3.2.2 Co-routines

Co-routines are often used in those cases when there is a strong need to save the amount of memory (very small microcontroller) occupied by tasks. In fact, co-routines are conceptually similar to tasks but with a main limitation of the stack

Tasks	Priority	Period
Task 1	2	10 ms
Task 2	7	20 ms

Table 3.1: Tasks

usage. Besides, they share the same stack unlike tasks that have an own stack. The consequence in this case is due to losing the content of a variable whenever a routine is blocked. Additionally, co-routine use prioritized cooperative scheduling. These routines are managed in the same way of tasks and they use the same set of lists.

3.2.3 Time management

In FreeRTOS, the time is handled using the Systick module provided by CORTEX-M3 (see section 2.4.3). The core, triggers an interrupt according to the frequency set in FreeRTOS. Usually, the rate frequency of this interrupt is around 1000Hz (1ms) or 100Hz (10ms), this is called *tick rate*. FreeRTOS measures time as number of ticks, where each tick corresponds to an interrupt raised by Cortex core. For this reason, FreeRTOS saves the number of ticks in its internal variable called *xTickCount*. The system provides two types of this variable, one with 32bits and another one with only 16 bits. The following equation shows how often the overflow occurs.

$$Time_{sec} = \frac{2^N * resolution}{sec\_one\_day}$$

Where N is the number of bits, $resolution$ is expressed in seconds and one_day_sec contains the number of seconds in one day, respectively 86400.
According to the equation shown above, using a variable of 32 bits with a resolutions of 1ms, an overflow occurs approximately every 50 days. While, if the resolution is 10ms, the overflow occurs every 500 days (less than 1 and half year). It is important to remark that every time an overflow occurs, FreeRTOS swaps the lists of *DelayedList* with *DelayedOverflowList* (see section 3.2.1.2).
Moreover, when the system receives an interrupt, it updates the xTickCount variable. After that, it checks if a task with higher priority is present in *Ready List*. If any, it performs a context switching with the new ready task. The role of xTickCount is extremely important for the correct functioning of FreeRTOS. If some interrupts are neglected, the system might miss the deadline of some tasks. An example how FreeRTOS schedules tasks, is shown in Figure (3.2).

The table 3.1 contains the list of tasks with corresponding deadline and priority. For the sake of simplicity, the two tasks are always in ready list and the duration

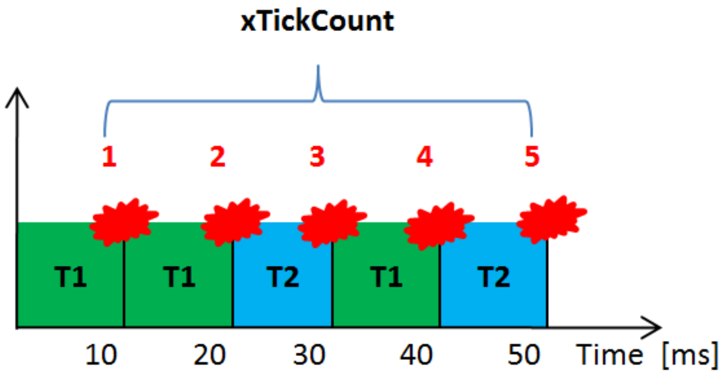


Figure 3.2: Time management in FreeRTOS

of each task is only a tick timer (10 ms). $Task_1$ occurs every 10 ms with a lower priority than $Task_2$ that occurs every 20ms. A tick rate arises every 10 ms. Upon interrupt(red stain), FreeRTOS updates the value of xTickCount variable and it checks if a new task with higher priority is ready in Ready List. If any, it performs a context switched with the new task. In the example, the context switching is performed only two times (20ms and 40ms when the task 2 is ready to be executed). This example gives only an idea how FreeRTOS handles its internal time management. Next chapters discuss with all possible drawbacks of this model. Moreover, the number of clock cycles occurring between two ticks is strictly dependent by the oscillator's frequency. If the oscillator is 10MHz, then a clock cycle occurs every 10 µs, therefore each tick interrupt employs 10000 clock cycles.

The previous example roughly illustrates the way as FreeRTOS organizes tasks and timing. The next following chapters propose other possible approaches of using timing module. It is necesary to analyse with more details how FreeRTOS interfaces its kernel with CORTEX-M3. All the concepts introduced so far, are still valid, but it is not clear how FreeRTOS performs a context switching and how it handles exceptions (systick). The exception are asynchronous events and they can occur any time, therefore FreeRTOS needs to keep the kernel protected when is being interrupted. It must also be able to resume the system from the blocked instruction. Thereafter, it makes sure that the system is running the highest priority task without missing other important deadlines.
A context switching is also dependent by external exceptions. The system can perform a context switch in two cases, when the *tick timer expires* or if after an interrupt a *task with higher priority* becomes the new running task. The core internally handles all external interrupts and context switching. In particular, the context switching is perfomed by an exception called *pendSV*. The unique burden of FreeRTOS is to schedule interrupts and tasks with a well defined priority. Basically, exceptions have higher priority than any tasks. FreeRTOS configures (some

exceptions have fixed priority) the priority of systick timer and pendSV, as the *lowest priority interrupts*. Thus, any exceptions have higher priority than systick and pendSV but they have higher priority than tasks.
Figure (3.3) shows an example how exceptions and tasks are handled in FreeRTOS. In the example, Task A has higher priority than Task B. The systick handler always issues a request of context switching, hence, after its execution a pendSV is always performed. This allows to run the highest priority task every time the system is being resumed.

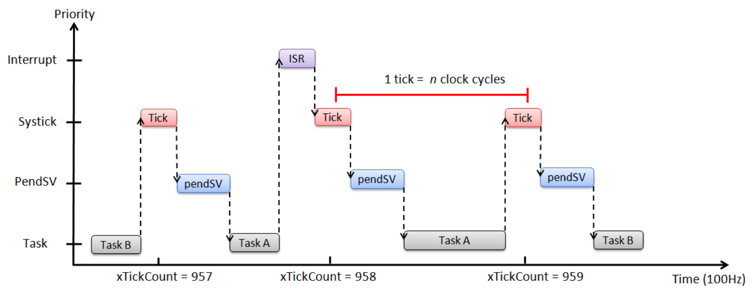


Figure 3.3: How exceptions and tasks run in FreeRTOS

Initially, Task B is running and is being interrupted by a systick exception. Since the handler issues a request of context switching, FreeRTOS checks if a task with higher priority is ready. FreeRTOS switches to Task A and it continues to run until an ISR occurs. In the meanwhile that the ISR is executed a systick exception also occurs, hence, the systick is being performed and a context switching is executed. The system keep running mantaining the same rules. FreeRTOS updates the xTickCount variable every tick exception.

3.3 Example Demo Project

FreeRTOS provides a demonstration project for each platform supported. The demo is a ready application that can be run on the development kit of vendors without warnings or errors. The aim of demo project is to help the customer to be familiar with FreeRTOS environment and getting started with it.
The demo project provided by FreeRTOS is an application that works with LEDs and LCD.

3.4 License and Platform supported

FreeRTOS is licensed under the GNU General Public License (GPL), with an exception. If FreeRTOS is linked to other independent modules using only the FreeRTOS API interface the code can be distributed under different licenses than GPL. This exception makes it possible to use FreeRTOS in commercial applications without

paying any royalties.
FreeRTOS is ported in many platforms, officialy support 23 architectures (counting ARM7 and ARM Cortex M3 as one architecture each). A list of the currently supported architectures are: *Altera, Atmel, Cortus, Energy Micro, Freescale, Fujitsu, Luminary Micro, Microchip, NEC, NXP, Renesas, Silicon Labs, ST Microelectronics, Texas Instruments, Xilinx, x86*(real mode).
For each supported platform, the code includes a demo project demonstrating how to use the code on that specific platform. Some parts of this section is taken from my Specialization project[7].

Chapter 4

Time Management in a RTOS

4.1 RTOS overview

A real-time system must satisfy bounded response-time constraints, otherwise it can produce catastrophic consequences. Real-time systems are classified as *hard*, *firm* or *soft* systems.

Hard real-time Are those systems where the failure of a response-time constraints lead producing serious risk of the whole system (e.g. death of people, nuclear reactor, control systems).

Firm real-time Are those systems with hard deadlines, but where a certain low probability of missing a deadline can be tolerated (e.g. food processing plant system).

Soft real-time Are those systems with soft real time constraints which require time execution of tasks at coarse temporally resolution, but missing a deadline do not cause catastrophic event.(e.g. vending machines)

Time management unit has an important role in a RTOS because it guarantees that tasks are scheduled on the right time. In some systems, the Time Management Unit represents an issue to overcome. In fact, this unit involves an increase of power consumption introduced in the whole system. In this section are discussed those techniques that reduce the bad effects of Time Unit Management.

4.2 Periodic Interrupt Timer

Usually, operating system implements time management using a periodic interrupt triggered by hardware timers. Periodic interrupts are implemented using hardware clock interrupts called *ticks* (see section 3.2.3). The periodic interrupt approach has several limitations in terms of power consumption and efficiency. For example, if the programmer provides a bad driver, then the system risks to spend a long time in its functions. Therefore many tick interrupts are being lost [14]. Thus, a task can miss its deadline. Possible drawbacks of periodic interrupt, are:

Wasting energy in workless timer ticks Also in idle state, the tick periodically wakes the processor up and cause the execution of handler. In embedded system the idle state sometimes takes a lot of time (e.g. hours or even days) and this cause a huge amount of energy wasted [17].

Resolution accuracy A further issue about periodic interrupt is due to the resolution of systick timer. In fact, the system usually wakes up every 1/10 ms and this can become a limitation if an application requires a lower resolution. The value to set as resolution time is a bit tricky because if the resolution is too high, the number of ticks increase, then also the energy consumption rise. But if the resolution is too low the system lose time efficiency[13].

Overhead inflicted by Hardware interrupts The tick interrupt introduces a further overhead even when the system is performing a task. Indeed, the job execution is interrupted every tick rate and the system endures repeatedly preemption between job and interrupt handler. The overhead introduced on a serial program depends from frequency and number of processes. The performance of the system decreases around 1% - 1.5% [16]. The whole overhead introduce in the system is classified as *direct* and *indirect* overhead. The direct factor is the time to perform the context switching and running instructions inside the handler routine. While the indirect overhead is the difference between the total overhead and the direct one. Find a way to determine the amount of the indirect overhead is a tricky job, this value should be attributed to external effects. Furthermore, since a task is interrupted many times by tick interrupts, this causes a growth its time execution. This factor must to be kept into account when the system works with strictly deadline environment. Exceeding a deadline can compromise the performance of the system.

Nowadays, most of the common operating system are adopting different approaches to cope issues presented above. One of the way to overcome this problem is to adopt a *totally tickless system*. Moreover, in order to guarantee a better resolution of tick timer a *hrtimer* representation has been implemented in some Operating System[15]. This new approach is used by modern platforms. Currently, some embedded system uses hrtimers representation. Since in FreeRTOS the time is represented only with an internal variable, I thought to keep using this representation and disregard the htimers support. This choice is due to the fact that otherwise FreeRTOS needs to be redesigned.

4.3 Totally Tickless kernel

Originally, Linux kernel (from 2.6.13) used an implementation similar to the approach used by Martin Tverdal[3]. In the latest versions, Linux has implemented

a totally tickless system. I want to draw briefly the approach used by Martin Teverdal in his thesis. He removed the tick interrupt only when the system was in idle state. Additionally, his approach provide to move the core unit in a sleep state when there were not tasks to schedule. In this case FreeRTOS benefits of a further performance improvement but still does not solve the drawbacks explained above. In *Totally Tickless kernel* in spite of generating an interrupt every tick rate it generates a new tick only when a next event is ready to be processed. This approach allows to wake processor up only when it needs to be awake and not every time. Hence, this strategy guarantees best performance and it reduces the effect of drawbacks analyzed above. Using a totally tickless kernel, the scheduler can also schedule tasks with a lower time resolution and never interrupt the execution of a task. Even though this new solution can represent a good approach to overcome the issues described above, the tickless system introduces a further problem. In fact, the kernel introduces an additional overhead to compute the next events to perform in the near future. The experiments in [13] show that this approach is still better than previous one.
To implement a totally tickless kernel the OS still needs to use a time counter, but not the timer embedded in core unit. A possible timer to keep track of time in FreeRTOS is the RTC unit.

4.4 Tickless RTOS on the market

Most RTOS for small microcontrollers are able to run on many different devices including wireless sensor nodes (WSN), but their general approach does not exploit the energy saving capabilities of modern MCUs. Using this strategy allows WSNs nodes to have the longest possible runtime such as months or even in years. Nowadays, the market offers a limited amount of RTOS able to deal with these requirements. Some new OSes like *TinyOS* [19] and *Contiki* [18] follow an event-driven approach but still they do not guarantee all functionality provided in most common OS like FreeRTOS.
A totally tickless strategy has been adopted by famous desktop OSes but not in RTOSes. Linux provides even in embedded system a distribution that includes these features, but it is not always possible to apply Linux in WSN nodes. By the way, a new RTOS equipped with totally tickless system is *TiROS (Tickless Real-Time Operating System)* [20]. TiROS is a pre-emptive priority based real-time task scheduler for embedded systems with limited memory resources. It was developed at Sandia National Laboratories and is released as open-source. It is more closely comparable in resource usage to FreeRTOS. A proprietary RTOS that provides a tickless system is *QKernel* [21] developed and deployed by *Quasarsoft*.
Recently, RTOSes like *FreeRTOS* and *eCOS* are growing a lot and, extremely, they require a support to handle power management in WSN nodes and tiny microcontrollers. As already explained above, the aim of this thesis is to introduce in FreeRTOS an energy aware support for as many platforms as possible.

Part II

Methodology

Chapter 5

Energy Management Framework

"Demand for power management chips continues to grow, and our end-equipment customers want innovation that saves energy. We have a small but significant presence in Cork working on innovation that gives our customers access to products that solve any energy management issue."
- **Steve Anderson**, Senior Vice President of TI's Power Management Business

Recently, most of MCUs available on the market, provide a support to manage different sleep modes inside the microcontroller itself. Nowadays, only few proprietary RTOSes, offer a support to handle sleep modes of MCUs. Thus, the aim of this chapter, is to propose and present a module that accomplishes the fulfillment of these needs in FreeRTOS.
The chapter has been organized with the following structure: section (5.1) gives an overview regarding reasons why it is useful to build a new *Power Management Framework* (PMF), section (5.2) proposes some examples of issues arise when a programmer tries to use the energy features provided in modern MCU. Section (5.3) shows three possible different solutions to develop a new PMF, focusing on *pros* and *cons* of each approach. Section (5.4) explains the chosen solution and it also shows its functioning. In section (5.5) there is a detailed explanation how make a porting of new architecture using the proposed PMF. Afterwards, section (5.6) explains the *porting* made for EFM32. The last section (5.7) summarizes which goals have been fulfilled in this chapter.

5.1 Problem description

There are several reasons why it is useful to embed a Power Management Framework in a RTOS, such as:

- Using an high level framework allows to handle internal sleep modes[1] in efficient manner.
- the MCU can save a huge amount of energy during all tasks. Tasks can lead automatically the MCU in the right sleep mode whenever an idle task occurs;
- Achieve power friendly applications with a minimal effort;
- Currently, power management is only provided in some proprietary RTOSes, such as AVIX-RT [22], CMX-SYSTEM [23] and IAR-SYSTEM [24];

The power management framework introduced in this chapter is a continuation of the work done by Martin Tverdal in his thesis [3]. On the contrary, the new framework introduces a more robust approach to handle sleep modes for EFM32. Indeed, in this case, the system is also able to manage issues regarding management of sleep modes in ISRs. The porting made for EFM32 is more full-bodied because it guarantees that the energy modes are only used by PMF.

5.2 Issues of sleep modes with ISRs

Unlike tasks, the management of interrupts is much more difficult, hence, this requires to undertake a deep research in those cases when power modes are used. This section shows issues involved in the system when the Power Management Framework neglects to use power modes (see section 5.2.2) inside interrupts. Besides, this section shows also an example of misleading use (see section 5.2.1) of the framework developed by Martin Tverdal.

5.2.1 Troubles involved using only a Task Power Management Framework

This section presents those drawbacks that arise, when the system uses only a Task Power Management Framework (TPMF)[2]. The framework developed by Martin Tverdal [3] manages sleep modes using only TPMF. I think that this is the main drawback of his approach.
Adopting this strategy, the system is subjected to meet some undesired situation, such as: use of not ideal sleep mode for idle task (therefore, wasting more energy than predicted) or move the MCU in a wrong power modes (therefore, block the system and compromise its functioning).
Before getting started with the explanation of Figures (5.1,5.2), I would highlight some useful and important points to understand the meaning of the next examples. For the sake of simplicity, I decided to report on y-axis only those power modes

[1]Sleep mode allows to save a consistent amount of energy in the mean while that MCU is running. Besides, in order to achieve a further saving of energy, each energy mode provides to shut-down unecessary peripherals. The differences among sleep modes is linked to the number of peripherals turned off and the amount of energy saved.

[2]This Framework provides to handle the requests coming from tasks to stay in a sleep state. TPMF does not manage requests coming by ISRs.

present in CORTEX-M3, such as *Running*, *Sleep Mode* and *Deep Sleep Mode*. On the other hand, the x-axis represents how tasks and ISRs are scheduled over time. When there are not tasks to schedule, the system arises an idle task where the system can move the MCU in one of the power modes. The color of each task and ISR is the power mode where they can stay after their executions. Respectively, *green* for sleep mode and *blue* for deep sleep mode. In some examples, the request to stay in a sleep mode might last long until another event occurs. Finally, the next fourth examples make use of two handlers, called *ADC_Handler* (used to sample the output signal of a joystick) and *USART_Handler* (used to send data on a serial cable, such as RS232). The administrator is the entity that manages the requests coming from tasks and ISRs and it also decides which energy mode to use during idle task.

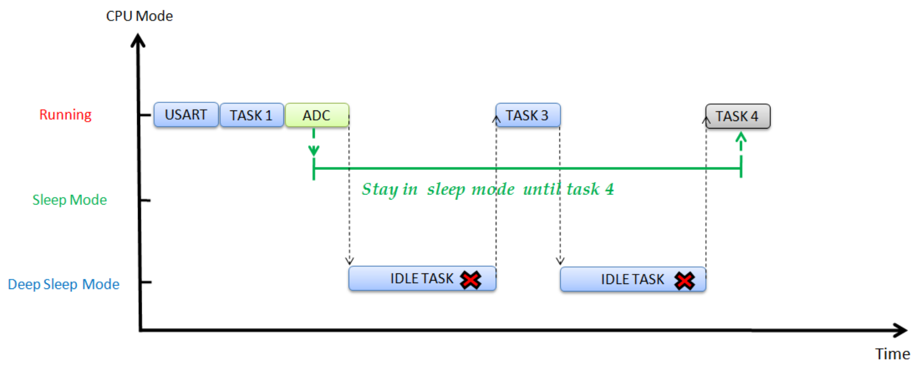


Figure 5.1: Wrong use of Power Management framework without considering sleep modes inside ISRs

In Figure (5.1) is shown how the system behaves when it does not consider the necessity of an ISR to stay in a sleep mode after its execution. This situation, involves in a misleading use of sleep mode inside an idle task. In fact, the administrator chooses a wrong power mode for both idle tasks.
Initially, an USART_Handler is being performed by MCU. Let's suppose that after its execution, it does not allow to go in a power modes lower than 2 (because it needs to use another peripheral that can not work only in mode 3 or 4). After that, $Task_1$ is scheduled, during its execution it receives a request by programmer to use sleep mode 2 for future idle tasks (if any). Afterwards, the system receives another interrupt and it executes the ADC_Handler. The constraint of this interrupt is to stay in sleep mode 1 until $Task_4$ occurs (it needs to wait for an answer from a device that runs in mode 1).
When the idle task occurs, the administrator has to provide the ideal sleep mode for it. Since the administrator handles only the requests coming from tasks, it moves the system in mode 2. Actually, the system needs to run in mode 1 because this was a request coming by ADC. Here, the system can enter in a undesired state

and it might compromise its functioning. If the system manages this issue (hence, if the wrong mode did not turn off the MCU),$Task_3$ is executed. After that, an idle task occurs and it meets again the same problem. After MCU schedules $Task_4$ (if the system overcomes the previous traps) it can use mode 2 for future incoming idle tasks.
Whereas, the second Figure (5.2) shows how the system would behave using

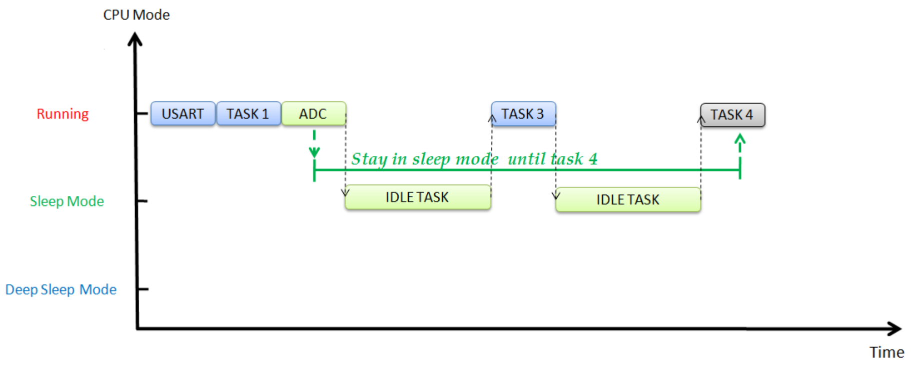


Figure 5.2: Correct use of Power Management Framework with sleep modes issued inside ISRs

a framework that also deals with ISRs' constraints. Now, the administrator can choose the most suitable sleep mode for incoming idle tasks.
The example shows that now the system can communicate the right power modes for both idle tasks. Unlike the first example, now the ADC_Handler issues a request to stay in sleep mode 1 until $Task_3$ instead of $Task_4$. This involves to put the second idle task in power mode 2 instead of 1 (as needed in first example).

5.2.2 Energy losses without using sleep mode inside ISRs

This section aims to show which are the advantages of adopting a management of power modes for ISRs. The use of sleep modes can achieve good results if each idle task is pushed in its ideal power mode. If the idle task uses a wrong sleep mode the system can have two possible problems, such as: waste energy or enter in a *jeopardizing state* (as shown in previous examples).
Figure (5.3) shows how the system behaves without using sleep modes. Moreover, the example outlines the case when nested ISRs (FreeRTOS manages nested ISRs) occur and how the system handles this burden. In fact, before using nested ISRs is necessary to understand which are the possible consequences involved in the system switching among ISRs. If the nested ISR performs a request to stay in a different sleep mode, this must be take into account if we want to preserve the system from the problems described above.

Initially, *USART_Handler* is performed until an ADC interrupt occurs. Since the latter has a higher priority respect to first handler, *ADC_Handler* is nested (it

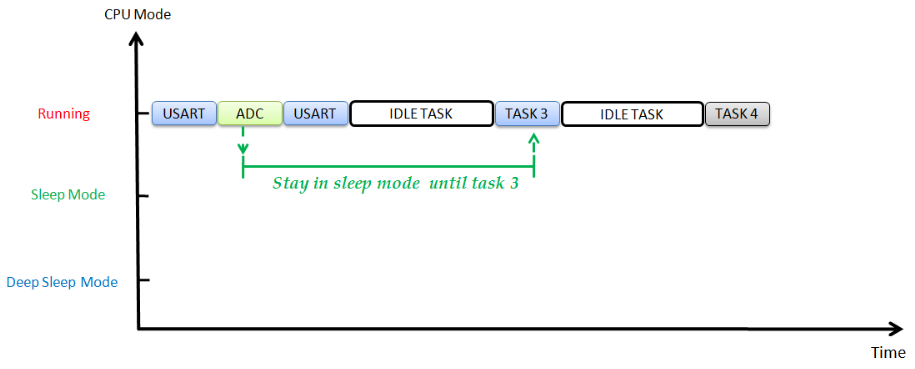


Figure 5.3: Behaviour of the system without using power modes provided by MCU

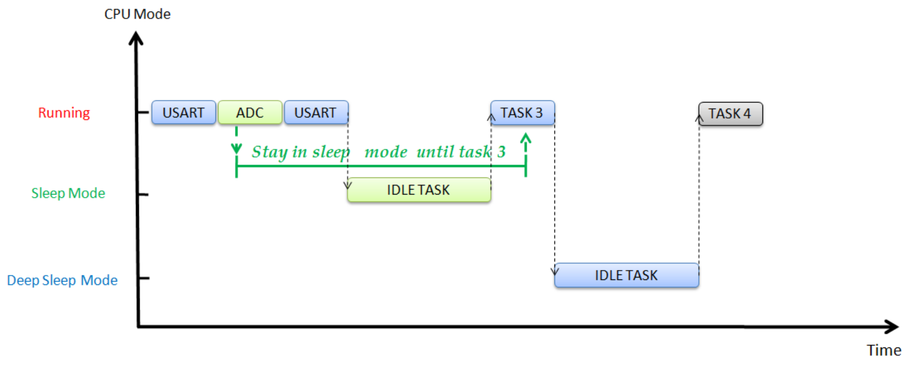


Figure 5.4: Behaviour of the system using power modes provided by MCU

needs to stay in power mode 1 until $Task_3$ occurs) and it performs its instructions. After that, the MCU return to *USART_Handler* (it needs to stay in mode 2 after its execution). An idle task occurs and the administrator does not have any idea about which power mode is needed to use for all tasks. It runs the idle task in *running mode* without saving energy. Actually, the system could run in power mode 1 saving some energy.
Afterwards $Task_3$ occurs and the *ADC_Handler* does not need anymore to stay in mode 1. When the next idle task occurs, the administrator has the same doubts as before and it leads to run the *idle task* again in *running mode*. Actually, now the system could run in mode 2 instead of 1 (other energy is being wasted) . At the end $Task_4$ is executed and it does not affect the power management.
Figure (5.4) shows how the previous example would behave respecting the requests (sleep modes) done by tasks and ISRs. It is important to remark that since idle tasks occurs periodically, the energy saved in each period is replicated for the whole system execution.

	Limited memory	**List**	**Array**
Footprint	*low*	*medium*	*medium*
Overhead	*high*	*high*	*low*
Data Structure	*simple*	*complex*	*simple*

Table 5.1: Evaluation of proposed solutions

5.3 Proposed solutions

In this section are being proposed some possible solutions to manage the sleep modes in FreeRTOS. The parameters used to evaluate the complexity of each solution are:

- Space occupied in kernel footprint;
- Overhead introduced in the system;
- Data structure's complexity;

All solutions require that users must define the sleep mode to go in idle task directly inside the body of task or ISR. Other possible ways have been thought, but according to the reasons explained in the fall project [7] they have been discarded. All proposed solutions change the structure of TCB (for more details see section 3.2.1.1) created for each task. Thus, the power management framework uses the TCB to store the sleep mode of each task. Then, inside the TCB it has been added a new field called *ucSleepMode*. This variable keeps track of the current sleep mode of that task and it is constantly kept updated.
Listing (5.1) shows the code added in FreeRTOS to insert the new field inside TCB.

```
#if ( configUSE_POWERMANAGER == 1 )
   unsigned portCHAR ucSleepMode; /*< Used to store the sleep mode of each task */
#endif
```

Listing 5.1: New Task Control Block (TCB) with sleep mode field

In Table (5.1) are illustrated the three solutions (*List memory*, *List* and *Array*) explained in the following sections. According to the three factors highlighted above, it is possible to understand the possible impact of these solutions. In next sections (5.3.1, 5.3.2, 5.3.3), there is a more detailed explanation about the functioning of all of them.

5.3.1 Limited memory solution

This first solution has been thought to use as less space as possible. Indeed, the solution provides to store the sleep mode of each task in own TCB. This value is stored in a 8 bit variable, but only 4 bits are used, while the other 4 bits are reserved for a future use.

Furthermore, the system keeps track about the lowest sleep mode using a further 8 bits variable stored inside the Power Management. The purpose of this variable, called *LSleepMode*, is to store the lowest sleep mode according to the requests received by all tasks. In practice, if a task requires to stay in sleep mode 1 and another one in sleep mode 2, this variable contains sleep mode 1, because there is a task that needs to stay in a higher level.

The functioning of power framework is reported as follow.

1. When a task is being created inside the variable *ucSleepMode* of TCB is copied the default sleep mode (it is the highest possible sleep mode);
2. The *LSleepMode* is kept updated every time a new task is being created, removed or it changes its sleep mode;
3. Whenever there is an idle task, the system reads the value of *LSleepMode* and it uses this value to move the MCU in the right sleep mode;
4. If a task is being deleted, the system provides to keep updated the value of $lowestSleepMode$;

Figure (5.5) illustrates a possible scenario of limited memory solutions with three tasks. For the sake of simplicity, the example works only with two sleep modes such as sleep mode 1 (green square) and sleep mode 2 (blue square). In the example are shown only tasks, without reporting their deadlines and periods (these assumptions are also valid for other examples reported in following sections).
In Figure (5.5) the requests generated by the programmer are redirected to the administrator, then the programmer does not know what is going on inside the PMF.
The first three steps create respectively $Task_1$, $Task_2$ and $Task_3$. After their creation the *LSleepMode* is equal to 1 because creation implies to initialize TCB with sleep mode 1. After that, programmer sends a request to $Task_3$ to change its sleep mode from 1 to 2 but this does not affect the value of *LSleepMode* (still sleep mode 1). Right away, an *idle task* occurs and administrator communicates that it can move in sleep mode 1. Afterwards, the programmer issues a request of removing $Task_1$ and $Task_2$. The administrator changes the value of *LSleepMode* with sleep mode 2 only after removing $Task_2$, because the only task available is $Task_3$ (sleep mode 2). Finally, the new *idle task* can run in sleep mode 2.

5.3.1.1 Advantages

The good thing of adopting this kind of solution is to use a limited amount of memory. In fact, with n tasks, the memory wasted in the whole system is:

$$Space\_used = (n + lowestSleepMode) * (8bits)$$

Additionally, the complexity of data structure is very low, because there are used

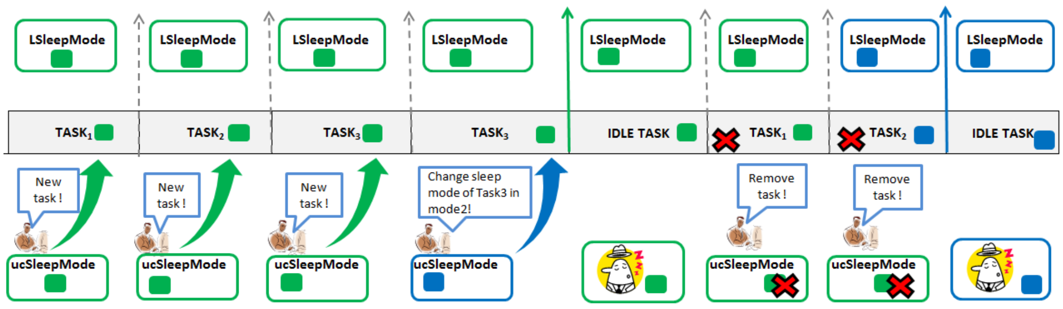


Figure 5.5: Functioning of limited memory solution

only two variables, one to track the lowest sleep mode and another one for the TCB of each task.

5.3.1.2 Shortcomings

The main drawback of this solution is due to the fact that there is a considerable overhead in the whole system. Every time a task is deleted or changes its sleep mode, *lowestSleepMode* must be kept updated. A different use of this variable might check the sleep modes of all tasks only when there is the need, (idle task). But, even in this case, the system introduces an high overhead, because all tasks must be checked, this means that the PMF must go through all TCBs' tasks.

5.3.2 List sleep mode solution

The second solution uses an array of sleep modes. For each position (therefore for each sleep mode) there is a list of those tasks that requested to stay in. This idea comes out from looking the way how FreeRTOS manages the events and task in its kernel. The functioning of power framework is very simple, every time there is a new task, this is added to the list of sleep mode where it belongs to. If a task is deleted, it is removed from the list where it belongs to.

The Figure (5.6) shows an high level abstraction of the interactions among tasks and sleep modes. There are two possible kind of requests, one coming by programmers and tasks and another one by idle tasks. Whenever an idle task occurs, the administrator searches inside the array of lists the most suitable sleep mode for idle task. It is also responsible to update the data structure every time a task changes its context. The data structure consists of an array where in each position (sleep mode) there is a list of those tasks belonging in that sleep mode. It is important to remark that tasks are ordered as list of their pointers. Respectively, $Task_2$, $Task_3$ and $Task_5$ are in sleep mode 1. While, $Task_4$ in sleep mode 2 and $Task_6$ in sleep mode 3. For the reasons presented in the following section (5.3.2.2) it has not been proposed any example about a possible functioning case of this idea.

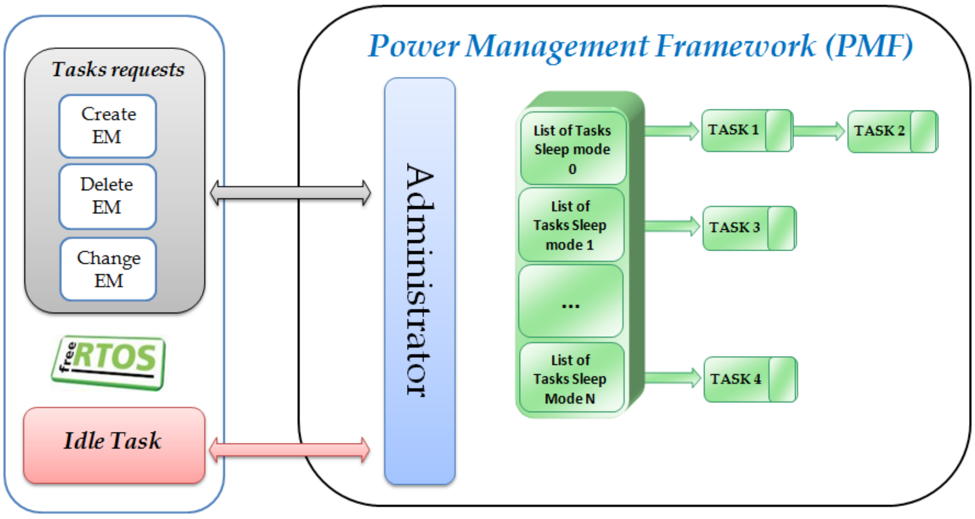

Tasks requests
Create EM
Delete EM
Change EM
freeRTOS
Idle Task
Power Management Framework (PMF)
Administrator
List of Tasks Sleep mode 0
List of Tasks Sleep mode 1
...
List of Tasks Sleep Mode N
TASK 1
TASK 2
TASK 3
TASK 4

Figure 5.6: Power management Framework with list sleep mode solutions

5.3.2.1 Advantages

There are not particular advantages linked to this solution, but actually it does not use a large amount of memory. Indeed, the data structure uses only pointers to link the tasks belonging for each sleep mode.

5.3.2.2 Shortcomings

There are several drawbacks connected to this solution because it involves an high overhead inside the system. This solution is a bit heavy since it needs to write the whole code to handle the lists of TCBs and it would introduce a further overhead for insertion and deletion of tasks. Additionally, the whole data structure is not as efficient as the first solution and the complexity of the whole algorithm is mainly complicated. This solution has been dropped right away because it implies many changes inside PMF.

5.3.3 Array sleep mode solution

This idea is a continuation of Martin's thesis [3]. Actually, I preferred to continue using the same solution because it is the easiest and it does not involve an high overhead inside the system. The idea has been enhanced introducing a support for management of sleep mode also for ISRs.
I divided the solution in two sub-frameworks called TPMF (*Task Power Management Framework*)(see section 5.3.3.1) and IPMF (*ISR Power Management Framework*)(see section 5.3.3.2). In each framework are explained their modality of functioning and are also proposed some examples. Finally, there is a description of

the real framework implemented in FreeRTOS, called GPMF (*Global Power Management Framework*) (see section 5.3.3.3). GPMF is a combination of TPMF and IPMF.

5.3.3.1 Task's data structure (TPMF)

The data structure is composed by an array of sleep modes. For each position (therefore for each sleep mode) there is a counter that keeps track the number of tasks belonging in that sleep mode. Moreover, the field *ucSleepMode* of TCB (see Listing 5.3 in section 5.1) informs which sleep mode is used for every task.

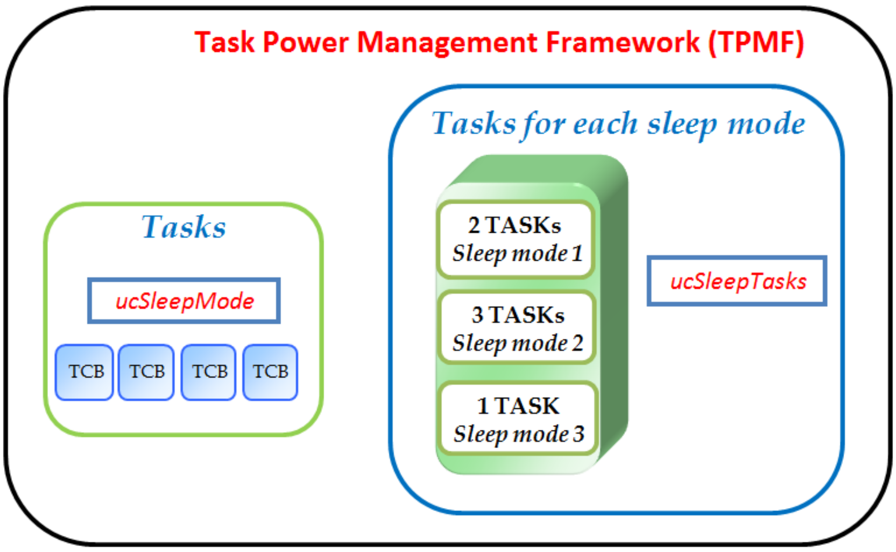


Figure 5.7: Task Power Management Framework (TPMF)

The Figure (5.7), illustrates the data structures used to handle the tasks requests. The *administrator* is the entity in charge to manage the incoming requests issued by programmers and idle tasks. The example shows that two tasks have requested to stay in *sleep mode 1*, while three in *sleep mode 2* and one in *sleep mode 3*.

The functioning of Task Power Management Framework (TPMF) is reported below.

1. When a task is being created, it updates own TCB with a default sleep mode (defined by TPMF) and it also updates the corresponding sleep mode position inside the array;

2. If a task changes own sleep mode during its execution, then TPMF provides to keep updated the corresponding positions inside the array (*ucSleepTasks*) and own TCB;

3. If a task is being deleted, the TPMF removes the reference of its sleep mode inside *ucSleepTasks*;

4. If an idle task occurs, the administrator searches inside the array the lowest sleep mode.

5. If the programmer does not define any sleep modes inside Task's body, a default mode is used to manage idle task. This value is defined by Global Power Management Framework (GPMF) (see section 5.3.3.3);

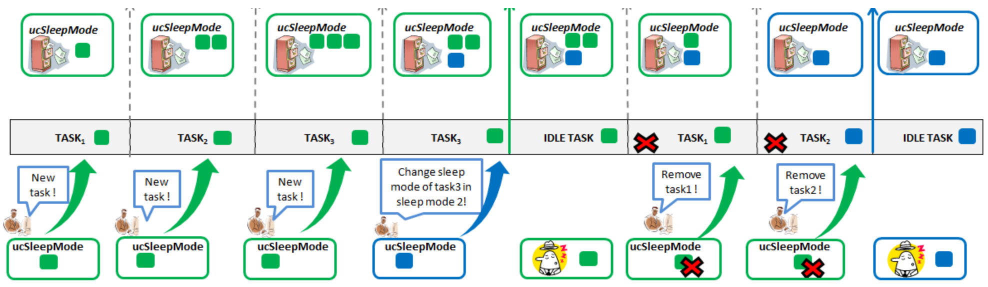


Figure 5.8: Possible use of Task Power Management Framework (TPMF)

Figure (5.8) shows a possible example how to manage sleep modes using TPMF. The operations used in TPMF, are: *create a new task*, *remove a task* and *change the sleep mode of a task*.
The example illustrates that three tasks are created in first stages (3 green squares). TPMF's *administrator* provides to keep updated the number of tasks associated for each sleep mode inside *ucSleepMode*.
At the fourth stage, *programmer* (user) changes the sleep mode associated to $Task_3$ from sleep mode 1 to 2, therefore *ucSleepMode* is immediately updated (1 blue square). An *idle task* (sleepyhead) occurs and the administrator searches inside *ucSleepMode* the most suitable sleep mode. The *green arrow* before idle task means that idle task can sleep in *sleep mode 1* (green). Afterwards, $Task_1$ and $Task_2$ are removed. Thus, *ucSleepMode* has only the sleep mode of $Task_3$ (blue). Thus, next idle task goes in *sleep mode 2* (blue).

5.3.3.2 ISR's data structure (IPMF)

Unlike tasks, ISRs have a different behavior because they can occur everywhere. Besides, the system might have the need to stay in a different sleep mode after their execution.
Thus, there are some factors that do not allow to manage the sleep modes very easily. Some of those factors are reported below.

- ISRs does not have own structure (like TCB for tasks);

- ISRs are asynchronous events and it is not easy to manage all of them;

- The number of ISRs is not constant for all applications and it depends from the number of resources that the developer is intended to use;
- In some cases it is possible to have nested ISRs, therefore there is a need to keep a sleep mode frozen for a while;
- An ISRs can accomplish its job in different ISRs, hence even in this case, there is a need to hold a sleep mode locked until the end of last ISR's execution;
- It is not easy to predict for how long the sleep mode set inside an ISR must be held valid after its execution;

The ISR Power Management Framework (IPMF) outlined in this section is the entity in charge to handle the sleep mode issued by each ISR. For the reasons explained in section (5.2) IPMF must be able to deal with all possible pitfalls. Therefore, this solution can set a sleep mode inside an ISR in two different ways, such as:

- *Simple Sleep Mode(SSM)*: The sleep mode defined inside the ISR is kept valid only until the first idle task is met. Afterwards, the value defined in that ISR is not taken into account for further idle tasks. See Figure (5.10);
- *Lock Sleep Mode(LSM)*: The sleep mode defined inside the ISR is locked for a while. There are two places where it is possible to unlock the frozen sleep modes, such as: *ISR* or *Task*. See Figure (5.11);

Thus, it is up to developer to choose the way to set a sleep mode inside an ISR. If there are nested ISRs or their execution is divided in different times, then it is more useful to lock a sleep mode. Even though the use of this feature seems to be user friendly, it must be used carefully, because the system might work for a while in a not ideal sleep state. Additionally, an application can also lock concurrently many sleep modes. The data structure used to manage the sleep modes for ISRs has a different meaning. The IPMF implements both LSM and SSM. Regarding the *SSM* case, the system saves the ISR's sleep mode (in *usSleepISR*) only until the completation of the first idle task. While in *LSM* case, it is used an array to store the locked sleep modes. Each position of array (*ucSleepLockISR*) corresponds to a sleep mode, thus when a sleep mode is being locked or unlocked the related sleep mode position is updated.

The Figure (5.9) shows the data structure used to manage the ISRs requests. In spite of IPMF, now the array *ucSleepLockISR* is used to keep on track the number of ISRs locked for each sleep mode. Moreover the example shows that two ISRs have requested to lock *sleep mode 1*, while an ISR locked to stay in *sleep mode 2* and another one in *sleep mode 3*. This array is only used when the *LSM* feature is chosen by the programmer, otherwise the ISR makes a normal allocation of sleep mode inside the variable *ucSleepISR*. Finally, the variable *ucFlagISR* is used to manage the end of the first idle task (used to satisfy the requirement of *Simple Sleep Mode (SSM)*). The functioning of IPMF is reported below.

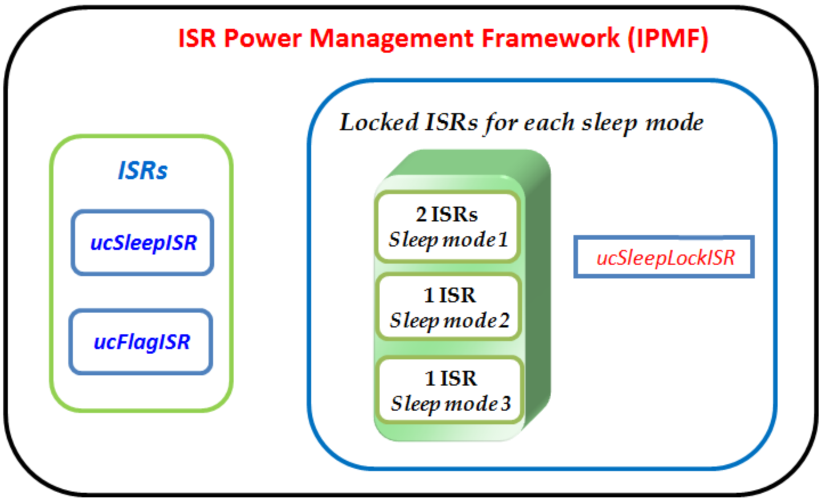


Figure 5.9: ISR Power Management Framework (IPMF)

1. If the developer intends to use a *Lock Sleep Mode* (LSM), the system provides to increment or decrement the corresponding value in the array (whenever locking or unlocking operations occur);
2. If the developer intends to use *Simple Sleep Mode* (SSM), the system stores its value inside *ucSleepISR*;
3. When an idle task occurs, IPMF checks both *ucSleepISR* and the array of locked sleep modes (*ucSleepLockISR*). IPMF returns the most suitable sleep mode among requestes done by all ISRs;
4. If the programmer does not define any sleep modes inside ISRs, a default mode is used to manage idle task. This value is defined by Global Power Management Framework (GPMF) (see section 5.3.3.3);

The three following examples show a possible use of both features described above, SSM and LSM. Respectively, there is an example for each modality (one for SSM and one for LSM) and another one that shows a possible combination when both of them are merged.

Figure (5.10) shows a possible example to manage the sleep modes using *SSM* in IPMF. Initially, the system starts with a *default configuration* (red). This state is used whenever a programmer does not express any requests to move in specific sleep mode (default value is defined in an internal variable of the system).
The programmer sets a SSM in mode 2 inside the $ISR_1's$ body (the request can be issued everywhere inside the body). Then, the *administrator* provides to update the content of *ucSleepISR* variable with chosen sleep mode.
Afterwards, an *idle task* occurs and the administrator searches which is the sleep mode stored in *ucSleepISR* (if any). It found mode 1 (green), but if it does not

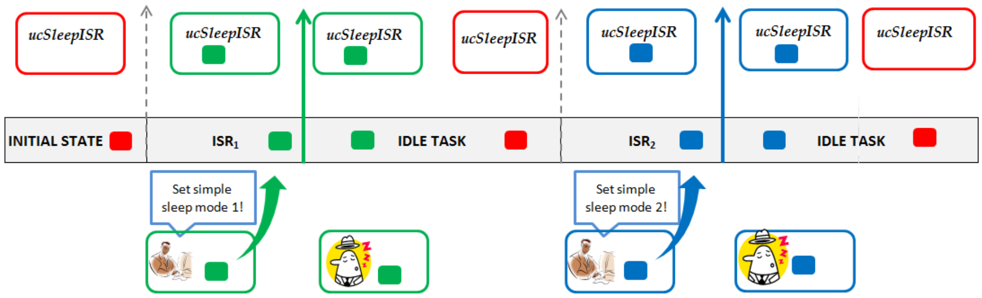


Figure 5.10: Possible use of ISR Power Management Framework (IPMF) with *Simple Sleep Mode(SSM)*

find anything, it returns the default mode. Right away, the administrator pushes *ucSleepISR* in a default state, thus it avoids that the content of that variable is used again for future idle tasks. After idle task, ISR_2 occurs and the programmer communicates sleep mode 2, now IPMF proceeds same actions it did before (but with a different sleep mode).

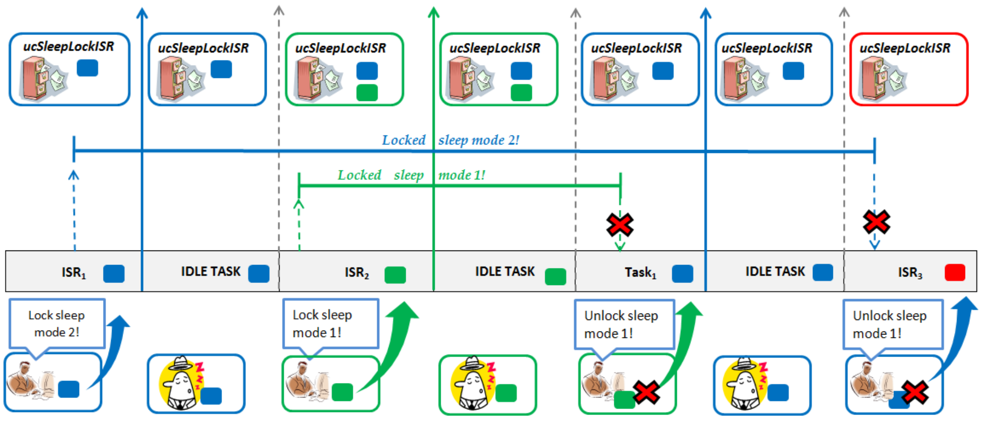


Figure 5.11: Possible use of ISR Power Management Framework (IPMF) with *Lock Sleep Mode(LSM)*

Figure (5.11) shows a possible example to manage the sleep modes using *LSM* in IPMF. At the first step, ISR_1 needs to *lock sleep mode 2*. Then, the *administrator* provides to update the value inside *ucSleepLockISR* with a new request for sleep mode 2. After that, an *idle task* (sleepyhead) occurs and its sleep mode is modality 2, as shown by *blue arrow*. Afterwards, a new lock request comes from ISR_2 for sleep mode 1, this involves to use modality 1 (green) for next idle task. Right away, after idle task in $Task_1$, the programmer unlocks the sleep mode 1. From this point forward, the sleep mode 2 is back valid. In fact, this is used for last idle task before to unlock it in ISR_3. After that the system enters in a default state

(red).
As shown in the example, locking sleep mode for a while, might force the system to use an unintentional sleep mode during idle task. So, this involves to use locking functionality only when the programmer knows exactly the instants when tasks and ISRs occur. Otherwise, the programmer may push the system in a *loop forced sleep mode*, in other words, if an ISR occurs very often, this will always force its sleep mode. Thus, other tasks or ISRs will never acquire the control of idle task.
Some possible cases when it is useful to use *LSM* are: nested ISRs or when the system is waiting an answer from a peripheral.

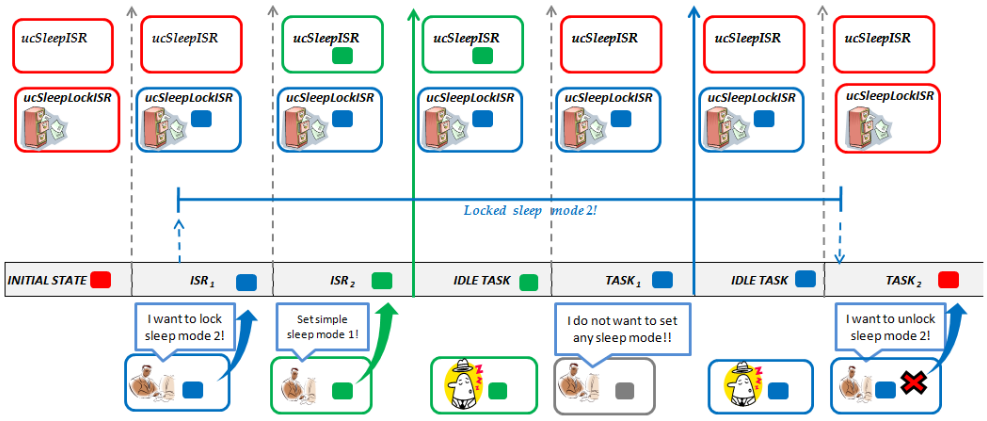


Figure 5.12: Possible use of ISR Power Management Framework (IPMF) with *Lock Sleep Mode(LSM)* and *Simple Sleep Mode(SSM)*

Figure (5.12) shows a possible example to manage the sleep modes using both *LSM* and *SSM* in IPMF. The purpose of this example is to show how IPMF really works in the final framework and how LSM and SSM interact each other.

In the first step, both LSM and SSM are in a default state. After that, ISR_1 locks sleep mode 2 with LSM while ISR_2 sets simple mode 1 with SSM. The *administrator* provides to update their requests inside *ucSleepISR* and *ucSleepLockISR*. Now, when *idle task* occurs (sleepyhead) the administrator checks which is the structure that contains the highest sleep mode. The idle task goes in sleep mode 1 (green) because *ucSleepISR* has higher priority than *ucSleepLockISR*. At the same time, the administrator provides to restore *ucSleepISR* with its default value (this operation inhibits its value for future idle tasks).
From this point forward, only the sleep modes locked in *ucSleepLockISR* are taken into account. In fact, when idle task occurs, it goes in sleep mode 2 (blue) because this mode has been locked previously by ISR_1. Afterwards, $Task_3$ occurs and it issues the necessity to *unlock sleep mode 2*. After that, both data structures work in default state until a new request occurs.
This example shows a simple case how it is possible to use the whole IPMF. Furthermore, if necessary, IPMF handles even the possibility to issue requests by LSM

and SSM inside the same ISR. It is important to remark that IPMF can *only* issue a request inside ISR's body, but those requests can be unlocked *both* inside ISRs and Tasks.

5.3.3.3 Global Power Management Framework (GPMF)

GPMF is the entity in charge to select the most suitable sleep mode in idle task. It embeds both TPMF and IPMF features, thus whenever an idle task needs to know its sleep mode, GPMF searches the value inside its data structures. Respectively, *ucSleepTasks* for TPMF and *ucSleepISR* and *ucSleepLockISR* for IPMF. If none of them define a sleep mode for idle task, then, GPMF according with an its internal value, it defines a default value to use for idle task. How to define the default value is explained in following section (see section 5.4.1).

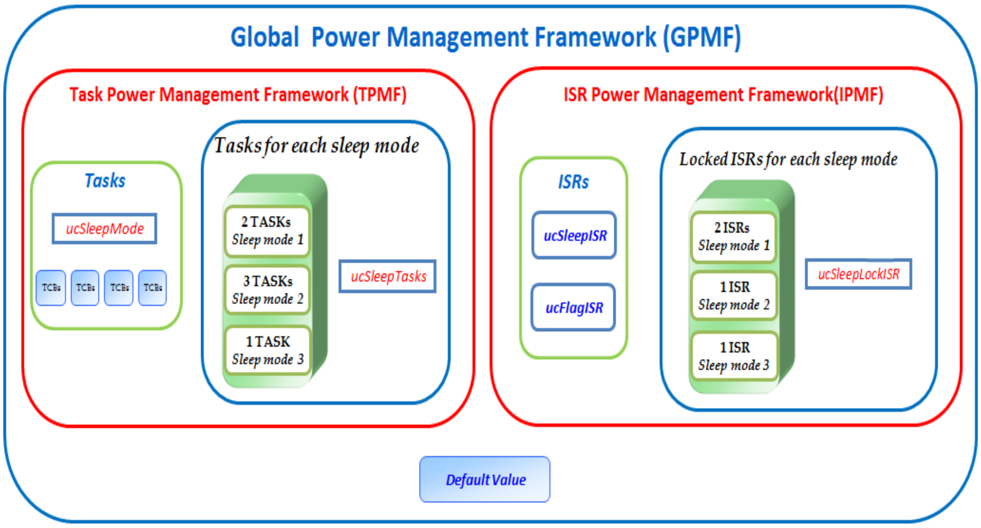


Figure 5.13: Global Power Management Framework (GPMF)

Figure (5.13) illustrates an overview of its internal data structures. It is composed by structures used in IPMF and TPMF. Additionally, it provides an internal value *DEFAULT_VALUE* used to keep stored the default value . The *administrator* is the entity in charge to deal with external incoming requests.

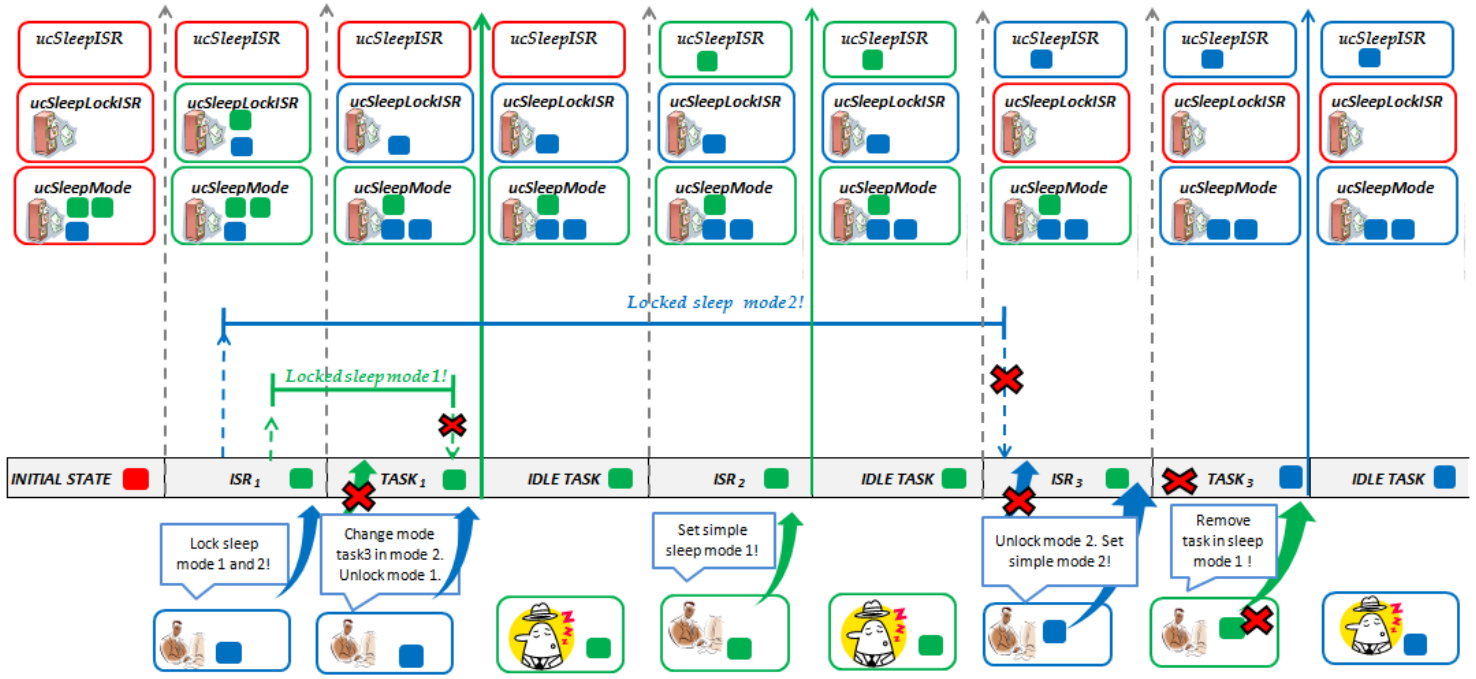


Figure 5.14: Example of Global Power Management Framework (GPMF) functioning

Figure (5.14) shows an example of using GPMF among tasks and ISRs. At the beginning, the data structure of tasks (*ucSleepMode*) is initialized with some values (in order to reduce the steps of the example), while the other structures *ucSleepISR* and *ucSleepLockISR* are in default state. After that, ISR_1 occurs and the programmer locks both sleep mode 1 and sleep mode 2, these changes are pushed inside corresponding data structure, *ucSleepLockISR*.
Upon $Task_3$, the programmer changes its sleep mode and he also unlocks sleep mode 1 (locked in previous ISR). When *idle task*(sleepyhead) occurs, the *administrator* notifies that the ideal sleep mode for idle task is mode 1 (green), as shown on the green arrow. After that, the programmer sets mode 1 using *SSM* inside ISR_2. For the next idle task, the sleep mode 1 is hold by *ucSleepISR* and *ucSleepMode*, then idle task goes in sleep mode 1(green). While, the next ISR unlocks sleep mode 2 and it also fixes a mode 2 in *ucSleepISR*. Right now, *ucSleepLockISR* is set as default state. The next request removes the last task in mode 1 and it allowes to move the incoming idle task in mode 2 (blue).

Moreover, GPMF moves idle task in a lowest sleep mode only if IPMF agrees with TPMF. Upon *idle task*, the administrator searches the content of all data structure (*ucSleepISR*, *ucSleepLockISR* and *ucSleepMode*) and it returns that value in common among them.
In the next sections, I will explain the steps necessary to port GPMF in a new platform (see section 5.5) and its porting for EFM32 (Energy Micro) (see section 5.6).

5.3.3.4 Advantages

This solution allows to build a framework with a very low overhead in terms of code and complexity. Moreover, the data structure used to implement this solution with n tasks, p sleep modes and m locked ISRs, is:

$$Space\_used = (n + p + m + ISR\_sleep\_mode + ucFlagISR) * (8bits)$$

Nevertheless, this formula takes into account only those variable used as data structure. In fact, GPMF does not consider some other variables used for internal management (inside functions) and other constants used internally (such as DEFAULT_VALUE).
Additionally, the functions provided to developers are easy to use and they do not involve a burden for GPMF (see section 5.4.2).

5.3.3.5 Shortcomings

The main drawback of this feature is due to the fact that system freezes a sleep state for a while, thus if it takes a long time, then the system is forced to work for a fixed sleep mode. Furthermore, an excessive use of locking sleep modes is useless.

5.4 GPMF

In order to lead an easy interaction between programmers and GPMF, this section is aimed to present with more details its internal features (see section 5.4.1,5.4.2). GPMF is a *Generic Framework*, therefore, it is not been designed to run on a specific platform. These customers eager to use its functionality, before to getting started with it, they have to *porting* their architecture with GPMF.
In practice, each platform has a different device hardware, hence, GPMF needs to know instructions and features belonging of that platform. It provides a section (PPMF) where it is possible to define compatible functions with the new architecture (see section 5.5).

5.4.1 GPMF module

The whole GPMF module is divided in three different parts, such as *FreeRTOS*, *GPMF* and *programmer interface*. The Figure (5.15) shows how these three entities interact and communicate each other.

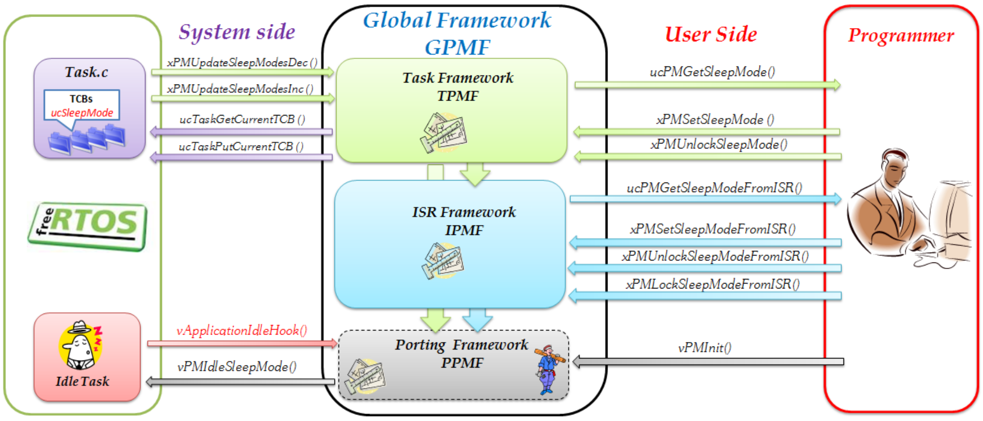


Figure 5.15: GPMF module

GPMF Framework (in the middle) is the main part of the whole module. It provides to have an high level of abstraction between system and programmers, therefore it avoids the possibility to lead the system in a *jeopardy* state. Furthermore, the interaction with GPMF can occur in two different ways, one is the interaction with programmers (*user side*) and another one between its framework and FreeRTOS (*system side*). Now, GPMF is able to cope the needs discussed in section (5.1).
GPMF module is composed of three different sub-modules, such as *Task Framework* (TPMF) , *ISR Framework* and *Porting Framework* (PPMF). The first two frameworks have been widely discussed respectively in section (5.3.3.1) and (5.3.3.2).

Whereas, PPMF is the place where customers[3] can provide a porting of their platforms (the image of *worker* means that PPMF is built for each architecture). This part is not visible to programmers and mostly it does not affect the interactions with TPMF and IPMF (except for `vPMInit()`).

The part regarding *user side* allows programmer to use the functions provided by TPMF and IPMF. The programmer, before using any functions has to invoke `vPMInit()` at the beginning of program (before the scheduler and creation of tasks) . Nevertheless, the use of this function depends of the platform used. The purpose of this function is to allow customers to initialize some internal registers useful for the power management. Hence, this function is compulsory only if the customer declares to use it, otherwise it can be discarded (this is the reason why the arrow is grey). *ISR Framework* (see section 5.3.3.2) provides the possibility to use SSM or LSM inside ISRs. The function used to set a *Simple Sleep Mode* for ISR is `xPMSetSleepModeFromISR()`. This function sets a sleep mode (given as input) only until the first idle task is met. The function used to *Lock Sleep Mode* is `xPMLockSleepModeFromISR()`. On the contrary, the function used to unlock a previous locked sleep mode is `xPMUnlockSleepModeFromISR ()`. Both functions receive as input the sleep mode to lock or unlock. The last function available in IPMF `ucPMGetSleepModeFromISR()` returns the current power modes where the system can go to. *Task framework* (see section 5.3.3.1) provides the possibility to define power modes for tasks and also a small part for ISRs. Function `xPMSetSleepMode()` provides to set a sleep mode (given as input) for a task. Whereas `ucPMGetSleepMode()` is the duality function described in IPMF. `xPMUnlockSleepMode()` is another function available in TPMF and it allows to unlock the sleep mode (given as input) previously locked inside an ISR.

The part regarding *system side* manages the interaction between GPMF and FreeRTOS. In this section have been reported only those modules that interact with GPMF, such as *Task.c* and *Idle Task*. The task module contains the TCB (see section 3.2.1.1) allocated for each task. Then, when a task being removed, GPMF has to know in which sleep mode that task was for. The function used to accomplish this request is `xPMUpdateSleepModesDec()`. It removes the associated sleep mode inside *ucSleepTasks*. On the contrary, when a task has been created, the framework has to know in which sleep mode it wants to go. The function used in this case is called `xPMUpdateSleepModesInc()` and it also updates *ucSleepTasks*.
During task execution GPMF might receive a request to associate a sleep mode for the task running. Hence, it updates *ucSleepTasks* and it also need to inform the TCB that its sleep mode (*ucSleepMode*) might be changed. Since GPMF does not have the authorization to change the value of TCB, two functions have been defined inside task module[4]. These functions are called `ucTaskPutCurrentTCB()` and

[3]Customers are those companies that provide Microcontrollers, such as Energy Micro, Texas Instruments, STMicroelectronicas, etc...

[4]Task module is composed by task.c and task.h. Here, the system defines all the functions used to interact with TCB

`ucTaskGetCurrentTCB()`. The first function looks for storing the power mode (given as input) inside *ucSleepMode*, while the second provides to return the value contained in *ucSleepMode*.
Whereas, when an idle task occurs, the module of idle task establishes a communication with GPMF. Actually, FreeRTOS uses `vApplicationIdleHook()` to inform GPMF when it needs to go in idle (the color of arrow is red because this function is provided by FreeRTOS). GPMF communicates the sleep mode to use for idle task using `vPMIdleSleepMode()`. This function is widely discussed in PPMF (see section 5.5) because it is one of the functions that need a porting.

5.4.2 GPMF Reference Manual

This section gives a detailed explanation for each function provided by the whole framework.

5.4.2.1 xPMSetSleepMode()

```
#include "PowerManager.h"

 portBASE_TYPE xPMSetSleepMode ( unsigned portCHAR xMode)
  {
  portBASE_TYPE xReturn;

  portENTER_CRITICAL();
  /*Check if the value given in input is valid*/
  if ( xPMCheckMode( xMode ) == pdTRUE){
     ucSleepTasks[ucTaskGetCurrentTCB()]--;
           vTaskPutCurrentTCB(xMode);
           ucSleepTasks[xMode]++;
     xReturn = pdTRUE;
  }else{
           xReturn = errINSERT_NOT_VALID_SLEEPMODE;
  }
  portEXIT_CRITICAL();

  return xReturn;
  }
```

Listing 5.2: Prototype ad implementation of xPMSetSleepMode()

Description
This function provides to associate a sleep mode for a Task. This function can be invoked everywhere inside the task's body.

Parameters
`xMode`: The sleep mode must be a value within the range allowed by the system.

Return values
`pdTRUE`: Sleep mode has been set successfully inside data structure.

`errINSERT_NOT_VALID_SLEEPMODE`: Sleep mode passed as input is not a valid value. The value must be within the range of configDEFAULT_SLEEPMODE and configSLEEPMODES.

Note
This function uses `ucTaskPutCurrentTCB()` and `ucTaskGetCurrentTCB()` to interact with TCB stored inside task module.

5.4.2.2 xPMSetSleepModeFromISR()

```c
#include "PowerManager.h"

  portBASE_TYPE xPMSetSleepModeFromISR ( unsigned portCHAR xMode )
  {
  portBASE_TYPE xReturn;

  /*Check if the value given in input is valid*/
  if ( xPMCheckMode( xMode ) == pdTRUE){
           ucSleepISR = xMode;
     ucFlagISR = 1;
     xReturn = pdTRUE;
  }else
           xReturn = errINSERT_NOT_VALID_SLEEPMODE;

  return xReturn;
  }
```

Listing 5.3: Prototype and implementation of xPMSetSleepModeFromISR()

Description
This function sets a sleep mode for the ISR where is being called. The approach used to set the sleep mode is SSM.

Parameters
`xMode`: The sleep mode must be a value within the range allowed by the system.

Return values
`pdTRUE`: Sleep mode has been set successfully inside data structure.

`errINSERT_NOT_VALID_SLEEPMODE`: Sleep mode passed as input is not a valid value. The value must be within the range of configDEFAULT_SLEEPMODE and configSLEEPMODES.

Note
The sleep mode is kept valid until the first idle task is executed.

5.4.2.3 ucPMGetSleepMode()/ ucPMGetSleepModeFromISR()

```c
#include "PowerManager.h"

unsigned portCHAR ucPMGetSleepMode ( void )
  {
    unsigned portCHAR i, ucReturn = configDEFAULT_SLEEPMODE;

    portENTER_CRITICAL();

   /*Check which is the suitable mode between tasks and LockISRs*/
    for (i=0; i<configSLEEPMODES; i++)
  if ( (ucSleepTasks[i] > 0) || (ucSleepLockISR[i] > 0)){
     ucReturn = i;
     break;
        }

   /*Check which is the suitable mode among simple ISR, tasks and LockISR */
  if ( ( ucFlagISR) && ( ucReturn > ucSleepISR))
     ucReturn = ucSleepISR;

    portEXIT_CRITICAL();

    return ucReturn;
  }
```

Listing 5.4: Prototype and implementation of ucPMGetSleepMode() and ucPMGetSleepModeFromISR()

Description
This function retrieves the current sleep mode among the data structures used to save the requests done by tasks, SSM and LSM.

Return values
`xMode`: Lowest sleep mode found among Tasks, SSM and LSM.

Note
Initially, the function checks the sleep mode inside the array of tasks and Locked ISRs. Then, if SSM has a request of a sleep mode higher than the value found in previous structures it becomes the new power mode.
`ucPMGetSleepModeFromISR()` has the same body but the instructions are performed without using critical region. The functions used to delimit a critical regions are `portENTER_CRITICAL()` and `portEXIT_CRITICAL()`.

5.4.2.4 xPMLockSleepModeFromISR

```
#include "PowerManager.h"

  portBASE_TYPE xPMLockSleepModeFromISR ( unsigned portCHAR xMode)
  {
    /*Check if the value given in input is a valid one*/
    if ( xPMCheckMode( xMode ) == pdTRUE){
       ucSleepLockISR[xMode]++;
    return pdTRUE;
       }else
    return errINSERT_NOT_VALID_SLEEPMODE;
  }
```

Listing 5.5: Prototype and implementation of xPMLockSleepModeFromISR()

Description
This function locks a sleep mode for the ISR where is being called.

Parameters
`xMode`: The sleep mode must be a value within the range allowed by the system.

Return values
`pdTRUE`: Sleep mode has been locked successfully inside data structure.

`errINSERT_NOT_VALID_SLEEPMODE`: Sleep mode passed as input is not a valid value. The value must be within the range of configDEFAULT_SLEEPMODE and configSLEEPMODES.

Note
This function can be invoked only inside an ISR. Besides, this function can be used as an alterantive solution to `xPMSetSleepModeFromISR()` whenever the application needs to keep a sleep mode locked for a while (including more idle tasks). Since the use of this function can involve in a misleading use of sleep modes, it must be used only when it is necessary. The function can lock more sleep modes and it also handles nested ISRs.

5.4.2.5 xPMUnlockSleepModeFromISR/ xPMUnlockSleepMode

```
#include "PowerManager.h"

portBASE_TYPE xPMUnlockSleepModeFromISR (unsigned portCHAR xMode)
  {
    /*Check if the value given in input is a valid one*/
     if ( xPMCheckMode( xMode ) == pdTRUE){
    /*If a request is issued and its value is equal zero, it means that there is no↩
          ISR to unlock*/
     if (ucSleepLockISR[xMode] == 0)
      return errCOULD_NOT_UNLOCK_SLEEPMODE;
     else{
      ucSleepLockISR[xMode]--;
      return pdTRUE;
     }
     }else
    return errINSERT_NOT_VALID_SLEEPMODE;
  }


portBASE_TYPE xPMUnlockSleepMode (unsigned portCHAR xMode)
  {
    portBASE_TYPE xReturn;

       portENTER_CRITICAL();

       /*Check if the value given in input is a valid one*/
       if ( xPMCheckMode( xMode ) == pdTRUE){
          /*If a request is issued and its value is equal zero, it means that there↩
                is no ISR to unlock*/
      if (ucSleepLockISR[xMode] == 0)
        xReturn = errCOULD_NOT_UNLOCK_SLEEPMODE;
      else{
        ucSleepLockISR[xMode]--;
        xReturn= pdTRUE;
      }
       }else
      xReturn = errINSERT_NOT_VALID_SLEEPMODE;

       portEXIT_CRITICAL();

    return xReturn;
  }
```

Listing 5.6: Prototype and implementation of xPMUnlockSleepModeFromISR() and xPMUnlockSleepMode()

Description
This function unlocks a sleep mode locked with `xPMLockSleepModeFromISR()`. Respectively, `xPMUnlockSleepModeFromISR()` unlocks a sleep mode inside an ISR, whereas `xPMUnlockSleepMode()` inside a Task.

Parameters
`xMode`: The sleep mode must be a value within the range allowed by the system.

Return values
`pdTRUE`: Sleep mode has been set successfully inside data structure.

`errINSERT_NOT_VALID_SLEEPMODE`: Sleep mode passed as input is not a valid value. The value must be within the range of configDEFAULT_SLEEPMODE and configSLEEPMODES.

`errCOULD_NOT_UNLOCK_SLEEPMODE`: Attempt to unlock a wrong sleep mode.

Note
If the function tries to unlock a sleep mode not locked previously, the function discards the request and returns `errCOULD_NOT_UNLOCK_SLEEPMODE`. `xPMUnlockSleepModeFromISR()` has the same body but the instructions are performed without using critical region. The functions used to delimit a critical regions are `portENTER_CRITICAL()` and `portEXIT_CRITICAL()`.

5.4.2.6 vPMIdleSleepMode

```
#include "PowerManager.h"

 portBASE_TYPE ucPMIdleSleepMode ( void )
  {
    unsigned portCHAR sleep;
    portBASE_TYPE xReturn;

      vTaskSuspendAll();
      portENTER_CRITICAL();

      /* Retrieve the sleep mode to use for idle task */
      sleep = ucPMGetSleepMode();
      xPMGoToSleepMode(sleep); /* Go in sleep mode */
      /* Reset ucFlagISR if SSM issues a request before idle task */
      ucFlagISR = 0;

      portEXIT_CRITICAL();
      xTaskResumeAll();

    return xReturn;
  }
```

Listing 5.7: Prototype and implementation of vPMIdleSleepMode()

Description
This function is called by the system when the idle task occurs. It provides to move the MCU in the right sleep mode.

Return values
`xMode`: Lowest sleep mode found among Tasks, SSM and LSM.

`errCOULD_NOT_GOTO_SLEEPMODE`: Attempt to go in a wrong sleep mode

Note
If the customer declares to use `vPMInit()` and the programmer does not use it, then the function could not work properly. This function does not manage the tickless feature. In next chapters is proposed a new `vPMIdleSleepMode` that removes the tick interrupts that occur during idle task.

5.4.2.7 xPMUpdateSleepModesInc/ xPMUpdateSleepModesDec

```
#include "PowerManager.h"

 /* This function is used inside prvInitialiseTCBVariables() to keep updated ↩
     ucSleepTasks
   after  creation of a task*/
void vPMUpdateSleepModesInc (unsigned portCHAR xMode )
{
   ucSleepTasks[xMode]++;
}

/* This function is used inside vTaskDelete( xTaskHandle pxTaskToDelete ) to keep ↩
     updated ucSleepTasks
   after  deletion of a task*/
void xPMUpdateSleepModesDec ( unsigned portCHAR xMode )
{
   ucSleepTasks[mode]--;
}
```

Listing 5.8: Prototype and implementation of xPMUpdateSleepModesInc() and xPMUpdateSleepModesDec()

Description
`xPMUpdateSleepModesInc()` and `xPMUpdateSleepModesDec` are used to update the task data structure from task module. Both are system functions and they can not be used by programmers.

Parameters
`xMode`: The sleep mode must be a value within the range allowed by the system.

Note
`xPMUpdateSleepModesInc()` is used when the system creates a new task.
`xPMUpdateSleepModesDec()` is used when the system deletes a task.

5.4.2.8 ucTaskGetCurrentTCB/ucTaskPutCurrentTCB

```
#include "PowerManager.h"

#if ( configUSE_POWERMANAGER == 1 )
PRIVILEGED unsigned portCHAR ucTaskGetCurrentTCB( void ){
    return pxCurrentTCB->ucSleepMode;
}

PRIVILEGED void ucTaskPutCurrentTCB( unsigned portCHAR mode ){
    pxCurrentTCB->ucSleepMode = mode;
}

#endif
```

Listing 5.9: Prototype and implementation of ucTaskGetCurrentTCB() and ucTaskPutCurrentTCB()

Description
`ucTaskGetCurrentTCB()` and `ucTaskPutCurrentTCB` are used to interact with the sleep mode field (*ucSleepMode*) stored inside the TCB. Both are system functions and they can not be used by programmers.

Parameters
`xMode`: The sleep mode must be a value within the range allowed by the system. Used only in `ucTaskPutCurrentTCB()`.

Return values
`xMode`: Sleep mode associated to a TCB. Used only in `ucTaskGetCurrentTCB()`.

Note
Both of them refer to the current TCB used by the system. This means, that the operation are proceeded only with the running task. Declaration and definitions of these functions is situated inside task module.

5.5 Porting Power Management Framework (PPMF)

The previous sections describes the functioning of the whole module, GPMF. As explained above, GPMF is a *Generic Framework* with the possibility to *porting* it in any platform.
The porting space includes some functions and constants that must be modified according to the *architecture* intended to use with GPMF. Therefore, these functions must be modified directly by customers and they have to follow some constraints, so, in this way, the whole framework works properly. In section (5.5.1) there is a description about the functioning of PPMF module (functions and data structures

to fill). Whereas, in section (5.5.2) there is a small manual reference about the whole framework.

5.5.1 PPMF module

According to a research carried out for those customers that provide a porting for FreeRTOS, I noticed that most of their MCUs handle the power modes using different sleep modes. Some companies offer MCUs equipped by seven power modes and others with two or three power modes. The porting for a new platform is divided into two parts called *constants porting* and *functions porting*. Figure (5.16) shows the module employed to handle *porting* operations.

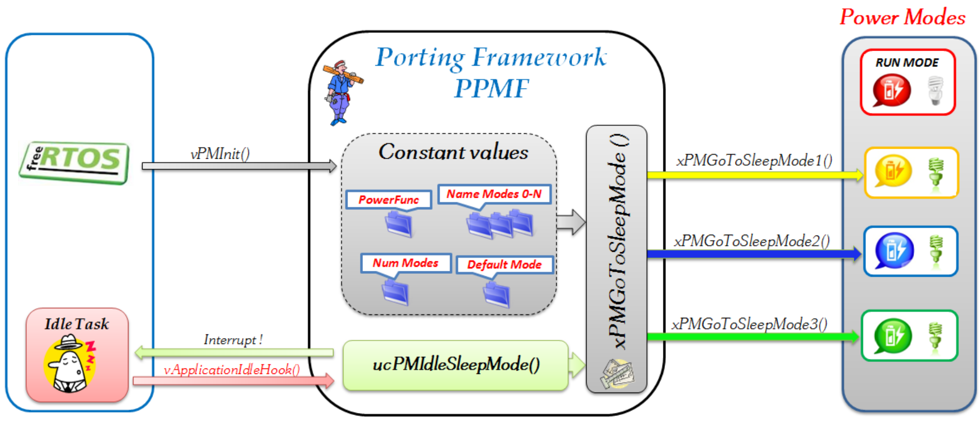


Figure 5.16: PPMF module

In *constants porting*, PPMF includes some internal definitions, such as: *Power* (configUSE_POWERMANAGER), *Default Mode* (configDEFAULT_SLEEPMODE), *Num Modes* (configSLEEPMODES) and *Name modes*. The customer has to adapt the value of these constants to the new architecture.
The constant configUSE_POWERMANAGER is used to enable GPMF in FreeRTOS. If its value is 0, the system does not use GPMF's features otherwise it does. Besides, configSLEEPMODES contains the number of sleep modes (it also includes running mode) provided by MCU. While the constants *Name modes* are used to give a name for each power mode (running mode is also included). The value to assign of each power mode goes from 0 (running mode) to N (deepest sleep mode). It is mandatory to assign 0 for RUNNING MODE. As I discussed in section (5.3.3.3), GPMF uses a default sleep mode when the user does not make used of its features. Thus, customer has to define inside *configDEFAULT_SLEEPMODE* the sleep mode that he wants to use as default state. Hence, according to the value used in *Name modes*, if configDEFAULT_SLEEPMODE is equal to 0, FreeRTOS uses as default state the RUNNING mode. If the customer intends to use the sleep mode 1, he has to put in configDEFAULT_SLEEPMODE the reference number assigned to mode 1.

The instructions needed to push a MCU in a sleep mode are different among customers. For this reason, *function porting* consists to write the instructions needed to move the MCU in each sleep mode. These functions have the following prototype `xPMGoToSleepMode_NUM_()`, where the last word `NUM` refers the sleep mode associated to it. Actually, this is a conventional standard used to allow compatibility with the whole framework, but the customer can choose another name. Hence, the system does not undergo any consequences. When PPMF receives a call (`vApplicationIdleHook()`) by idle task, PPMF uses a system function `vPMIdleSleepMode()` (see section 5.4.2.6) to find the ideal sleep mode to go in. Now it knows the sleep mode and it uses `xPMGoToSleepMode()` to interface with power modes. `xPMGoToSleepMode()` receives the sleep mode (given as input) and it invokes the associated function (provided by customer). When the MCU receives an interrupt or an event, `vPMIdleSleepMode()` provides to wake it up. The function `xPMGoToSleepMode()` needs to know the name of those functions provided by customer. Thus, it requires some small changes. It is forbidden to change the name of this function because it is being invoked in some other *system functions*, such as `vPMIdleSleepMode()`. The last function provided in PPMF is called `vPMInit()` and it is used only in some cases. Indeed, the customer inside the body of this function can declare some particular operations before FreeRTOS starts to run. If the customer does not want to use it, the function need to have an empty body.

5.5.2 PPMF Reference Manual

This section gives a detailed explanation how to set the values for both parts described above, such as *constant porting* and *function porting*. The implementation of the whole framework is divided into two files, such as *PowerManager.h* and *PowerManager.c* (both inside the *source* folder of FreeRTOS).

5.5.2.1 PPMF configuration

The constants are defined in the following files *FreeRTOSConfig.h* and *PowerManger.h*. The latter provides to set those constants used only in GPMF.

```
#ifndef PM_H
#include "FreeRTOS.h"
#include "projdefs.h"
#include "task.h"

#if ( configUSE_POWERMANAGER == 1 )

  /*Number of possible sleep modes*/
  #define configSLEEPMODES        N

  /*PAY ATTENTION!!!
  The order of sleep modes must be ascending and the running mode must always be 0↩
      */
  #define RUNNING        0
  #define MODE1      1
  #define MODE2      2
  #define MODE3     3
    /*Insert a voice for each sleep mode */

  /*Default sleep mode used when the programmer does not use GPMF's features. If ↩
       the value is 0, GPMF uses as default mode the RUNNING state.
  configDEFAULT_SLEEPMODE contains the reference of that sleep mode to use as ↩
       default. If I want to use mode 1 as default, then I copy the
  value of MODE 1.
  */
  #define configDEFAULT_SLEEPMODE   1  /* Mode 1 as Default value */

#endif
#endif
```

Listing 5.10: Constants definition in PowerManager.h

While, the declaration inside FreeRTOSConfig.h is used to enable GPMF. Make sure that configUSE_POWERMANAGER contains 1.

```
#ifndef FREERTOS_CONFIG_H

/*Insert next define within FREERTOS_CONFIG_H delimitators in FreeRTOSConfig.h */
#define configUSE_POWERMANAGER      1

#endif /* FREERTOS_CONFIG_H */
```

Listing 5.11: Constants definition in FreeRTOSConfig.h

5.5.2.2 xPMGoToSleepMode_NUM_

Before writing the body of this function, remember to change also the prototype in *PowerManager.h*.

```
#include "PowerManager.h"

#if ( configUSE_POWERMANAGER == 1 )

    /*Generic function used to move the MCU in power modes 1 */
  void vPMGoToSleepMode1( void )
  {
     /* Include all instructions needed to moce the MCU in power mode 1*/

  }

    /*Generic function used to move the MCU in power modes N. Replace N with ↩
         referred sleep mode*/
    void vPMGoToSleepMode_N_( void )
  {

     /* Include all instructions needed to moce the MCU in power mode N*/

  }
#endif
```

Listing 5.12: Prototype of xPMGoToSleepMode_NUM_ defined in PowerManager.c

Description
The function includes those instructions used to move the MCU in mode N. After

this function the MCU switches in FreeRTOS. Customer has to provide the instructions (if any) also needed to wake up MCU. After its execution, PPMF does not check if the MCU has awakened succesfully.

Note
The customer can also change the name of this function.

5.5.2.3 xPMGoToSleepMode

This function is used in `vPMIdleSleepMode()` when an idle task occurs. The purpose of this function is to move the MCU in the sleep mode received by GPFM.

```
#include "PowerManager.h"

#if ( configUSE_POWERMANAGER == 1 )

 portBASE_TYPE xPMGoToSleepMode ( unsigned portCHAR xMode )
 {
    switch(mode)
  {
          case RUNNING:
        /*Insert code (if necessary)*/
            break;
          case MODE1:
            //vPMGoToSleepMode1(); /*Possible use*/
            break;
          case MODE2:
            //vPMGoToSleepMode2(); /*Possible use*/
            break;
          case MODE3:
            //vPMGoToSleepMode3(); /*Possible use*/
            break;
          case MODEN:
            //vPMGoToSleepModeN(); /*Possible use*/
            break;
          default:
            return errCOULD_NOT_GOTO_SLEEPMODE;
  }
 return pdTRUE;
 }

#endif
```

Listing 5.13: Prototype of xPMGoToSleepMode() defined in PowerManager.c

Description
Function used as a bridge between the idle task and those functions provided by customer.

Parameters
`xMode`: The sleep mode must be a value within the range allowed by the system).

Return values
`pdTRUE`: MCU has been moved in sleep state successfully.

`errCOULD_NOT_GOTO_SLEEPMODE`: Attempt to go in a wrong sleep mode.

Note
The customer has to adapt the switch case if there are more or less sleep modes. Besides, he has to invoke each `xPMGoToSleepMode_NUM_()` inside new switch case. It is forbidden to change the name of this function because it is used in other *system functions*.

5.5.2.4 vPMInit

```c
#ifndef FREERTOS_CONFIG_H

  /* Init function used to initialize registers*/
  void vPMInit ( void )
  {
     /* Instructions needed to initialize some useful registers or something else*/
  }

#endif
```

Listing 5.14: Prototype of vPMInit() defined in PowerManager.c

Description
This function must be called compulsory at beginning of the program (before the scheduler and tasks creation). If the customer does not want to use it, the programmer can discard its use.

Note
Inside the body of this function the customer can define some register's initialization or something else useful for GPMF.

5.6 Porting for EFM32 (Energy Micro)

One of the goals of my thesis requires to make a porting of the whole GPFM for EFM32. In order to *porting* EFM32 for GPMF, I have configured variables values and functions present in PPMF for Energy Micro. Actually, Energy Micro gave me the possibility to use their library EFM32lib[5] to manage the energy modes.
EFM32 library has already provided some functions to manage sleep modes and I used most of the content of these functions. Moreover, I improved some functions in order to make a more efficient and robust porting with my GPFM Framework. Technical information about the use of energy modes and their internal management are dealt in section (2.3.1, 2.3.2). Besides, a more detailed explanation of EMU (Energy Management Unit) of EFM32 is outlined in [5].

[5]EFM32lib contains the Energy Micro Peripheral Support utilities for the EFM32G series of microcontrollers.

5.6.1 PPMF Reference Manual for EFM32

This section illustrates for each function which are my changes and how they impact in the final functioning. The final code for the whole framework realized for EFM32 is shown in Appendix A. Next sections highlight only the considerable chunks of code.

5.6.1.1 PPMF configuration values

```
#ifndef PM_H
#include "FreeRTOS.h"
#include "projdefs.h"
#include "task.h"

#if ( configUSE_POWERMANAGER == 1 )

  /*Number of energy modes. It is also included the running mode*/
  #define configSLEEPMODES        5


  The order of sleep modes must be ascending and the running mode must always be ↩
       0*/
  #define RUNNING   0
  #define EM1     1
  #define EM2     2
  #define EM3     3
    #define EM4     4

    /* If the programmer does not use sleep mode, then GPFM at least use mode 1 ↩
         when idle task occus */
  #define configDEFAULT_SLEEPMODE   1

#endif
#endif
```

Listing 5.15: PPMF configuration values for EFM32

Description
EFM32 provides 4 energy modes plus the runnig state. The default value used in idle task is EM1, thus even if the programmer does not define any EMs (in Tasks and ISRs), the system can save some energy using EM1.

5.6.1.2 vPMInit

```
#ifndef FREERTOS_CONFIG_H

  /* Init function used to initialize registers*/
  void vPMInit ( void )
  {
      /*Lock EMU in init function*/
      EMU->LOCK = EMU_LOCK_LOCKKEY_LOCK;
  }

#endif
```

Listing 5.16: Prototype and implementation of vPMInit()

Description
The EMU of EFM32 gives the possibility to lock (EMU_LOCK) the use of EM during program execution. Then, if the programmer tries to change an energy mode, the MCU discards his request and keep running without performing that operation [5]. Since EFM32 library does not use this feature, I thought to lock the Energy Modes from the beginning until the end. In this way, the framework has the full control of EMU and the programmer does not alter the behaviour of the system. So, the only way to declare a sleep mode is to use GPMF.

5.6.1.3 vPMGoToSleepModeEM1

```
#include "PowerManager.h"

#if ( configUSE_POWERMANAGER == 1 )

    /*Generic function used to move the MCU in power modes 1 */
  void vPMGoToSleepModeEM1( void )
  {
    /*Unlock EMU registers*/
    EMU->LOCK = EMU_LOCK_LOCKKEY_UNLOCK;

    /*It is used to gurantee that the user can not change EMs in the rest of ↩
         program*/
    EMU->CTRL |= EMU_CTRL_EM2BLOCK;

    /*If EMU register was locked, then lock it again */
    EMU->LOCK = EMU_LOCK_LOCKKEY_LOCK;

    /* Enter Cortex-M3 sleep mode */
    SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk;
    __WFE(); /*If I want to wake up with WFI. Then, use __WFI()*/

  }

#endif
```

Listing 5.17: Prototype and implementation of vPMGoToSleepModeEM1

Description
Additionally, the EMU provides also another feature (EMU_CTRL_EM2BLOCK) to lock an energy mode. This possibility is valid only when the system is running in mode 1. If the programmer tries to set EM2 or lower, the system discards again this request. Actually before to set this bit I need to unlock the register previously set in vPMInit(). My point of view is that repetitive locking and unlocking might introduce an higher energy consumption. Thus, I left it only to show the possibility to use this feature. Besides, the system is awakened when any event occurs. If I want to wake it up only when an interrupt occurs, I have to use __WFI(). The difference among WFI and WFE are discussed in section (2.3.2).

5.6.1.4 vPMGoToSleepModeEM2/vPMGoToSleepModeEM3/ vPMGoToSleepModeEM4

These functions are the same used in EFM32 libray except for those operations done over the two bits introduced above (EMU_CTRL_EM2BLOCK) and EMU_LOCK.

When the MCU is awakened, the HFRCO core is automatically restored. Thus, xPMRestoreSleepMode() restores (except HFRCO) of those oscillators (if any) enabled before to move the MCU in sleep mode.

5.7 Goals achieved

The contribution of my work compared with initial motivation presented in section (1.1), tries both to cope some issues presented embedded system field and also proposes a green approach to solve it. Indeed, this framework allows to save energy in accordance with green computing perspective which proposes new possible ways to accomplish same job but wasting less energy. The framework copies issues (see section 1.1.3, 5.1) regarding power management in today's open source RTOSes. Moreover, this framework represents a valid initial alternative to handle sleep modes in any open source platform with a low overhead.
This chapter is a fulfillment of the goal number one proposed in section (1.3). Developing an energy framework and make its porting for EFM32 were the requirements of this goal.

Chapter 6

Keeping time in FreeRTOS with a RTC

"You may delay, but time will not."

- **Benjamin Franklin**,was an American political theorist, politician, postmaster, scientist, musician, inventor, satirist, civic activist, statesman.

This chapter aims to propose and discuss the main benefits of using a Real Time Counter (RTC) as tick timer for a RTOS. Furthermore, it also deals with a possible drawback encountered when the oscillator of RTC works in extreme temperatures environment.
The chapter is organized as follow, sections (6.1, 6.2) illustrate advantages and some useful considerations regarding the use of RTC as tick timer in FreeRTOS. Section (6.3) describes the changes applied in FreeRTOS in order to port the new solution for EFM32. Finally, section (6.4) discusses the temperature problems that affect the behaviour of RTC oscillator. This section also provides an evaluation of possibile solutions currently used to cope temperature issue.

6.1 Advantages of using a RTC as Tick Timer

Basically, most of RTOSes use a module provided by the internal core to handle the timekeeping of the system. As described in 2.4.3, CORTEX-M3 manages the timing using the SysTick module.
Furthermore, FreeRTOS uses an internal variable (see section 3.2.3) to keep updated the timing of the system. Upon a systick interrupt module, the kernel permits to update the value of this variable. Since FreeRTOS schedules tasks according to timer value, it is necessary to update the variable also when the idle task occurs (see section 3.2.1.3). Most likely, in idle state the system runs in one of the two sleep modes, such as *sleep mode* and *deep sleep mode* (see section 2.3.1).
Unfortunately, FreeRTOS can use only one (sleep mode) of these two sleep modes,

because when it goes in the lowest sleep mode (deep sleep mode) the systick module is turned off. In addition, during idle state, the system periodically awakes the core to handle the systick interrupt and to update the xTickCount variable. This causes to waste a considerable amount of energy.

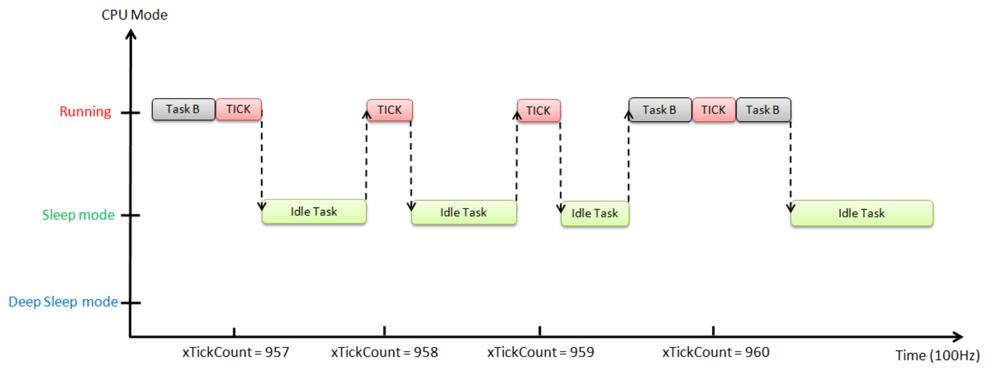


Figure 6.1: Example using FreeRTOS with a Systick module

Figure (6.1) illustrates a classic example how FreeRTOS behaves when it makes use of a systick module. Every time a tick occurs, it handles the interrupt of systick and updates the value of xTickCount. After updating, it returns to sleep or schedule a new task. For the reason explained above, the core never goes in *deep sleep mode*.
As we can see on the example, the major drawback of using a systick timer is due to the lack of using the deepest sleep mode and the possibility to stay in idle state without being interrupted. The first issue can be solved replacing the systick module with a RTC timer. While a possible solution that copes the second problem is widely discussed in next chapter.

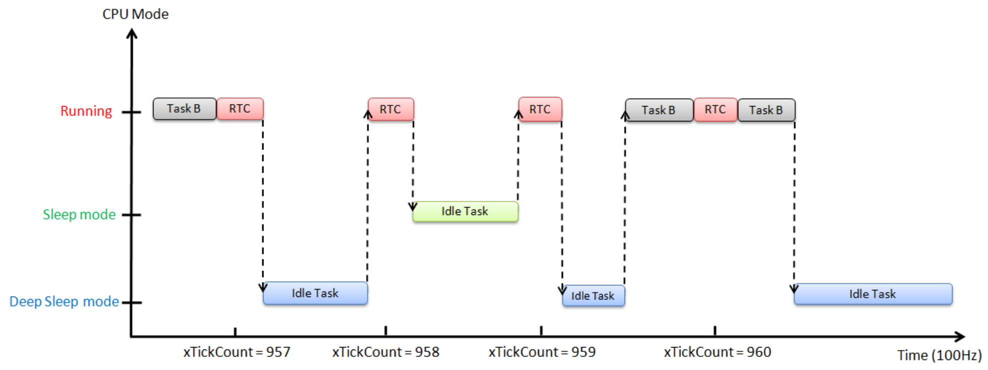


Figure 6.2: Example using FreeRTOS with a RTC as tick timer

The idea to use a RTC as main timer for FreeRTOS enables to put the MCU also in deep sleep mode. Differently, the new solution must handle the RTC Handler instead of the systick interrupt. Figure (6.2) shows how FreeRTOS runs using the RTC. It can choose to stay in one of the two sleep modes according to the needs of the microcontroller.

6.2 Tips to configure a RTC in FreeRTOS

As is described above, FreeRTOS has been built to support the Systick Timer provided by CORTEX-M3. This involves to perform some changes inside the kernel in order to enable that it works also with RTC. However, the new FreeRTOS uses the same variables and constants defined for Systick module. Some further considerations must be take into account when we want to use this new solution.
Indeed, one of the main concern of using a RTC in FreeRTOS is the value of its internal crystal oscillator. Most of RTCs are designed around a standard frequency, such as 32.768 Khz. FreeRTOS defines a constant (`configTICK_RATE_HZ`) where the user can declare in Hz the desired resolution to use in the system. The RTC frequency must be a multiple of this resolution. The division between RTC frequency and kernel resolution represents the number of clock cycles necessary to accomplish the tick rate (time to wait before the RTC issues an interrupt). If the result of this division is not an integer number, then it is being truncated to the upper or lower value. Therefore, this causes to use a wrong tick timer because it will be triggered with a tick before or after. The system would schedule a task with some delay (or before). This drifting value is correlated to the resolution used by the system and it increases with a higher tick rate (10ms or 100ms).
For example, using a resolution of 100Hz the result of the division between the crystal oscillator (32768 Hz) and the resolution of the system is 327,68. This means that a tick occurs every 327,68 clock cycles. Thus, the system will round off this number to 327 or 328. If the system truncates the number to 327, the tick would occur ahead almost a clock cycle before (0,68). This error is cumulative and it affects in a negative way over the time scale.
Therefore, some valid frequencies to use with RTC can be 128Hz with a time resolution of 7.8ms and 1024Hz with a time resolution of 977 µs. This could be a burden in some cases, but it is extremely important to define carefully the value of tick rate. In the following sections are proposed some possible parameters tailored for some applications running on FreeRTOS.

6.3 Porting for EFM32 (Energy Micro)

As is shown in Figure (6.2) the new FreeRTOS configures a RTC as main timer of the system. Basically, the changes applied inside the kernel are strictly coupled to the *timer* setup and *RTC handler*. The functions changed in FreeRTOS are:

- **prvSetupTimerInterrupt**: It configures the kernel timer in order to generate the tick interrupt at the required frequency (`configTICK_RATE_HZ`).

The timer must be configured before the scheduler starts to schedule tasks.

- **RTC Handler**: It provides to update the internal time variable (xTick-Count) and reset the tick interrupt.

6.3.1 Changes in prvSetupTimerInterrupt

In the original version of FreeRTOS, *prvSetupTimerInterrupt* configures the internal (SYSTICK) registers of CORTEX-M3. While, in the new version it sets the RTC parameters and the time (resolution) interleaved between two ticks. The new function is shown in Listing (6.1).

```
 1 /*
 2  * Setup the RTC timer to generate the tick interrupts at the required
 3  * frequency.
 4  */
 5 void prvSetupTimerInterrupt( void )
 6 {
 7     /* Enable the clock source for RTC */
 8         CMU->HFCORECLKEN0 |= CMU_HFCORECLKEN0_LE;
 9         CMU->OSCENCMD      = CMU_OSCENCMD_LFXOEN;
10         while (!(CMU->STATUS & CMU_STATUS_LFXORDY)) ;
11 
12         CMU->LFCLKSEL &= ~(_CMU_LFCLKSEL_LFA_MASK);
13         CMU->LFCLKSEL |= CMU_LFCLKSEL_LFA_LFXO;
14 
15         /* Enable RTC clock */
16         CMU->LFACLKEN0 |= CMU_LFACLKEN0_RTC;
17 
18         /* Set Prescaler. Prescaling is 1 */
19       CMU->LFAPRESC0 &= ~(_CMU_LFAPRESC0_RTC_MASK);
20         CMU->LFAPRESC0 |=   CMU_LFAPRESC0_RTC_DIV1;
21 
22     /* Define the time for next tick */
23     RTC->COMP0 = ((RTC_FREQ/PRESC)/configTICK_RATE_HZ);
24 
25         /* Wait until all registers are updated */
26         while (RTC->SYNCBUSY & RTC_SYNCBUSY_COMP0);
27 
28       /* Set lowest priority (7)*/
29         NVIC_ClearPendingIRQ(RTC_IRQn);
30         NVIC_SetPriority (RTC_IRQn,7);
31 
32         /* Enable interrupt*/
33         RTC->IEN = RTC_IEN_COMP0;
34         NVIC_EnableIRQ(RTC_IRQn);
35 
36         /* Start Counter */
37         RTC->CTRL = RTC_CTRL_COMP0TOP;
38         RTC->CTRL |= RTC_CTRL_EN;
39         while (RTC->SYNCBUSY & RTC_SYNCBUSY_CTRL);
40 }
```

Listing 6.1: New prvSetupTimerInterrupt for FreeRTOS with RTC

Initially the function sets the internal registers of EFM32 to enable the RTC clock (line 7-16). The RTC may also use a prescaling factor (line 18-20) that is not required in this case (prescaling is widely used and discussed in the next chapter). The function configures the number of clock cycles necessary to accomplish a tick (line 23). As happened in SYSTICK, the tick handler must have the lowest interrupt priority. Thus, using the CMSIS framework it is possible to set and enable a lowest priority for RTC (line 28-34). After that, the RTC is being started (line 36-39) and it will periodically interrupt FreeRTOS to notify that a new tick has just elapsed.

6.3.2 RTC Handler function

As already explained in section 2.4.2, the RTC provides two possible compare registers. Now COMP0 is used to handle the tick interrupt. The new timer handler is shown in Listing (6.2).
Therefore, every time the kernel defines (line 7-9) a new wake up time, it loads in this register the number of clock cycles necessary to trigger a new tick. The number of clocks is the division between `RTC_FREQUENCY` and `configTICK_RATE_HZ` (see section 6.2 for more details).
As happened in SysTick module, it is necessary to increment (line 16-21) the variable (xTickCount) containing the number of ticks elapsed since the system is started. In addition, the RTC Handler must also issue (line 11-14) a pendSV (context switching) request (see section 3.2.3).

```
 1  void RTC_IRQHandler  ( void )
 2  {
 3    unsigned long ulDummy;
 4
 5      if ( RTC->IF & RTC_IF_COMP0 ){
 6
 7         /* Set next tick time */
 8         RTC_IntClear(RTC_IFC_COMP0);
 9         RTC_CompareSet(0,(RTC_FREQ/configTICK_RATE_HZ));
10
11         /* If using preemption, also force a context switch. */
12         #if configUSE_PREEMPTION == 1
13            *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
14         #endif
15
16       /* Increment xTickCount */
17         ulDummy = portSET_INTERRUPT_MASK_FROM_ISR();
18         {
19            vTaskIncrementTick();
20         }
21         portCLEAR_INTERRUPT_MASK_FROM_ISR( ulDummy );
22      }
23
24  }
```

Listing 6.2: RTC Handler used to configure the tick timer

6.3.3 How to configure Power Manager Module (GPMF)

The new solution of FreeRTOS adds some more parameters inside GPMF. The values to configure in `PowerManager.h` module are:

- `RTC_ FREQ`: Standard value of RTC frequency is 32768;

- `PRESC` (if any) : Default value of prescaler is 1;

It is recommended to use 128Hz for a resolution of 8ms (real is 7.8ms) and 1024Hz for a resolution of 977 µs. If a higher resolution is required, 62.5ms can be achieved using a frequency of 16Hz. The resolution value (`configTICK_RATE_HZ`) must be configured in `FREERTOSconfig.h`.
In order to avoid the problem of truncation (see section 6.2) it is also important to adjust the value of `portTICK_RATE_MS` (inside `portmacro.h`) to 1024. This allows a correct division between `configTICK_RATE_HZ` and `portTICK_RATE_MS`.

6.4 Crystal Oscillator accuracy

For some RTOSes is important (see Chapter 3) to schedule tasks within a limited amount of time. Thus, the role of using a precise system timer module represents the main concern when there is a need to guarantee the reliability of the whole system. In this section, is presented a design issue that may hurt the correct behaviour of the system.
EFM32 DVK provides two Low Frequency Oscillators called LFXO and LFRCO. Since LFXO is more stable and precise than LFRCO, therefore, RTC uses the LFXO. Although LFXO might enhance the precision of RTC, it does not provide a high accuracy over a wide temperature range.

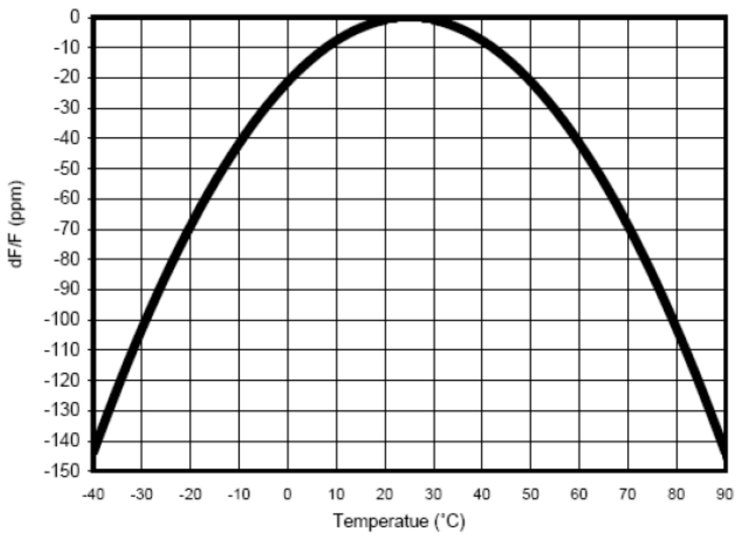


Figure 6.3: Typical Tuning Fork Crystal Frequency vs Temperature[25]

As is shown in Figure (6.3) the crystal's frequency characteristic depends on the shape of the crystal. The crystal is cut such that its frequency behaves as a parabolic curve centered at 25 °C . The curve can be modeled with the following equation:

$$F = F_0[1 + \beta(T - T_0)^2]$$

Where F_0 is the typical parabolic coefficient for an RTC oscillator, usually is equal to -0.04[ppm/T^2]. Whereas T_0 is the turnover temperature and T is the temperature. F_0 is the frequency deviation at room temperature (+25 °C). The response of crystal's frequency is strictly dependent of the above factors.
As Figure (6.3) shows, the accuracy slumps at the extreme of temperature's axis. A crystal oscillator of this type loses 8 minutes per year at 20 degrees Celsius above (or below) room temperature[26]. Hence, it is not safe to use this kind of oscillator in environment that suffers rapidly changes of temperatures. Indeed, it might compromise the reliability of the whole system. In order to avoid the above issue, some hardware or software solutions might be used to overcome this problem. Some of these solutions are reported below [25][26]:

- Crystal screening;
- Calibration registers;
- Temperature compensation;

6.4.1 Crystal screening

This solution delegates the burden to manufacture a precise crystal problem to suppliers. Hence, this requires that suppliers should minimize the deviation of crystal's frequency at room temperature during cutting operation and before shipment. Nevertheless, crystal screening strategy does not remove the problem completely. Even though the temperature accuracy will be improved to 5ppm, this is not enough to definitely remove its inaccuracy (high and low temperatures). Moreover the manufacturer can also control the crystal turnover temperature tuning with the angle of the crystal. But this is a difficult and costly operation. In practice this strategy is rarely used and does not allow to achieve good results.
One further solution is to include the tuning-fork crystal in the same package of RTC device. This strategy reduces issues regarding the layout of Printed Circuit Board (PCB) and designer's workload. Also this approach does not remove the bad accuracy problem introduced in the oscillator when the system is exposed to extreme temperatures. Adding a crystal in the same package introduces a higher cost to the whole system.

6.4.2 Temperature Compensation

It is possible to achieve a better accuracy over a wide range of temperature using a temperature compensation strategy. This compensation might be employed via *Hardware* or *Software*. The first solution requires to introduce a temperature compensated crystal oscillator (TCXO) as clock source for the RTC. While, the second option requires a periodic measurement of temperature. According the temperature sampled, is possible to adjust the RTC clock source via software (using a look-up table). The next two following sections outline the main *pros* and *cons* using these two approaches.

6.4.2.1 Software temperature compensation

This strategy does not attempt to alter the parabolic curve of oscillator, but it adjusts the time according to the current temperature. The goal of this solution is to add or subtract clock cycles according to the chip temperature. Moreover it requires to use a temperature sensor (as close as possible to the RTC) that regularly samples the temperature of chip. Afterwards, the sampled value is used as index in a look-up table to retrieve the corresponding frequency deviation. This table contains a map of data presented in Figure (6.3).
Unlike the above solution, software compensation guarantees more accuracy and it also does not introduce an extra-fabrication cost. A careful evaluation of this strategy outlines some possible drawbacks encountered when it is being used in a real environment. First of all, it is not possible to achieve a good accuracy at the extreme temperatures. Secondly, it is required to periodically measure the RTC temperature, therefore introducing an unavoidable overhead in the whole system. The adjustment of curve is not performed in real time but with a certain frequency. As last downside, some amount of non-volatile memory is also required (wasting energy) to keep the calibration data accessible during runtime.
Because each crystal has a different behavior, a custom prior calibration is needed for each oscillator. In some cases this might represent a further burden for the programmer.

6.4.2.2 Hardware temperature compensation

Another option to better enhance the accuracy of crystal oscillator is to make use of a temperature-compensated crystal oscillator (TXCO). The goal of this oscillator is to definitely remove the imperfection of crystal oscillator when it is employed in extreme environment. Indeed, this new oscillator provides good accuracy and it automatically compensates the frequency error.
Figure (6.4) illustrates how the compensation occurs in TXCO. The red curve represents the affected frequency and dashed line shows how it is being compensated in order to achieve a better frequency (green curve). Recalling the Figure (6.3), now the parabolic curve is flatten as much as possible to 0ppm over -40 °C and 85 °C.
On the market it is possible to find different TXCO with a different frequency stability. The accuracy might vary between 1.5ppm and 5ppm over -40 °C and +85 °C. The cost of these oscillators vary between 4-15 €. These oscillators are widely used in embedded applications, such as: *GPS/Telematics devices, handheld devices, power metering, security systems and medical patient monitoring.*

As it is mentioned in section 6.4.1 it is also possible to integrate the crystal, inside the RTC package. This represent the best solution on the market. Because the single package combines either the advantages of using a TXCO (highly accurate) and also avoiding the design workload burden. On the contrary, the main

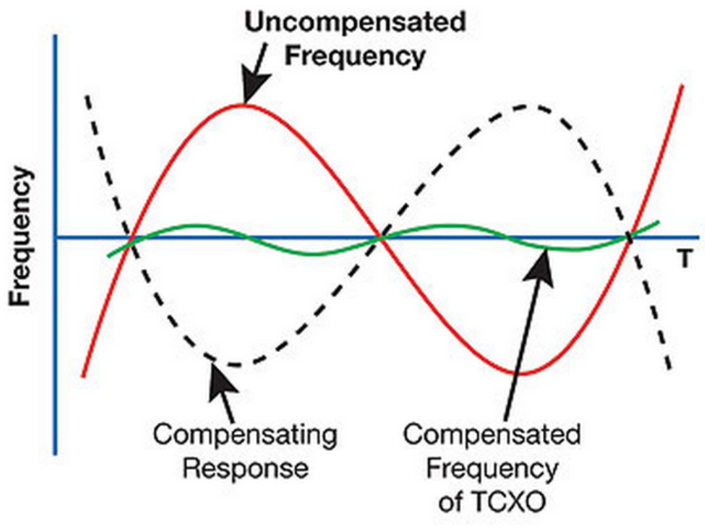


Figure 6.4: TCXOs compensation [27]

downside of this solution is the cost of the whole package.

6.5 Goals achieved

This chapter represents the first way to enhance the high energy impact introduced by systick timer. After the development of a power energy framework (GPMF), the new RTC solution allows to greatly reduce the amount of energy wasted. Also an Energy Micro's customer has implemented for himself an approach like this. He tried to use a RTC in order to minimize the power impact involved using a systick timer.
This chapter tries to fulfill the initial concerns explored in section 1.1. Finally, the RTC solution accomplishes the goal number two presented in section 1.3.

Chapter 7

Tickless Kernel for FreeRTOS

The aim of this section is to improve the solutions proposed in previous chapters. Combining the Energy Management Framework and the advantages of using a Totally tickless system allows to achieve good results in terms of energy saved.
The chapter has been organized as follow, section 7.1 presents the advantages of using a Tickless strategy, section 7.2 explains how the system can be awakened when it is sleeping. While section 7.3 discusses the drawbacks of the Tickless solution proposed by Martin Tverdal. Moreover, sections 7.4, 7.5 explain how the *Tickless Framework* works and how it is used for EFM32. Finally, section 7.6 deals with a drawback of using the Tickless Framework.

7.1 Advantages of using a Tickless System

As I described in previous chapter, the main benefit of using a RTC helps the whole system to save more energy. But the RTC solution still does not allow to definitely remove the overhead introduced by the original FreeRTOS. This chapter aims to remove the overhead of the tick timer introduced in the idle task. Recalling Figure 6.2, it is notable that during idle task the system is periodically awakened to update the content of the xTickCount variable. The tickless version removes all ticks occurred when the core is sleeping. In practice, a Tickless approach allows to place the core in the lowest available sleep mode and awake the MCU only when the next task must be scheduled. Therefore, the core is being awakened only when it is needed (save more energy). The amount of energy saved depends from the granularity used by the scheduler. The wasted energy increases with the lower resolutions. Figure 7.1 shows how the system would behave if we use a Tickless approach. The system is never interrupted when it runs in one of the power modes. On the other hand, the Tickless strategy introduces a slight overhead when the MCU is moved in sleep mode. Next sections explain when it is appropriate to use a Tickless approach instead of keep using a normal RTC solution.

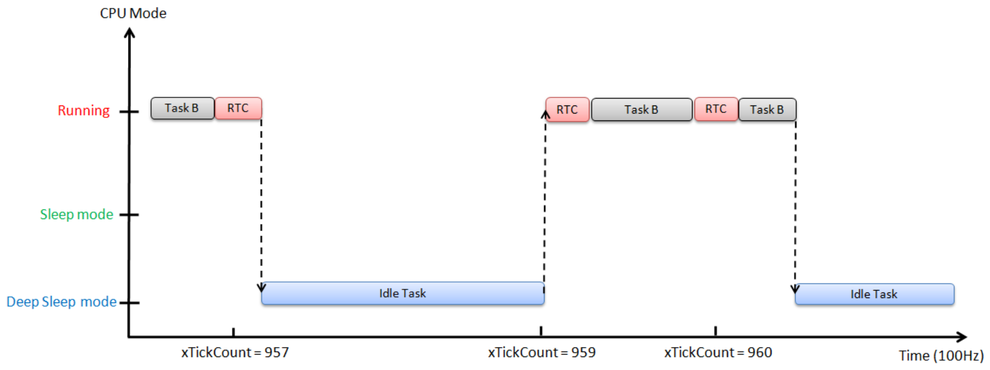


Figure 7.1: FreeRTOS with a Tickless module

7.2 Ways to awake the core from sleeping state

When the core runs in one of the power modes, the RTC is programmed to awake the MCU only when the next task occurs. But, if an asynchronous interrupt occurs when core is sleeping, FreeRTOS must handle its ISR and update the xTickCount variable. Therefore, in order to manage efficiently the RTC we have to add some new features inside the GPMF. The system can wake the core up in one of the following ways:

1. Task or co-routine to schedule;
2. An asynchronous ISR occurs;
3. An asynchronous ISR unblocks a Ready Task;

7.2.1 Task or co-routine to schedule

This is the case that happens every time the system is pushed in the idle state. The system before to go sleeping retrieves the time of next ready task or co-routine. The RTC is configured to awake FreeRTOS at the time to schedule the task/routine. The main limitation of this case is linked to the time that the RTC can stay in the idle state without that an overflow of its internal counter occurs.

7.2.2 An asynchronous ISR occurs

This is the scenario when an event interrupts the core before the expected wake up time. Since this is an asynchronous event it can not be predicted in advance. After the execution of its ISR, the xTickCount variable must be updated with the time elapsed since the core was moved in the sleep state. Afterwards, the MCU returns to sleep only if there are not other ISRs waiting to be performed. If any, the ISRs are scheduled with tail-chaining technique [28].

7.2.3 An asynchronous ISR unblocks a Ready Task

This case behaves as the previous one with the difference that the ISR unblocks a task after its execution. If the ISRs make use of the FreeRTOS API they might unlock a task that was waiting on an event (e.g. queues, semaphores). Consequently FreeRTOS schedules the unblocked tasks. Yet, the xTickCount variable must be updated before starting to schedule new tasks. Usually, FreeRTOS requires that users have to call at the end of each ISR a function (`portEND_SWITCHING_ISR()`) that notifies to the scheduler that a task has been unblocked.

7.3 Drawbacks of previous Tickless solution

The goal of this section is to outline the drawbacks of the solution proposed by Martin Tverdal. This section does not aim to diminish the work did by Martin Tverdal but to improve it with a more accurate approach.

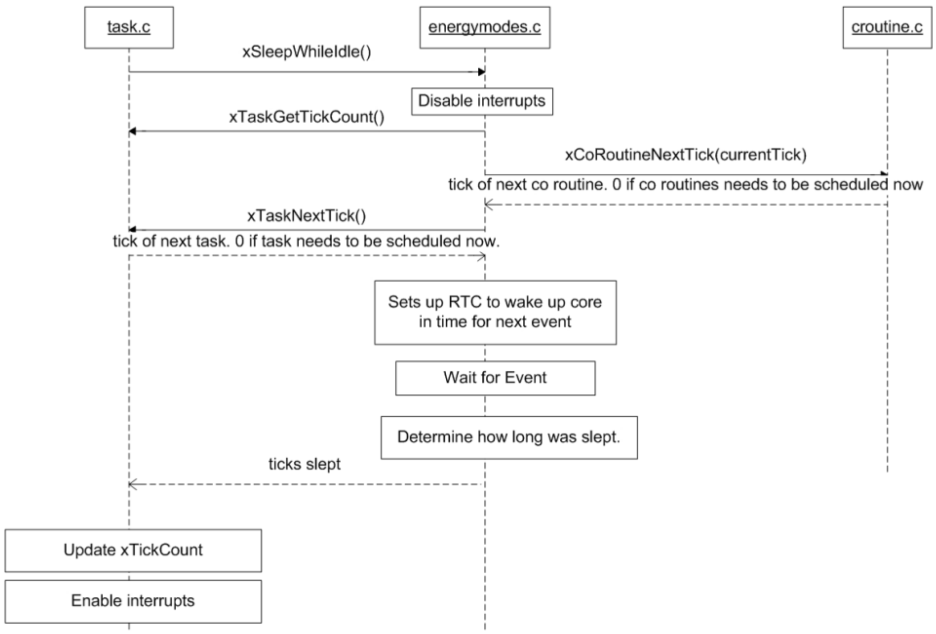

task.c
energymodes.c
croutine.c
xSleepWhileIdle()
Disable interrupts
xTaskGetTickCount()
xCoRoutineNextTick(currentTick)
tick of next co routine. 0 if co routines needs to be scheduled now
xTaskNextTick()
tick of next task. 0 if task needs to be scheduled now.
Sets up RTC to wake up core in time for next event
Wait for Event
Determine how long was slept.
ticks slept
Update xTickCount
Enable interrupts

Figure 7.2: Sequence Tickless diagram [3]

I want to recall the sequence tickless diagram shown in his thesis. In Figure 7.2, the diagram shows that the system disables all interrupts when it goes in sleep modes (except those above `configMAX_SYSCALL_INTERRUPT_PRIORITY`). The system checks the time of the next upcoming task or co-routine and it uses this value to set the wake up time of the RTC. When an event occurs, the system calculates (using the RTC counter) the time slept in a power mode. Since the core in tickless mode does not use the same time scale used in running mode it converts the RTC (RTC counter) time to FreeRTOS time (xTickCount). This overhead is added every time the system is being awakened. As consequence, the execution of the event (interrupt) is also delayed proportionally to the time spent to translate this value. After that, interrupts are enabled and all pending ISRs are being executed. The main drawbacks of this approach, are:

- Overhead introduced at wake up time;
- Delayed time for pending ISRs;
- Issues of switching between FreeRTOS and RTC (viceversa);
- Ticks lost due to the execution of many ISRs;
- Maximum time in sleep state (without awaking);
- Tickless threshold;
- Pitfalls of checking next upcoming routine;

The first two problems have already been discussed above. While all the other drawbacks are faced more accurately in the next sections. The Tickless GPMF strategy proposes a solution that fix problems listed above.

7.3.1 Switching between FreeRTOS and RTC

When the system is running, the SYSTICK module provides to update the xTickCount variable. On the contrary, when the core is sleeping and it is being awakened by an event, the system retrieves the time elapsed in sleep state reading the internal value of the RTC counter. Therefore, when the system comes back in running state it updates the xTickCount variable with the number of ticks elapsed since the core was sleeping.
Using this approach, the system uses two different time scales. A further problem is due to the error introduced switching from FreeRTOS to RTC and viceversa. In fact, Martin tried to overcome this problem truncating the number of RTC ticks when the system returns in running mode. This truncation does not guarantee that the system takes into account all elapsed ticks. My idea is to use the RTC counter as the main timer for the whole system. In this way the system is not forced to switch from different time scale and it is also more accurate. In addition, as explained in chapter 6.1 the RTC introduces a series of advantages in FreeRTOS.

7.3.2 Ticks lost at wake up time

One of the main drawback of Martin's solution is the way such his tickless approach handles the case when several ISRs occur simultaneously at wake up time. As I said above, an ISR can issue a context switching after its execution. When the MCU is awakened from several events it must handle all pending ISRs (according

their priority) before to schedule the unblocked tasks. The system might spend a considerable amount of time to handle all ISRs and the time elapsed during their execution is not well evaluated because the RTC Handler will be handled only one time.
In practice, since the RTC has the lowest priority it will be performed as last interrupt. As consequence, if a tick was occurred for other three times, those ticks were lost because the system had to handle the first tick. Therefore, FreeRTOS would update xTickCount variable only one time instead of three. The next following tasks would be executed with a delay equal to the ticks lost. This situation might occur when more than one devices (ADC or DAC) perform with a *fine grained* resolution.
For example, when the system is performing the ISR of the ADC, if a request arrives from DAC it puts the latter request in a pending list. The system never goes outside of the ISRs execution as long as the two interrupts are not overlapped anymore.

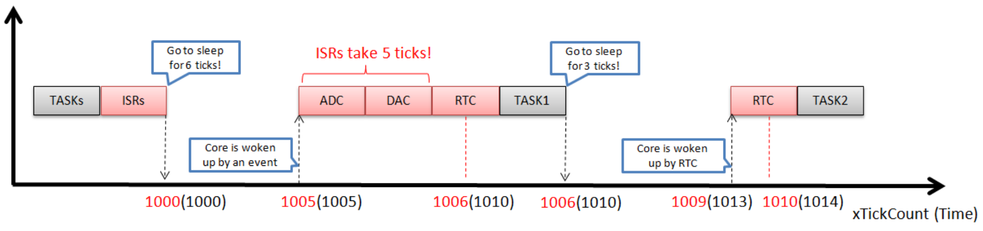


Figure 7.3: Wrong Tickless solution

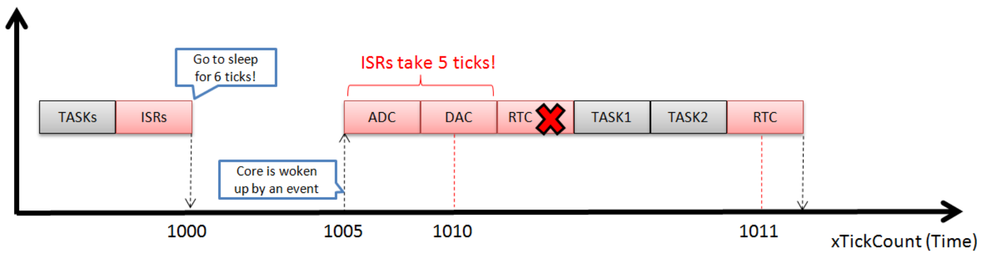


Figure 7.4: Tickless solution

The example in Figure 7.3 illustrates how the system behaves using the tickless approach developed by Martin. While the Figure 7.4 shows how the system would behave handling the time correctly. For the sake of simplicity, let's assume that ADC has higher priority than DAC and RTC. DAC has higher priority than RTC. $Task_1$ will be scheduled at tick number 1006 and $Task_2$ at tick number 1010.
In Figure 7.4 the time is represented in *red* using the wrong time and in *black* the real time. At tick number 1000 the system goes to sleep for 6 ticks. Two events

(ADC, DAC) as well as the RTC handler occur simultaneously after 5 ticks. Let's assume that ADC and DAC are overlapped for 5 times because they sample a value for many times. The system consumes 5 ticks without is being take it into account. When the RTC Handler is performed, it updates the xTickCount to a wrong value 1006 instead of 1010. Right now, $Task_2$ can be scheduled but since the xTickCount does not contain the correct value it will move the system to a sleep mode for others 3 ticks. This problem causes a need to schedule $Task_2$ four ticks later (missing its deadline). This is a very dangerous situation that can compromise the reliability of the system. Moreover the system does not recognize that it is scheduling tasks with some delay. The delay *increases* in respect to the *resolution* of the system. For example, if the resolution of the first example was 10 ms FreeRTOS would schedule $Task_2$ with a delay of 40ms.
On the other hand, Figure 7.4 shows the correct functioning of the previous example. Also in this example both tasks are scheduled with some delay because the ADC and DAC defer their execution. But $Task_2$ is scheduled immediately when it is ready to be scheduled (1011 instead of 1014). The reasons why the RTC Handler is cancelled are better explained in next sections.

7.3.3 Maximum time in sleep state

The main goal of Tickless approach is to keep the system in the idle state as long as possible. As I described in previous chapter, FreeRTOS uses an internal variable (COMP0 see section 2.4.2) to store the number of clock cycles needed to accomplish the next wake up time (tick rate). The RTC counter (CNT) variable uses only 24 bits, therefore, a number higher than 2^{24} is not allowed in the RTC. Since the Tickless solution requires to store a number of clock cycles greater than those needed for a tick rate. It stores 2^{24} when the Tickless requests to store a number that exceeds the maximum value. Therefore, the RTC is awakened every 2^{24} as long as the Tickless reaches the next idle task. Using a resolution of 100 µs the RTC can stay in idle state at most for 28 minutes. If an application takes longer idle tasks (e.g. days, hours) the system would awake the core for several times before it reaches the next routine/task to schedule. I think that awaking the core for several times might consume more energy (mostly for lower resolution).
The strategy proposed to overcome this problem is to use the prescaler provided from the RTC. The *prescaler* allows to maximise the time in idle state because it divides the original RTC frequency (32768) with a prescaler factor. In section 2.4.2 is shown a table that illustrates for each prescaler factor the corresponding *resolution* and *overflow*. The overflow represents the maximum time that the MCU can stay in idle state. It is necessary to find a *trade-off* between system's resolution and maximum time of the idle state.
As said in section 2.4.2, the prescaler also introduces an increase of the power consumption. In order to understand which solution achieves better results, the new *Tickless Framework* proposes both solutions (with prescaling and without prescaling).

7.3.4 Tickless threshold

Even though the Tickless approach aids to save more energy than a normal system. In some cases, it is not always appropriate to use a Tickless strategy (drawback proposed in section 7.3) because it consumes more energy. As I said before, if many ISRs occur at the same time the system might schedule tasks with some delay. Therefore, it is not safe to go in Tickless mode when the idle task takes few ticks. A *threshold* value called break even point defines the minimum ticks needed to cover the effort introducing the Tickless features.
The threshold value is not easy to calculate because it might be different among different platforms. Moreover, the break even point is chosen according to the tick system resolution and some external devices. Next section presents an example how to configure the break even time for EFM32.

7.3.5 Checking next upcoming co-routine

The system before to go in sleep mode it checks if there are any tasks or routines ready to be executed. Since task and routine make use of two different modules to handle their internal framework. The time task variable (xTickCount) is not always kept updated with the time routine variable (xCoRoutineTickCount). Therefore, the system might take a wrong decision if these two variables are not synchronized each others.
It is necessary to update the time routine variable (xCoRoutineTickCount) whenever tasks update the xTickCount. Since the routines are scheduled only when there are not tasks, the number of ticks to update in xCoRoutineTickCount can be very large. Hence, every time the system returns to routine it would spend a lot of time to perform all missed ticks. This causes a waste of energy and time. In order to avoid this problem, every time a tick timer occurs (RTC Handler), the new Tickless FreeRTOS updates either the time variable of task and also routine.

7.4 Tickless GPMF

This section describes the functioning of the new *Tickless GPMF module*. As I said in section 7.3.3, this new Framework provides two different versions of FreeRTOS, one *with* prescaling and the other one *without* prescaling. The two solutions have the same code except when the Framework uses the RTC. The following sections point out the most important functionS developed for the new Framework.

7.4.1 Tickless GPMF Module

The Tickless GPMF module (Figure 7.5) is divided in two parts, such as *GPMF* and *Tickless Framework*. The GPMF module uses its internal functions (see section 5.4.1) to interact with Tickless Framework.
On the other hand, the Tickless Framework manages the case when the system makes use of the tickless features. As I described in section 5.4.1, the whole module provides an interactions with FreeRTOS (*system side*) and programmers (*user*

side). The *Tickless Framework* mainly interacts with system side because before to go in sleeping mode it needs to acquire several information from the system. Basically, the main goal of the tickless system is to reduce as low as possible its complexity. The proposed Tickless Framework interacts with programmers exploiting an existing API provided by FreeRTOS (`portEND_SWITCHING_ISR`). This approach allows programmers to use efficiently the new user's APIs. The GPMF module interacts with programmers and FreeRTOS as happened before.

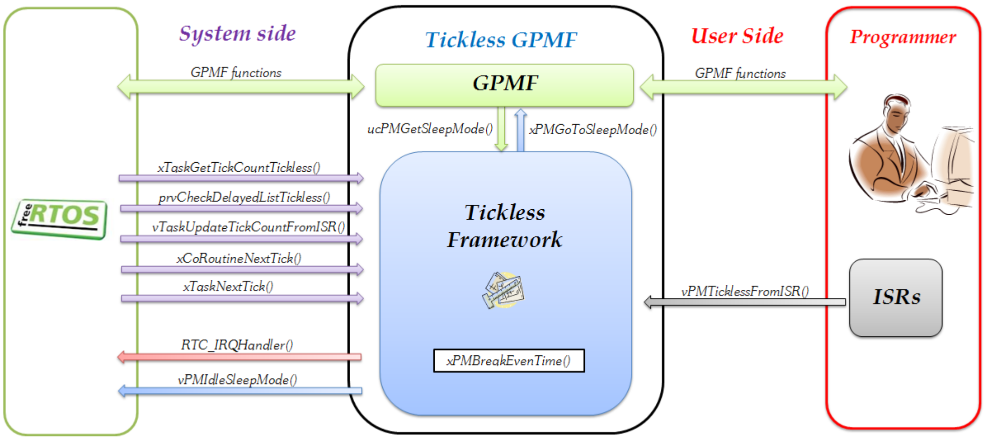


Figure 7.5: Tickless GPMF module

The part regarding the *system side* is the most important part of the whole framework. New functions have been implemented inside the task and routine module. In addition, also the tick handler (RTC) plays an important role in the new RTOS. Before to move in a sleep mode, the *Tickless Framework* retrieves the time of the next upcoming task and routine to be scheduled. FreeRTOS is held in sleep state as long as there are not ready tasks. The function that checks which is the time when the next task must be scheduled is called `xTaskNextTick()`. While `xCoRoutineNextTick()` checks which is the next routine to schedule. Since the system can sleep for several ticks, when the core is awakened it has to keep updated the xTickCount variable. Therefore, a function that performs an immediate updating has been implemented, it is called `vTaskUpdateTickCountFromISR()`. The function can be called only inside the body of an ISR.
Furthermore, this new framework controls the number of ticks needed to amortize the effort involved by tickless. If the system does not satisfy the threshold, it continues to run with a normal tick rate. The name of function that checks the *break-even point* is called, `xPMBreakEvenTime()`. This function needs to know the value of the xTickCount variable. This is performed by `xTaskGetTickCountTickless()`. All the above functions (except for the UpdateTickCountFromISR) are used when FreeRTOS notifies that an idle task has occurred. Therefore, they are employed in a unique function called `vPMIdleSleepMode()`. When the system is moved in a sleep state, it is awakened by `RTC_IRQHandler()`. If the tickless is enabled,

the xTickCount is being updated with the number of ticks elapsed since the core was sleeping. Otherwise, it only increments the xTickCount variable. When the system runs for a long time in sleep state the co-routine are not synchronized with the xTickCount variable. Therefore, the `prvCheckDelayedListTickless()` provides to keep updated the co-routine module.

The part regarding the *user side* allows to help the tickless module when the system itself is not able to manage alone the new framework. In order to reduce the complexity of the system level, FreeRTOS receives some notifications sent by programmers. It provides an internal API called `portEND_SWITCHING_ISR` used only inside the body of an ISR. It is used to force a context switching when the ISR has unblocked a task after its execution.
The idea is to substitute `portEND_SWITCHING_ISR` with a new internal function called `vPMTicklessFromISR()`. The new function is changed in order to manage either the normal case and the *Tickless Framework*. In practice, the only burden of the programmer is to use this new function in place of the old function implemented in FreeRTOS.

7.4.2 Possible states for Tickless GPMF

The new *Tickless Framework* solution works using different states during its execution. This section describes the meaning of each state and how they can be employed inside the new framework. The idea is to split the execution of the application in well-known *states* because in this manner the system knows which is the best decision to undertake every time. We divided the Tickless GPMF solution in four states, such as:

RUNNING This state includes all tasks and ISRs operations. It identifies that the system is in running mode and it is being interrupted every tick.

TICKLESS This state indicates that the system does not use anymore the tick interrupt. Now the system is moved in one of the sleep modes and it will be awakened when one of the following cases presented in section 7.2 occurs.

PREEMPTED This state indicates that the system was in TICKLESS mode and it was interrupted by an asynchronous event (except for RTC). This event made a request of performing a context switching after its execution.

NOPREEMPTED This state indicates that the system was in TICKLESS mode and it was interrupted by an asynchronous event (except for RTC). After the execution of the ISR it is not required any context switching.

Both `PREEMPTED` and `NOPREEMPTED` states can be interleaved each other. These two states can not occur when the system is in `RUNNING` state because they are always preceded by `TICKLESS`. Moreover these two states are able to retrieve the time elapsed since the core was moved in TICKLESS state. This represents the

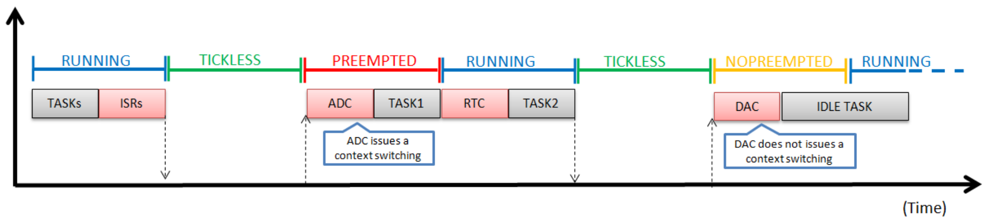


Figure 7.6: States arisen using Tickless Framework

main advantage of using the Tickless GPMF approach because it overcomes the problem discussed in section 7.3.2. The system becomes `RUNNING` at the beginning of an idle task or at the end of a RTC Handler. While it becomes TICKLESS only when the idle task is sure that there are not other tasks/routines to schedule.
Figure 7.6 shows an example how these states are handled during run time. Initially, the system is `RUNNING` until the core is moved in a sleep mode (`TICKLESS`). The ADC wakes the core up and it also needs to issue a context switching after its execution, therefore the state is moved in `PREEMPTED` state. The next time that the core is awakened by the DAC it does not issue a context switching after its execution, hence its state is `NOPREEMPTED`.

7.4.3 Interaction of Tickless GPMF with FreeRTOS (vPMIdleSleepMode)

This section explains how the Tickless GPMF module interacts with FreeRTOS. The *Tickless Framework* is used only when there are not ready upcoming tasks/routines to schedule. FreeRTOS has been modified that when an idle task occurs it calls *vPMIdleSleepMode* (Tickless Framework) and it continues to behave normally when it is being awakened. The state of the GPMF module is stored inside the `xPMTicklessENB` variable. One of the main benefit of using Tickless GPMF is to add an overhead proportional to the type (see sections 7.2) of the wake up time. For example, if an ISR issues a context switching it involves a higher overhead than awaking a core for scheduling a ready task.
Figure 7.7 illustrates how FreeRTOS and *Tickless Framework* interact each other. On the *left side* are represented steps performed by FreeRTOS, while on the *right side* are represented those stages performed before to move the core in `TICKLESS` state. FreeRTOS calls *vPMIdleSleepMode* when there are not tasks or routines ready to be scheduled.
Initially, the function disables either interrupts and task scheduler because the next following instructions must be executed in mutual exclusion. Any interrupts arisen at this time are moved in pending state (if the core is moved in sleep state with pending ISRs it will be immediately awakened). Now, the function retrieves the time when the next tasks or routines will be scheduled. According to the values of next task/routine, it is possible to check if it is affordable to enter in a TICKLESS state or keep using the running mode. This job is accomplished by a *Break even*

function (`xPMBreakEvenTime()`). If a task or a routine will be scheduled in few ticks, the system is left in running mode and it will not use the tickless features. Otherwise, the system is configured as Tickless mode and the core is moved at the lowest possible power mode. The power mode is retrieved from GPMF module.
The system is put in sleep state for the time needed to wait the next upcoming task or routine. A core will always be awakened by two possible events, such as an *asynchronous event* (generic ISR) or time *expired event* (RTC Handler). Any event, immediately causes for enabling all interrupts and resume the task scheduler. After that, if the awakened event was an *expired event*, the RTC Handler provides to update the xTickCount variable with the time slept in tickless mode. In addition, the state of FreeRTOS is changed in `RUNNING` mode.
On the contrary, if the awakened event was a generic ISR it must update the state of the system (`PREEMPTED` or `NOPREEMPTED`) and the xTickCount variable with the current time. It is **compulsory** that every ISR (except for RTC) must call in their body the `vPMTicklessFromISR` function. It will provide to update the state of the system and the xTickCount variable. The system does not return to FreeRTOS scheduler until all pending ISRs are executed. *Note*: the RTC Handler is always the last ISR to be executed because it has the lowest priority. The system performs the ready task (due to *time expired* or an *asynchronous event* with preemption) or the idle task (*asynchronous event* without preemption) when there are not other ISRs to execute. The Appendix B.1 reports the code that implements the flow diagram illustrated in Figure 7.7. The code also outlines the differences between *prescaling* and *no prescaling* solution.

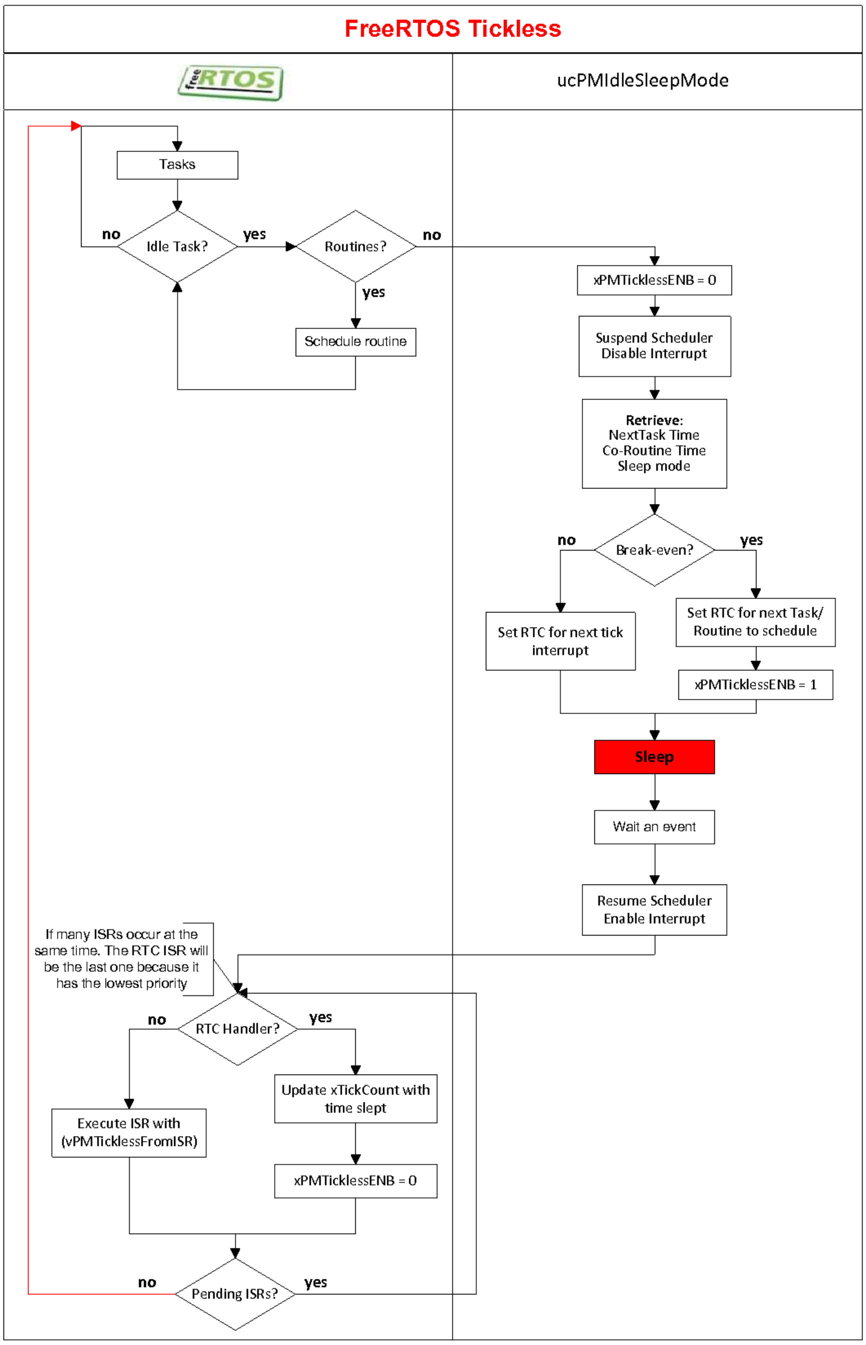


Figure 7.7: Interaction between FreeRTOS and Tickless Framework

7.4.4 xPMBreakEvenTime

Even though the *Tickless Framework* allows to save a considerable amount of energy it is not always convenient to use it when the system waits only for few ticks. In fact, if in the meanwhile some interrupts occur they could delay the upcoming tasks (the effort of tickless is not fully compensated). Therefore, before going in sleep mode, the *Tickless Framework* checks if it is worthwhile to go sleeping or keep running. This function uses a *threshold* value (`MinTICK`) to determine which is the minimum numbers of ticks necessary to compensate the effort introduced from the Tickless Framework. The threshold value is a bit tricky to calculate because it might dependent by several factors.
Actually, EFM32 uses a threshold value equal to two and it depends from the RTC counter. If the value of the *nextTask* or *nexRoutine* are equal to zero, it means that they run at the same priority of the idle task. Therefore, a system does not use the *Tickless Framework* when the *nexRoutine* or *nextTask* are equal to zero or less than threshold value. The code of this function is reported in Appendix B.2.

7.4.5 RTC_IRQHandler

The RTC Handler can be employed in two possible ways, such as *tick expiration* and *tickless expiration*. The *tick expiration* is used in place of the SYSTICK module and it allows to keep the system updated over the time. When the core receives a tick expiration it updates the xTickCount variable and it issues a context switching. Moreover, the RTC handler configures the RTC in order to awake the core for the next tick timer. On the other hand, the *tickless expiration* occurs

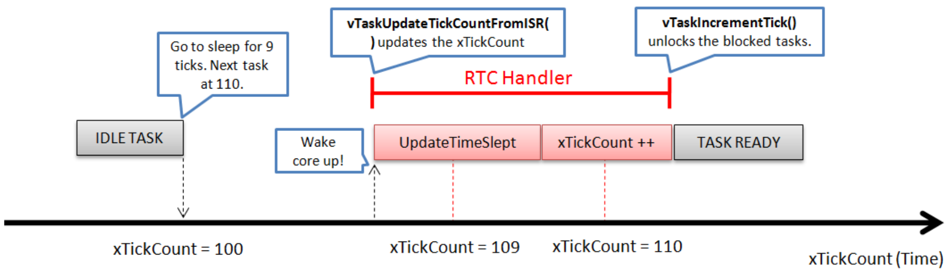

Go to sleep for 9 ticks. Next task at 110.
vTaskUpdateTickCountFromISR( ) updates the xTickCount
vTaskIncrementTick() unlocks the blocked tasks.
RTC Handler
IDLE TASK
Wake core up!
UpdateTimeSlept
xTickCount ++
TASK READY
xTickCount = 100
xTickCount = 109
xTickCount = 110
xTickCount (Time)

Figure 7.8: Updating of xTickCount when FreeRTOS uses the Tickless Framework

when the tickless mode has expired its timeout. Now, the xTickCount variable is directly updated with the number of ticks elapsed (minus 1 tick) since the core was sleeping. The function `vTaskUpdateTickCountFromISR()` provides to update the xTickCount variable. This function updates the variable with one tick less than time defined (retrieved from nextTask or nextRoutine) when the core was moved in tickless state. The reason is due to the fact that tasks are being unblocked using only the function `vTaskIncrementTick()`. The core also updates the routine module with the number of ticks missed using the function `prvCheckDelayedListTickless()`. Afterwards, the core sets the new tick timer, perform a context switching and it increments the xTickCount variable. Figure 7.8 shows how xTickCount variable is updated in *Tickless Framework*. The core must be awakened at 109 because at 110 a task is ready to be executed. Again, the core is woken up a tick before because the function that provides to unblock the tasks is *vTaskIncrementTick()*. The system changes its current state to `RUNNING` at the end of the RTC Handler. The code of this function is reported in Appendix B.3 and it is an improved version of that one proposed in 6.3.2. The code also outlines the differences between *prescaling* and *no prescaling* solution.

7.4.6 vTaskUpdateTickCountFromISR

As is described in section 7.4.5 this function is used to update the xTickCount variable when more than one ticks must be added. This allows to update the variable on-shot instead of updating the variable every tick. A problem arises when the variable overflows because it restarts the counter from 0. If this case is not handled properly, the system might miss some tasks. Furthermore, it would not be able to swap the two delayed listed (see section 3.2.1.2). When overflow occurs, the function sets xTickCount at portMAX_DELAY . From this point forward it updates the remained ticks one per time and the swap operation is performed correctly. The code of this function is reported in Appendix B.4.

7.4.7 xTaskNextTick

This function retrieves the next upcoming task to execute. It is used inside `ucPMIdleSleepMode()` to check if it is affordable to use the tickless mode. Basically, this function returns 0 if there are ready tasks. Otherwise, it returns the tick when the first task must be scheduled. This tick number is automatically stored inside `xNextTaskUnblockTime`. The code of this function is reported in Appendix B.5.

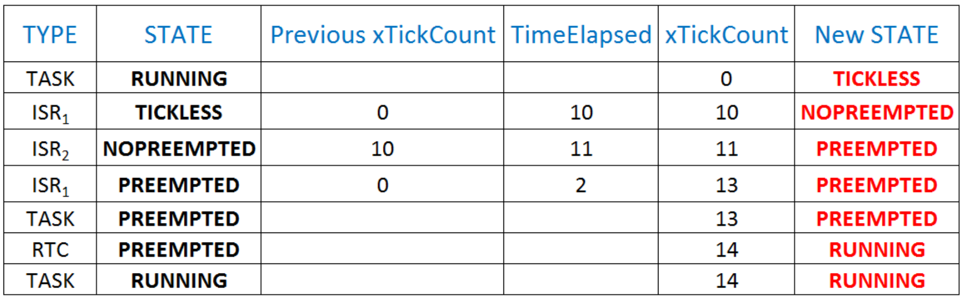
7.4.8 xCoRoutineNextTick

This function is used to retrieve the next upcoming routine to schedule. This new function updates the whole co-routine module before to get the tick of the next routine. This is the main difference from the function developed by Martin. Using his function the system was not always synchronized with FreeRTOS (see section 7.3.5). Reported in Appendix B.6.
The following functions prvCheckPendingReadyListTickless() and prvCheckDelayedListTickless() are used to update the routine module (*xCoRoutineTickCount*, *xLastTickCount* and *xPassedTicks*). They have been implemented as contribution of the *Tickless Framework* (code is reported in Appendix B.7). After their execution, the function checks that there are not ready routines to schedule with the lowest priority. Also this function returns 0 when there are ready routines to schedule. Otherwise returns the tick of the first routine to schedule.

7.4.9 vPMTicklessFromISR

This is the most important function contained inside the Tickless GPMF module. It is the only function that interacts with users and it must be used in place of the `portEND_SWITCHING_ISR`. This function must be **compulsorily** used inside the body of each ISR (except for RTC Handler). It can get in input two possible values, such as `pdTRUE` or `pdFALSE`. These values have the same meaning of the parameters given in input for `portEND_SWITCHING_ISR`. Figure 7.10 illustrates the behavior of *vPMTicklessFromISR*().
If the function is called during RUNNING state (*right side*) it issues a context switching, if required. While, if the system runs in tickless mode (*left side*) it can be awakened by an event that might require to issue a context switching (`PREEMPTED`) or not (`NOPREEMPTED`). As I said before, *vPMTicklessFromISR()* is able to retrieve the time elapsed since the core was sleeping. Therefore, the function must add some more controls in both states (`PREEMPTED`, `NOPREEMPTED`) because the time is retrieved in a different manner. The system must handle also the case when preempted ISRs are interleaved with nopreempted ISRs. It is important to remark that when the core enters in `PREEMTPED` state it can not return back to `NOPREEMPTED` state. When the ISR issues a context switching (in tickless mode), it retrieves the number of ticks elapsed since the last time xTickCount was checked. The function marks the state as `PREEMPTED` and a context switching is being issued. From this point forward, at least a task must be scheduled, therefore it needs to set the timer that will periodically preempt the task. The function also deletes the pending RTC request (if any) waiting to be performed. In fact, the RTC has already been programmed to issue the next tick.
On the contrary, if the ISR does not issue any context switching, the new state is marked as `NOPREEMPTED`. The function still retrieves the time elapsed in tickless state or the time elapsed since last ISR has occurred. The xTickCount variable must be updated regardless if the state was `PREEMPTED` or `NOPREEMPTED`.
The function uses the RTC to retrieve the time when it runs in `PREEMPTED` or `NOPREEMPTED` state. If many nopreempted ISRs occur at the same time, they are executed one after the other according to their priority. The RTC counter is reset only in `PREEMPTED` state. Figure 7.9 shows a possible scenario when ISRs require to stay in both states. For the sake of simplicity it is not reported the period of the task and are used two generic ISRs, respectively ISR_1 does not require preemption and ISR_2 with preemption.

Initially, the system enters in tickless mode at tick equal 0 and it changes its state from `RUNNING` to `TICKLESS`. After that, ISR_1 wakes the core up and the function gets how many ticks have elapsed in Tickless state and it updates xTickCount variable. The state is changed from `TICKLESS` to `NOPREEMPTED`. When ISR_2 is executed, it takes one tick to perform its job and this value is updated in xTickCount. The ISR_2 issues a context switching and the state is changed from `NOPREEMPTED` to `PREEMPTED`. From this point forward, the system can not return back to `NOPREEMPTED` (except if the system goes at once in `TICKLESS` state). The RTC counter is reset to 0 because if the task is immediately executed it has to be preempted. The ISR_1 is executed at once, it updates the xTickCount



TYPE	STATE	Previous xTickCount	TimeElapsed	xTickCount	New STATE
TASK	**RUNNING**			0	**TICKLESS**
ISR$_1$	**TICKLESS**	0	10	10	**NOPREEMPTED**
ISR$_2$	**NOPREEMPTED**	10	11	11	**PREEMPTED**
ISR$_1$	**PREEMPTED**	0	2	13	**PREEMPTED**
TASK	**PREEMPTED**			13	**PREEMPTED**
RTC	**PREEMPTED**			14	**RUNNING**
TASK	**RUNNING**			14	**RUNNING**

Figure 7.9: Example of interleaved ISRs

variable with the two ticks elapsed from when ISR_2 was executed. Now the task is ready to be executed until the tick interrupt occurs (note that the state remains still in `PREEMPTED` state). Upon the execution of the tick interrupt the state is immediately moved in `RUNNING` state and it continues to update normally the xTickCount variable. The code of this function is reported in Appendix B.8. The code also outlines the differences between *prescaling* and *no prescaling* solution.

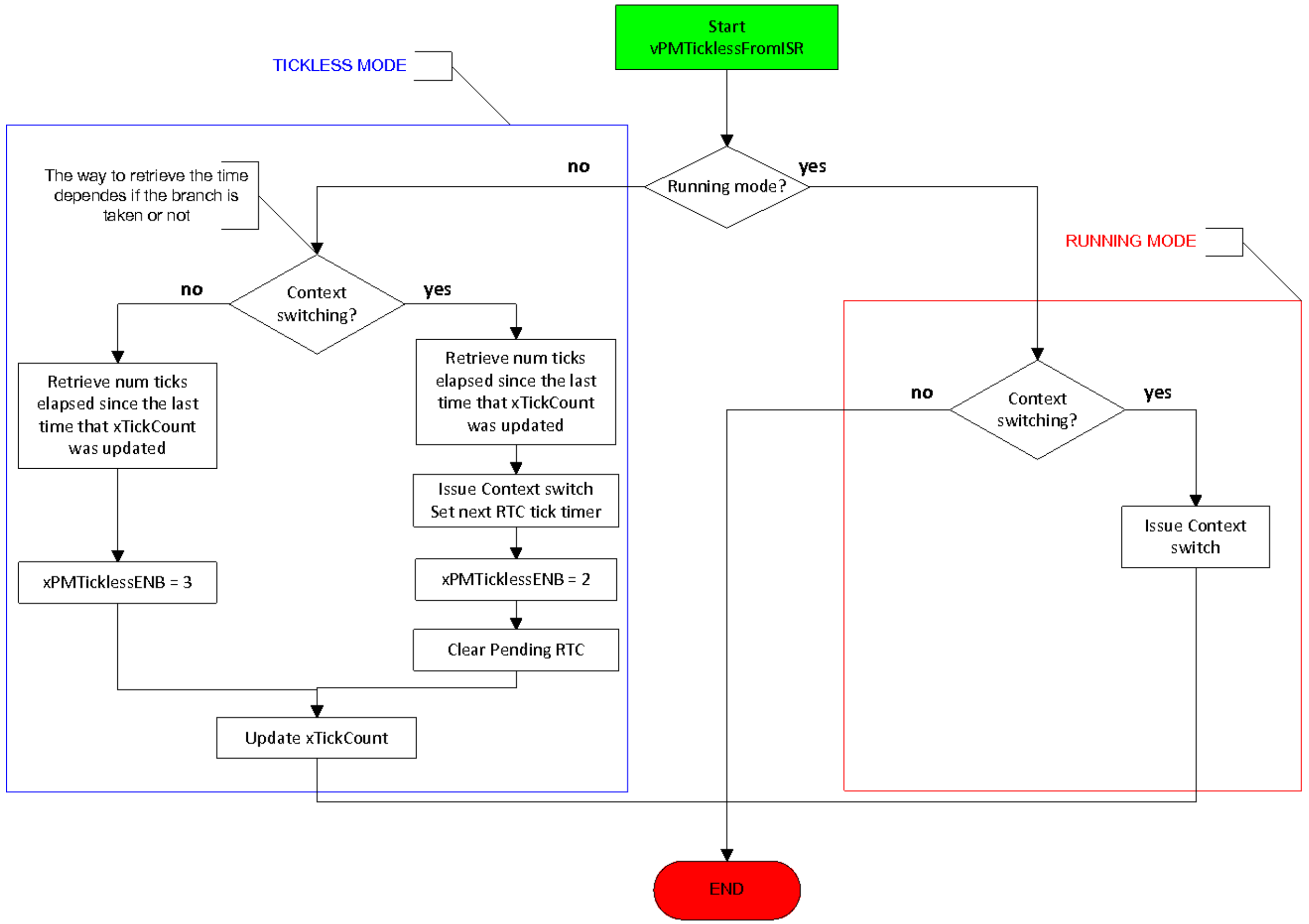


Figure 7.10: Flow sequence diagram of vPMTicklessFromISR()

7.5 How to configure Tickless GPMF for EFM32

The Tickless framework is enabled when the values of `configTICKLESS` and `configUSE_POWERMANAGER` are equal to 1. The constant `configPRESC` indicates if the Tickless Framework uses the prescaler. The constant is equal 1 when the prescaler is used. These constants are present inside `FreeRTOSConfig.h`.

`configPRESC = 1` The prescaler must be configured with the desired resolution of RTC. Table 2.6 inside the section 2.4.2 proposes the possible resolutions to use with prescaling. Function *prvSetupTimerInterrupt()* inside `port.c` needs to change its prescaler value. The value of `CMU_LFAPRESC0_RTC_DIV256` (PRESC = 8) must be changed according to the desired prescaler. For instance, with prescaler of 5 the value must be replaced with `CMU_LFAPRESC0_RTC_DIV32`.
It is also needed to change the value of `PRESC` present inside `PowerManager.h`. This value contains the resolution of the prescaler. For instance, using a prescaler of 5 (977 µs) it must be replaced with 32 and using a prescaler of 8 (7.8ms) it must replaced with 256. Finally, FreeRTOS's resolution must be configured inside `FreeRTOSConfig.h`. Section 6.3.3 explains with more details how to configure `configTICK_RATE_HZ`.

`configPRESC = 0` The prescaler is not used. The value of `PRESC` present inside `PowerManager.h` must be equal to 1 and the value of `configTICK_RATE_HZ` follows the same hints proposed before. It is not required to configure other values.

7.6 RTC's problem

The vPMTicklessFromISR allows to retrieve the time elapsed also when many ISRs are executed continuously. This function retrieves the time present inside the counter of the RTC. When the RTC reads the CNT value it might not return the right value. Sometime it returns the value with a margin error that is never higher than 2 ticks. If the system uses a higher resolution it could introduce a time drifting that might vary in accordance to the value read from the counter.
This is the reason why the RTC Handler does not use the CNT value to update the xTickCount variable (it updates xTickCount with the number of ticks computed before to move the core in sleep state). Unfortunately, when an asynchronous event wakes the core up, the only way to retrieve the time elapsed is to use the RTC value. The same problem was also analysed by Martin Tverdal in his work. Currently, the problem has been notified to my supervisor at Energy Micro and we are currently working to cope this trouble.

7.7 Summary

The solution presented in this chapter represents the combinations of those solutions explained previously. This new Framework has been implemented using a different approach than Martin's solution. While the previous solution used an approach where the overhead of the framework was completely released to the system. The new framework implements a module as flexible as possible during runtime. Finally, the new Tickless Framework has been implemented in two different ways, *with* prescaling and *without* prescaling. The reason of this choice is to understand in which way the Tickless solution performs better. It is not easy to understand without simulations which one introduces a higher power consumption. Next chapter shows the results of both solutions.

Part III

Results

Chapter 8

Simulations

"It is the weight, not numbers of experiments that is to be regarded."

- **Isaac Newton**,was an English physicist, mathematician, astronomer, natural philosopher, alchemist, and theologian.

This chapter aims to discuss the energy results achieved with a benchmark tailored developed for this thesis. The chapter outlines the advantages of using the new versions of FreeRTOS proposed in chapter 6 and 7.
It has been organized as follow, section 8.1 discusses the main tasks and functions used inside the benchmark, section 8.2 explains the tools used to measure the current consumption drawn by the MCU and external devices. Section 8.3 decribes which are the results achieved for each FreeRTOS. Section 8.4 shows a screenshot of the simulations. Finally, section 8.5 illustrates the lifetime of the benchmark running on the different versions of FreeRTOS.

8.1 Benchmark

One of the initial goal defined in section 1.3 is to develop a benchmark for EFM32. The *benchmark* is used to simulate the behavior of the different versions of FreeRTOS. In order to emphasise the importance of the idle task, this application spends most of the time running in the idle state.
The benchmark works as follow, it samples the *Ambient Light* value every 12 seconds and the value of the *internal Temperature* every 60 seconds. Moreover, using an *asynchronous event* it is also possible to retrieve and send via USART the values (X,Y,Z) of the *Accelerometer*. The event is being triggered when is pressed the button SW1 present on the DVK board. Table 8.1 shows the two tasks running on FreeRTOS, respectively *vLightLevel()* and *vTemperature().* The first task samples the value of the external light and display it on the screen. It is displayed using the ring symbol (enabling the internal spaces according to the light sampled) present inside the LCD. The second task dispays on the screen the temperature

Task Name	Description	Frequency(seconds)
vLightLevel	Sample Ambient Light	12
vTemperature	Sample internal Temperature	60

Table 8.1: Tasks running on FreeRTOS

of the internal sensor (using I^2C). The application makes use of the *efm32* library provided by Energy Micro. Moreover, it has been built combining different parts of the code present in some application notes provided by Energy Micro. The three following subsections explain with more details the functioning of tasks and the asynchronous event used in the benchmark.

8.1.1 vLightLevel

As shown in Table 8.1, this task occurs every 12 seconds. Figure 8.1 shows the steps that occur since the task is triggered and the value is displayed on the screen. In this example, it is visible how the GPMF module has been used either in the Task's body and the Handler of ADC. The function of the GPMF module are those functions with the red colour.
Initially, the task sets the sleep mode as EM1 because the task needs to execute the ADC (ADC works only with EM1). The value between Task and ADC are exchanged by means of queues provided by FreeRTOS API. Since the ADC takes some time to sample these values, the Task is being blocked on a semaphore until it is unblocked by ADC. When the ADC Handler occurs, it puts the sampled value inside the queue and it also unblocks the semaphore locked by the Task. After that, the ADC is turned off and the *vPMTicklessFromISR* (see section 7.4.9) checks if some other tasks with higher priority must be scheduled. The value is displayed on the screen when the core returns back to *vLightLevel()*. The task does not need anymore to stay in EM1, therefore it communicates that the vLightLevel can stay in EM2. After 12 seconds the task is executed again.

8.1.2 vTemperature

This task occurs every 60 seconds and it displayes on the screen the Temperature present on the DVK. Since this task does not use the ADC of EFM32 but an external device, it does not need to stay in EM1. Therefore, the task marks immediately its state as EM2.

8.1.3 Sending Accelerometer values

As is explaiend above, pressing the button SW1 is possible to send the values of Accelerometer via USART. The data are sent to USART only using the Handler of some devices, such as *ADC*, *DMA* and *Leuart*. Figure 8.2 shows the main steps performed by ISRs. When the button (SW1) is being pressed, the GPIO Handler is being executed and it sets the DMA to transfer the data from ADC to memory.

As long as the ADC is enabled, the system must run in EM1. When the ADC Handler occurs, it changes the sleep mode from EM1 to EM2 because the only devices enabled are DMA and LEUART (both can run in EM2). The DMA sets the channel to transfer the data from memory to Leuart. It clears the interrupt and unlock the EM2 when all data have been transmitted. Finally, when the leuart completes the transfer it returns back to FreeRTOS.

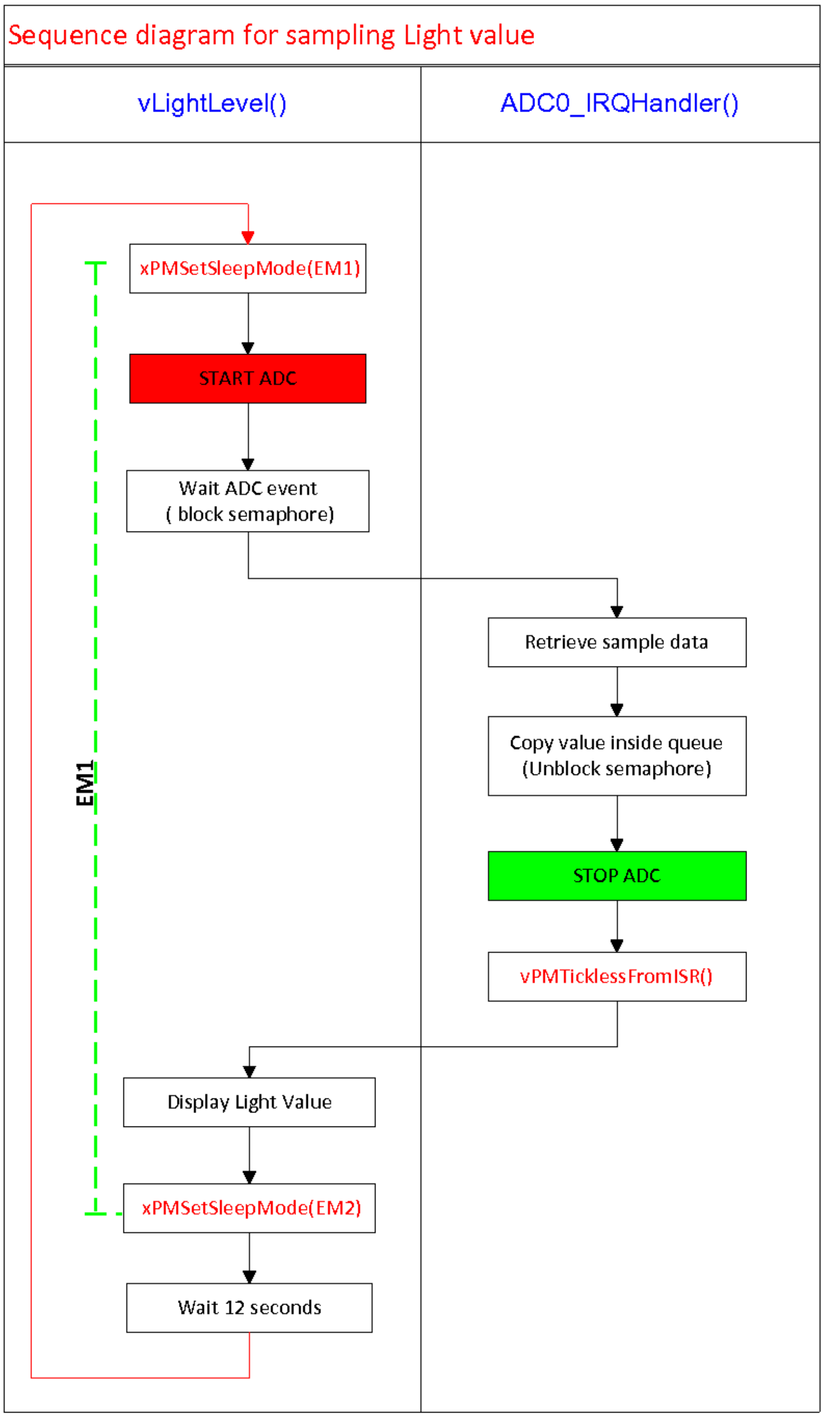


Figure 8.1: Sequence diagram of Task vLightLevel()

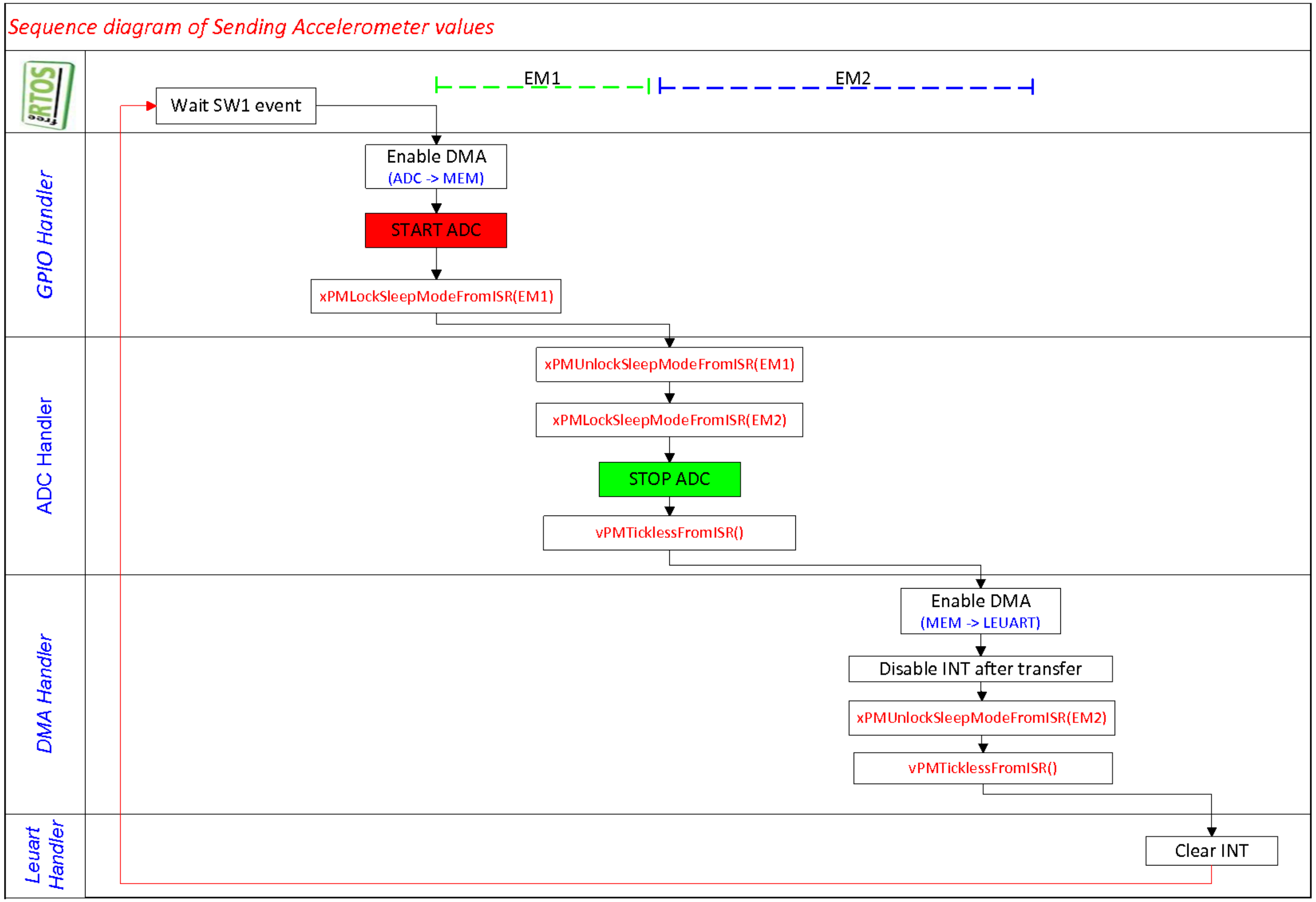

Sequence diagram of Sending Accelerometer values
EM1
EM2
GPIO Handler
ADC Handler
DMA Handler
Leuart Handler
Wait SW1 event
Enable DMA (ADC -> MEM)
START ADC
xPMLockSleepModeFromISR(EM1)
xPMUnlockSleepModeFromISR(EM1)
xPMLockSleepModeFromISR(EM2)
STOP ADC
vPMTicklessFromISR()
Enable DMA (MEM -> LEUART)
Disable INT after transfer
xPMUnlockSleepModeFromISR(EM2)
vPMTicklessFromISR()
Clear INT

Figure 8.2: Sequence diagram of Accelerometer

8.2 Advanced Energy Monitoring (AEM) and energyAwareProfiler

The DVK of EFM32 provides software and hardware tools that display the real-time energy consumption of the application running on the board. These two tools are called *Advanced Energy Monitoring* (AEM) and *energyAwareProfiler*.
The *AEM* built into DVK retrieves the real current values of the application running on the board and it displays these values on the LCD display located on the lower left side of the DVK. The dynamic range of the AEM varies from a minimum of 100nA to a maximum of 50mA. It measures either the current drawn by the MCU and the current drawn by other system components. The AEM uses a current sensor that samples the current flowing through the VMCU. It displays the data on the LCD and it also sends (voltage, current and timing) them through USB. It samples data with a rate of *60Hz* (16.6ms) when measuring currents below 200 µA and *120Hz* (8.3ms) when measuring currents above 200 µA. When AEM measures currents above 200 µA the maximum error is 0.1mA and below 200 µA the accuracy increases to 1 µA. The AEM detects changes in current consumption as small as 100nA.
The *energyAwareProfiler* gets data from the AEM on the kits via USB and displays these information in a current vs time representation. In order to represent the real-time current, the *energyAwareProfiler* combines the program counter sent from AEM and the correlated value found inside the object file running on the MCU. The current consumption is displayed by default on a *logarithmic* scale, but it can also be changed in a linear representation. The energyAwareProfiler allows to save the trace of data in CSV(comma separated values).

8.3 Power consumptions among different FreeRTOS versions

The following sections outline the energy consumption involved using the new versions of FreeRTOS. The simulations have been performed for the latest version of *FreeRTOS 7.0.1* and the new proposed solutions, *FreeRTOS RTC 7.0.1* and *FreeRTOS Tickless 7.0.1*. The latter has been simulated with and without prescaling. The results of simulations (running the benchmark described in previous section) have been processed and analysed using the CSV files provided by energyAwareProfiler. The tests have been performed for each version of FreeRTOS with different time resolutions. It is important to note that two versions of FreeRTOS (RTC and Tickless) use a different resolution (see section 6.2 and 6.3.3).
In order to fairly compare the energy consumption of the different versions of FreeRTOS, it has been used a resolution (7.8ms) common among all versions. The following simulations report both on y-axis and x-axis a linear representation. The power consumption in mW is reported on the y-axis and the time interval in ms on the x-axis. Finally, the benchmark has been run without sending the data of the accelerometer. The reason of this choice is due to the fact that in this way all

results have been compared in the same manner.

8.3.1 Original FreeRTOS 7.0.1

This is the latest FreeRTOS version available on the website. The first simulation has been performed using the benchmark without using any sleep modes. In practice, FreeRTOS is always running over the time. The tests Figure 8.3 a) have been performed varying FreeRTOS resolution with *1ms*, *7.8ms* and *100ms*.
The chart reports only the simulation of *No sleep* with resolution of 1ms. Simulations (no reported in the graph) with other resolutions (7.8ms and 100ms) consume less power because the system is interrupted less times than solution using 1 ms. The power dissipated by the MCU is steady to 18 mW and the average current is 5.59 mA.
The chart shows also the power consumption of the original version of FreeRTOS when it uses the sleep modes during idle tasks. For the same reasons explained in section 6.1 FreeRTOS can not use a sleep mode lower than EM1. Also for this case, the simulations have been performed for the three resolutions listed above (1ms, 7.8ms and 100ms). Results show that FreeRTOS consumes less energy using a resolution of 100ms and it dissipates more energy using a resolution of 1 ms. The power consumption decreases with increasing of the resolution. Therefore, a good compromise between resolution and power dissipation is 7.8ms. Using the resolution of 100ms and 7.8ms the average current oscillates around 1.93mA.

8.3.2 FreeRTOS RTC 7.0.1

This is the version of FreeRTOS proposed in chapter 6 with a RTC as tick timer. It also allows to use the EM2 when the core runs in idle state.
Figure 8.3 b) illustrates the simulations performed using the three different resolutions, such as *977* µ*s*, *7.8ms* and *62.5ms* (see sections 6.2 and 6.3.3). Also with FreeRTOS RTC is confirmed the trend that the power consumption decreases with increasing of the resolution. Using a resolutoin of 977 µs the average current is 343 µA and with a resolution of 7.8ms and 62.5ms the current is respectively 182 µA and 150 µA. As consequence the power dissipation decreases with increasing the resolution.

8.3.3 FreeRTOS Tickless 7.0.1 with prescaling

This is the version of FreeRTOS proposed in chapter 7. The tickless approach removes all ticks when the core is running in the idle task, therefore it needs to stay in a sleep state as long as the next task/routine occurs. The *prescaling* version enables the prescaler because it can keep the core in sleep state for a long time (the no prescaler version overflows more often).
The simulations for prescaler version have been performed using resolution of *977* µ*s*, *7.8ms* and *62.5ms* (see sections 6.2 and 6.3.3). One of the drawback of using the prescaler version is due to the energy overhead introduced by the RTC with larger resolution[5]. The Figure 8.4 a) shows that using the Tickless with a resolution of

977 µs does not affect the performance of the whole system. It provides an average current of 146 µA and a power dissipation of 500 µW. The results for FreeRTOS Tickless with resolution of 7.8ms are slightly higher than those discussed for 977 µs. On the contrary, increasing the resolution of *FreeRTOS Tickless* with 62.5ms it consumes more energy. Indeed, the prescaler in this case introduces a higher energy overhead with a double increase of the energy consumption. The power dissipation with prescaling is higher than 1 mW and the current oscillates around 315 µA. The Tickless strategy with prescaling must be avoided when the system uses a larger resolution.

8.3.4 FreeRTOS Tickless 7.0.1 without prescaling

This is the *FreeRTOS Tickless* solution without the use of prescaler. As is mentioned in previous section, if the system uses a no prescaling approach it awakes the core in idle state more often. On the other hand, this version of FreeRTOS is not affected by prescaler's overhead.
Figure 8.4 b) shows the results achieved of using a no prescaling strategy with the three common resolutions of *977* µ*s*, *7.8ms* and *62.5ms* (see sections 6.2 and 6.3.3). Without using prescaler, the system behaves more better for all resolutions because the overhead of presclaer is null. The best results are achieved using a resolution of 7.8 ms. All simulations have a current that oscillates between 129 µA and 137 µA. While the power varies between 431 µW and 460 µW.
It is important to note that this version of FreeRTOS consumes less energy with increasing of resolution (opposite than no prescaling).

8.3.5 Results Summary

Figure 8.5 outlines the results of the three different versions of FreeRTOS with a resolution of 7.8 ms. Only in this case, the chart uses on the y-axis a *logaritmic scale*. According to the simulations described in previous sections, the worst results are achieved using the *Original FreeRTOS 7.0.1* (blue line). Basically, the main drawback of Original version is due to the limitation of not using during idle state an energy mode lower than EM1. The results are more worse when the core does not use the sleeping state (brown line). Furthermore, the power dissipation remains steady for both versions.
The *FreeRTOS RTC 7.0.1* solution (green line) outperforms the *Original FreeRTOS* version and it also introduces few spikes during its execution. It also outperforms the *FreeRTOS Tickless prescaling solution* (violet line) because the latter introduces very large spikes during its execution. When the spikes of prescaling occur, they also exceed the power dissipated from the *Original FreeRTOS*. The *FreeRTOS Tickless no prescaling*(celestine line) solution represents the best choice among all FreeRTOSes. It also dissipates some spikes during its execution but they do not affect the final outcome.
Even though the results show that Tickless and RTC solution represent a good approach. This is not always true for all applications, because the benchmark used in these simulations introduces several idle states. Therefore, the choice of

the FreeRTOS solution depends from the type of the application to use. When the application uses larger idle task, the *FreeRTOS Tickless solution* is the winner strategy. Otherwise, using the *FreeRTOS RTC solution* is a good compromise when the type of application is unknown and the resolution of the system is not a limitation. Finally, the *FreeRTOS Tickless prescaling* solution it is convenient only with a low resolution.

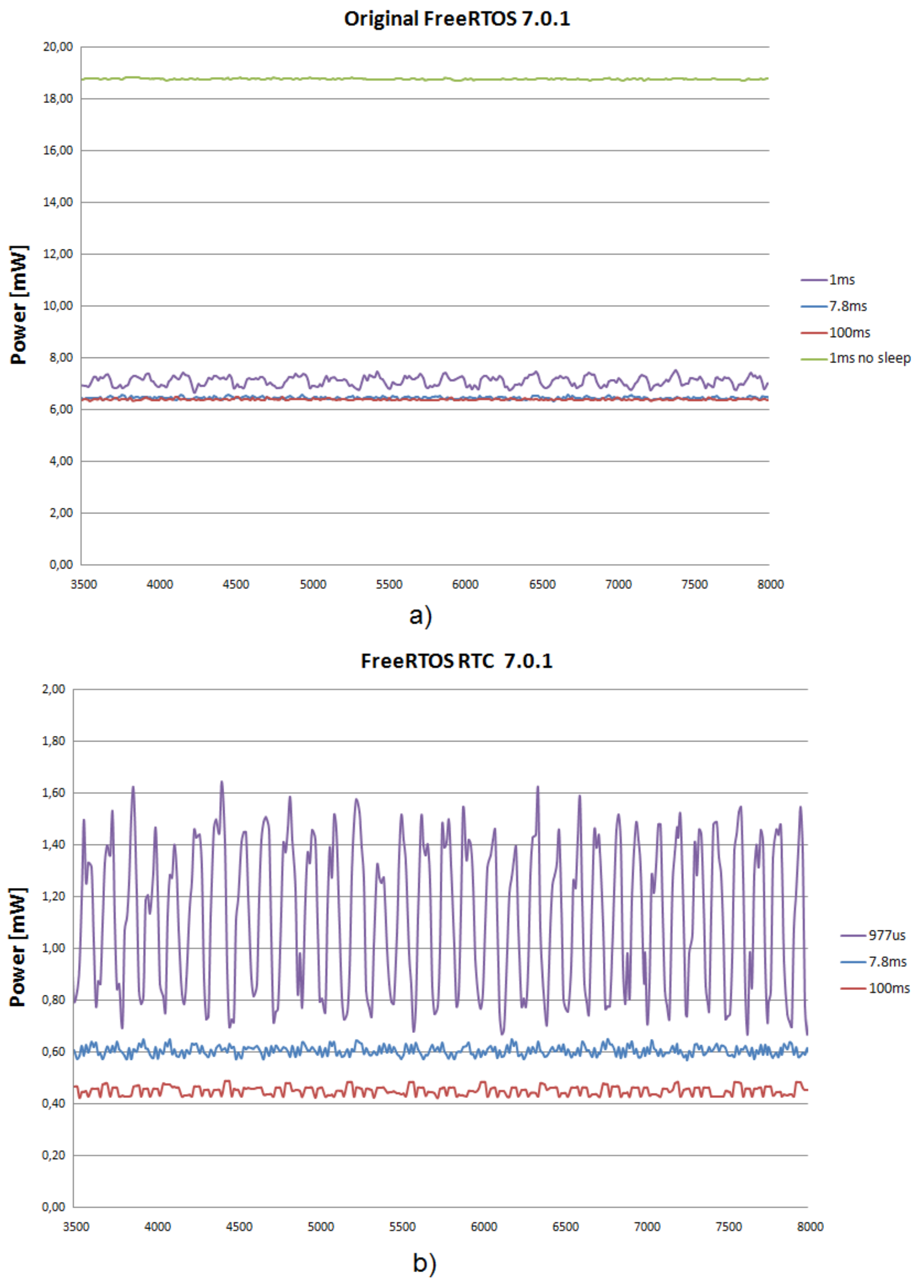
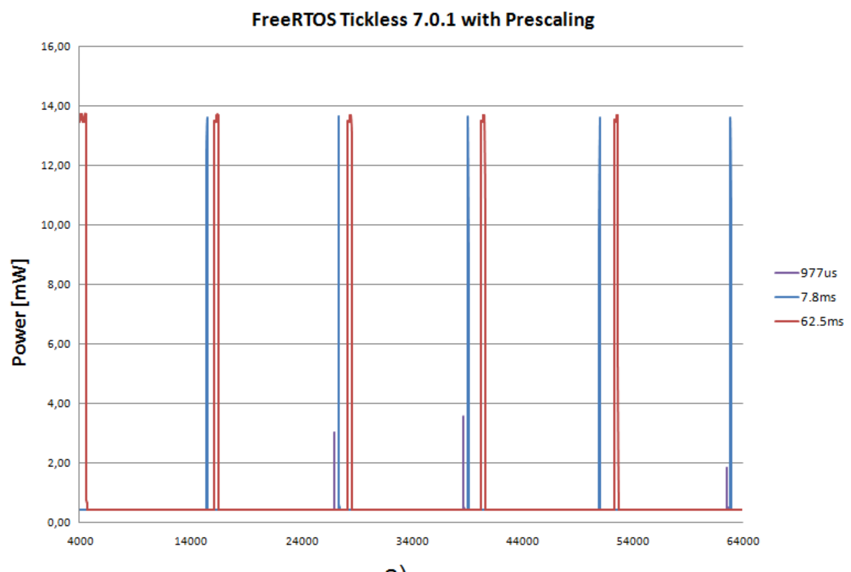
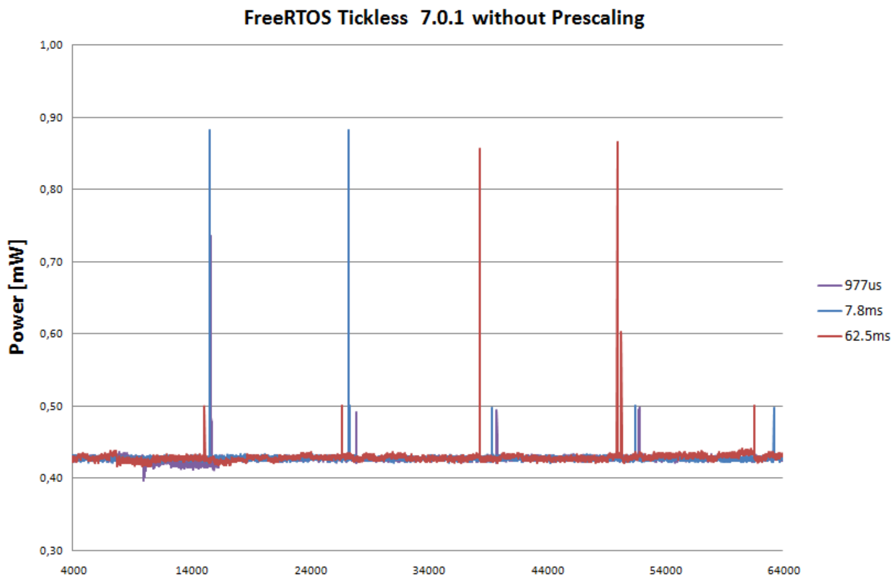


Figure 8.3: Power consumption with different versions of FreeRTOS. a) Original FreeRTOS 7.0.1 b) FreeRTOS RTC 7.0.1



a)



b)

Figure 8.4: Power consumption with different versions of FreeRTOS. a) FreeRTOS Tickless 7.0.1 with prescaling b) FreeRTOS Tickless 7.0.1 without prescaling

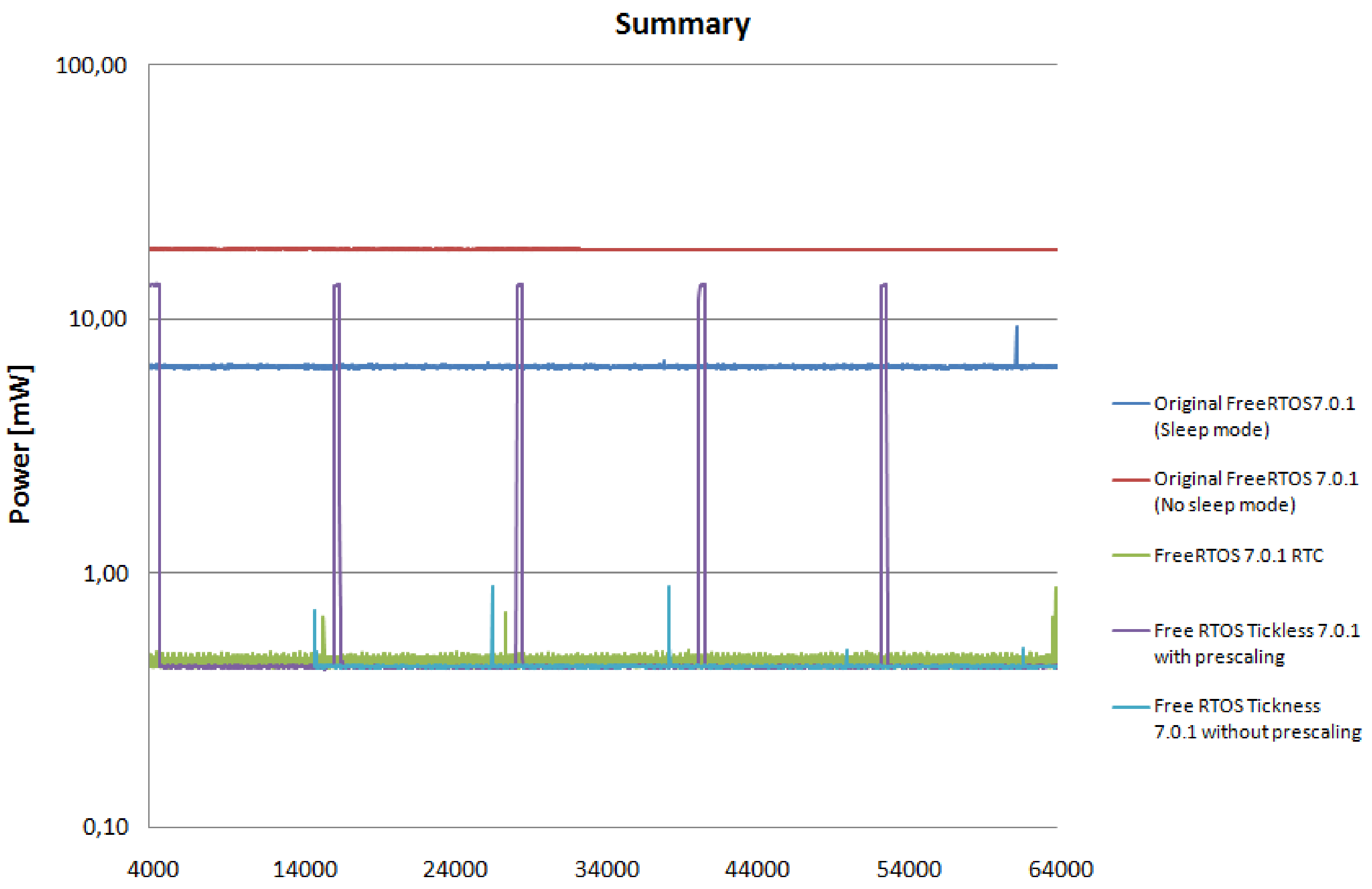


Figure 8.5: Power consumption of all FreeRTOSes

8.4 Benchmark's Screenshot

The energyAwareProfiler displays the behavior of the application in three different windows, such as the *code listing window*, *current graph* and *function listing*. The graph window shows in real time the current consumption of the running application.

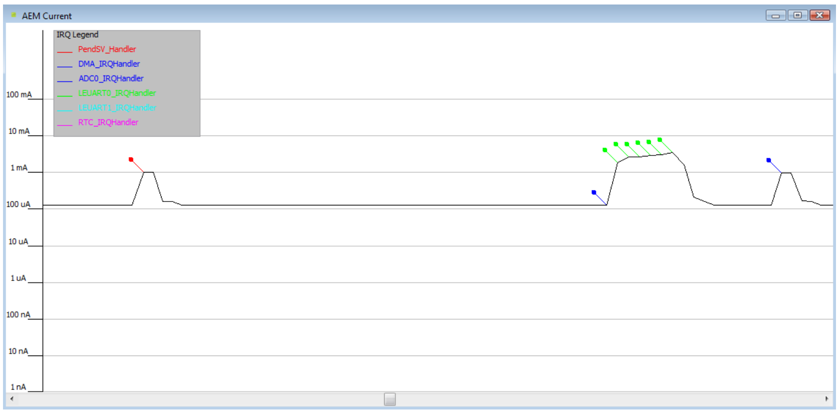


Figure 8.6: Time vs Current representation for FreeRTOS RTC 7.0.1 with resolution of 977 µs

The Figure 8.6 shows a screenshot of the benchmark when the MCU was running on *FreeRTOS RTC 7.0.1* with resolution of 977 µs. The first spike (*red* annotation) corresponds to the pendSVHandler, therefore when the RTC issues a context switching. The second spike is the instant when the core handles a request to send the accelerometer values via LEUART. The microcontroller handles the handler of DMA (*blue* annotaion), Leuart (*green* annotaion) and also ADC (*blue* annotation overlapped on the DMA). After that, the system manages the task to sample the temperature on the DVK or to sample the value of the light ambient sensor.
The energy consumed in idle state is slightly higher than 100 µs. According to the observation did from my supervisor at Energy Micro, the core should consume less energy when it runs in idle state. I think that the reason why the core runs in sleep mode with a current consumption above 100 µA is due to the memory *footprint* caused by FreeRTOS. Even though FreeRTOS provides a small footprint varying from 4-9 Kbytes, it however involves to consume more energy than a stand-alone application.
My supervisor is also investigating if the problem is caused because I have used during simulation a revision A of the MCU board[1] and an oldest version of the DVK board.

[1]Energy Micro released three different versions of the MCU board. The latest version is C, the oldest is A.

8.5 Battery lifetime

The Figure 8.7 illustrates the lifetime of the benchmark using a battery (e.g. ultra duracell alkaline battery) of 1500mAH 3V. The Figure 8.7 shows the number of days expected to run the application (ignoring battery self-discharge) for each version of FreeRTOS.
The lifetime of the *Original FreeRTOS* is around one month. Moreover, the lifetime of the system decreases (11 days) if it does not use any sleep modes. While the lifetime of the *FreeRTOS RTC* solution increases with increasing the resolution of the system. It can stay alive for at most 416 days.
Even though the lifetime of the *FreeRTOS Tickless prescaling* solution achieves good results (428 days). The lifetime of the system decreases with increasing of its resolution. However, the lifetime of each resolution is never worse than lifetime of the corresponding resolution in FreeRTOS RTC. The *FreeRTOS Tickless no prescaling* solution guarantees the longest lifetime and it behaves in the same manner also varying the resolution of the system. The best results are achieved using a resolution of 7.8ms. The new Tickless solution, in the best case saves energy for *44x* more than Original FreeRTOS. In the worst case it saves energy for more than *15x* (with any resolution).

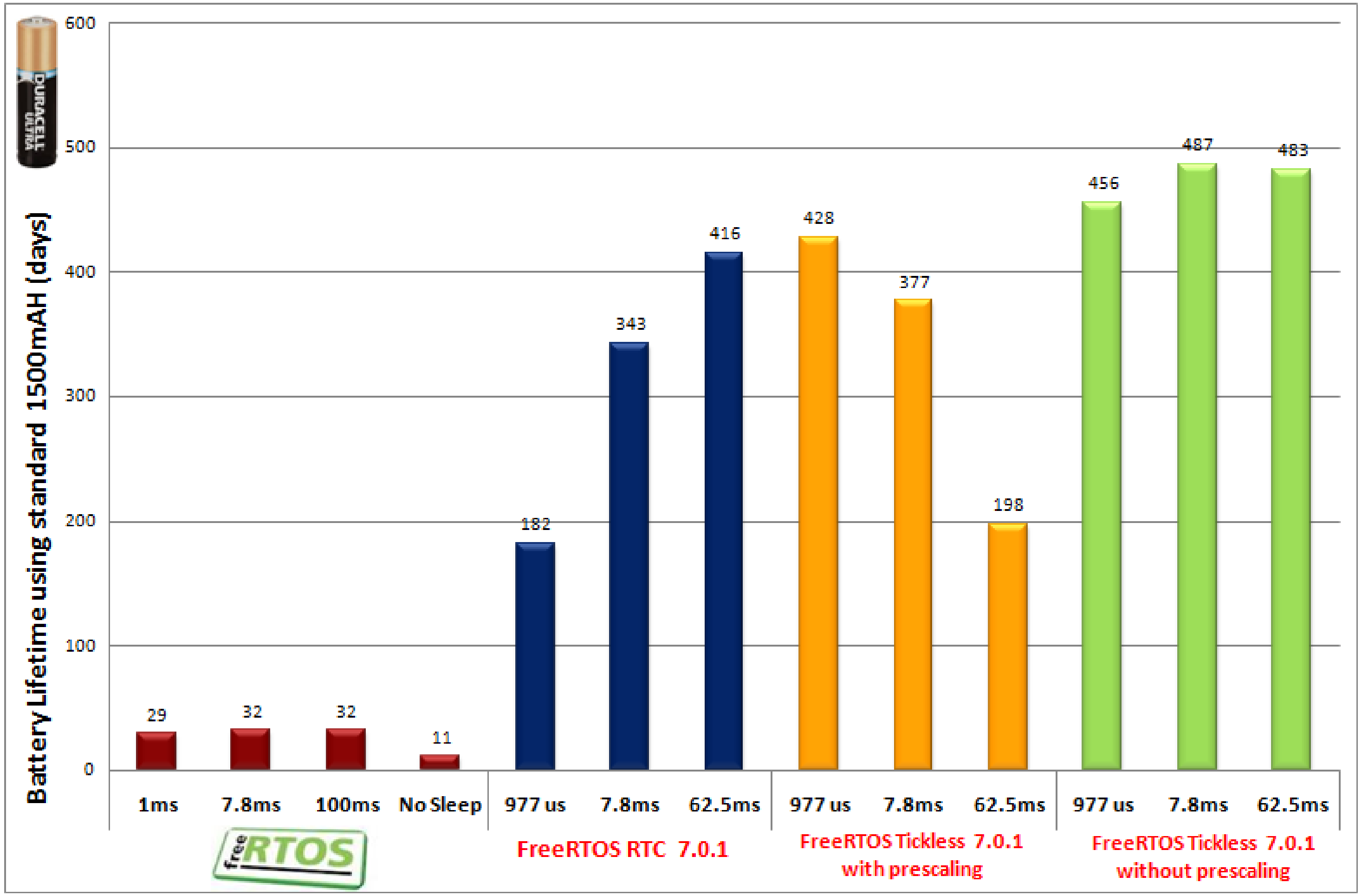


Figure 8.7: Expected battery lifetime using a common battery of 1500mAh

Part IV

Discussion

Chapter 9

Conclusions and Future work

"The biggest surprise for me was in the question, "Please select all of the operating systems you are considering using in the next 12 months." The number one choice was FreeRTOS. The reason I was so astounded was that FreeRTOS didn't even show up in the study last year. That's quite a gain-from not on the chart to the number one position!"

- **Richard Nass**, editorial director of Embedded Systems Design magazine and the Embedded Systems Conferences.

9.1 Conclusions

Recently, the market of low-cost and low-power 32-bit MCUs is growing rapidly. The grow of the new generation of low-cost 32-bit MCU integrates 16 to 256 kbytes of RAM and new advanced peripherals. As consequence, the new developed softwares reach a high level of complexity that can not be hide using the stand-alone applications. Therefore, the development of tailored operating systems allows users to achieve excellent benefits. They are built to save energy and achieve high performance in real time environments. *FreeRTOS* is a real-time kernel that is designed especially for embedded low-power MCUs, reaching a considerable popularity growth over its six year life [29].
The *EE Times Group* conducts every year through its readers, a survey regarding the environment they work in and the design process they employ. The research is carried out along different perspectives, such as *specific vendor choices*, *real time operating systems*, *cutting-edge processors* and so forth. Finally, the survey also aims to understand how the users use or plan to employ the analyzed factors in their projects. The survey outlines the limited use of commercial RTOSes inside the new projects. On the contrary, open source operating systems are on the rise, from 26% last year to 32% this year (from 21% in 2008). The biggest hit there is

against the commercial OSes, which fell from 47% to 38%[30]. The survey showed that readers plan to use FreeRTOS as operating system for upcoming projects. This trend is being confirmed looking the statistics of the number of times that FreeRTOS was download last years, *77,500* times.
The objective of this thesis is to propose and investigate the results of using a *Tickless Power Aware Framework* with the latest version of FreeRTOS 7.0.1. The thesis proposes *three* different approaches FreeRTOS RTC, FreeRTOS Tickless *with* prescaling and FreeRTOS Tickless *without* prescaling that can be used according to the needs of users. The *pros* and *cons* of each solution have been widely discussed in previous chapters.
The initial problem assignment of the thesis was to propose only an Energy Management Framework (called GPMF) combined with a RTC solution. The goal was to implement the Energy Framework above the Tickless solution proposed by Martin. I believed that was better to build a new Tickless Framework that was strictly coupled with the features provided by GPMF. Moreover, either the report and the code developed by Martin Tverdal allowed me to design a new Framework that was based on some issues that he partially faced in his thesis.
The FreeRTOS Tickless approach is a valuable alternative to employ only in those cases when the application introduces large and often idle tasks. Even though the new FreeRTOS solution might introduce some burden for systems that work in extreme temperature environment. It is possible to overcome these issues applying the techniques proposed in section 6.4. Therefore, it is worthwhile to apply these changes only when the trade-off between final cost and energy saved is well compensated.

The results showed that the *Tickless framework* solution *with* prescaling produces a higher power consumption than solution *without* prescaler. Thus, the results have clarified the doubt (see section 7.3.3) regarding which solution (prescaling and no prescaling) wastes more power. The prescaler produces an amount of energy that hurts the performance of the whole system.
Even though the Tickless Framework obtains excellent results without prescaler. In order to achieve these results the programmer must follow the limitation of using a new function in place of the standard FreeRTOS' API. This might be a further burden for programmers. The battery lifetime chart shows that FreeRTOS RTC achieves outstanding results with large resolution. Hence, my view is that this solution can achieve good results also when FreeRTOS has sporadic and few idle tasks. Finally, the results showed that in the best case the new Tickless Framework (without prescaling) saves energy for *44x* more than Original FreeRTOS. While in the worst case it saves energy for more than *15x* (with any resolutions). The battery (1500 mAh) lifetime has been increased from 11 days (No sleep) to 487 days (using the developed benchmark).

Although this thesis is a continuation of a previous work, I faced several challenges due to the lack of my initial background and the little support received from the FreeRTOS' maintainer. The latter has been the main issue to overcome whenever

it was necessary to undertake some technical decisions on the FreeRTOS' kernel. Therefore, the possible pitfalls of each solutions have been widely discussed with my supervisor at Energy Micro and the FreeRTOS' community (the official forum of FreeRTOS). I sent to the maintainer the new proposed versions of FreeRTOSes. It is hard to believe that the Tickless solution can be approved.
Moreover, for none of the supported ports in FreeRTOS it is not provides a module that manage the sleep modes. GPMF module has been designed to make a porting for any platforms. Thus, I think that this module can be used in FreeRTOS with little changes.

This master thesis could be considered as a valuable contribution to enhance the existing version of FreeRTOS. Combining this work with EFM32 allows to achieve excellent results in terms of the energy saving. Currently, some Energy Micro's customers are testing the new version of FreeRTOS in their applications.
Finally, I want to note that I spent some time to study and understand the different layers that compose the whole system. Using an approach bottom-up, the layers are divided as follow, *cortex-core*, *CMSIS library*, *EFM32 MCU board*, *DVK board*, *efm32library*, *FreeRTOS kernel* and *AEM.*

9.2 Future work

The previous chapters outline for every proposed strategies, the possible drawbacks of adopting them in the real environment. The possible improvement to apply in each solution are strictly dependent of two factors, *Hardware modification* and *costs*.
The latter is the main limitation that affects the final user. In fact, the proposed techniques might also affect the final cost of the system. It is important to find a good compromise between cost and final performances.
The *Hardware solution* requires to change the original hardware of EFM32. At this level there are several possible techniques that can be employed to enhance the existing hardware. One of the main limitation of the RTC counter is the number of bits reserved for its counter (24 bits). Using a counter with more bits, e.g. 32bits, it would allow to keep the core in the idle state for longer time. This would allow to drop definitely the Tickless no prescaling solution and use only the prescaling one. Furthermore, another important HW modification would be to provide a support of TXCO oscillator with EFM32. This would allow to keep the core running also in extreme temperature environment. If this HW approach is too expensive, an alternative solution would be to use a software approach with a *look up table* that stores for each temperature the corresponding drifting.
One of the main problem of Tickless solution is due to the *marginal error* that can occur when the time is being retrieved by the RTC. I have warned this problem to my supervisor at Energy Micro and he is currently investigating the reason why the RTC behaves in this manner. Therefore, it is necessary to develop a solution that fixes this problem.
However, some other possible improvements to perform in the code are strictly cou-

pled to the feedback of FreeRTOS' maintainer. It is not clear, if the modules of the *co-routine* and *timers* can have advantages of using a Power Manager Framework. This would allow to use a more robust and reliable Power Manager Framework. I think that before to implement these new feature, it is necessary to wait an answer from maintainer.

Part V

Appendix

Appendix A

PPMF for EFM32

In this Appendix are shown only files regarding GPMF (*PowerManager.h* and *PowerManager.c*). In order to use the GPFM module with FreeRTOS it is also necessary to use the new version of files *task.c*, *task.h* and *FreeRTOSConfig.h* (These are files used in FreeRTOS). Those files are not listed in this Appenidix because the changes have already discussed in previous section.

Here is shown the code in *PowerManager.h*.

```
#ifndef PM_H

#include "FreeRTOS.h"
#include "projdefs.h"

#if ( configUSE_POWERMANAGER == 1 )

  /*Number of possible sleep modes*/
  #define configSLEEPMODES              5

  /* This value is the default sleep mode used by power manager in idle task. If
   * the value is 0, then the Power Manager uses as default mode the running state.
   * Usually, this value must be set up to the next sleep mode after running state.
   */
  #define configDEFAULT_SLEEPMODE    1

  /* PAY ATTENTION!!!
  The order of sleep modes must be ascending and the running mode must always be 0↩
      */
  #define RUNNING      0
  #define EM1        1
  #define EM2        2
  #define EM3        3
  #define EM4        4

  /* ----------- Functions used by users to deal with sleep modes  -------------*/
  void vPMInit ( void );
  portBASE_TYPE xPMSetSleepMode ( unsigned portCHAR xMode);
  portBASE_TYPE xPMSetSleepModeFromISR ( unsigned portCHAR xMode );
  unsigned portCHAR ucPMGetSleepMode ( void );
  unsigned portCHAR ucPMGetSleepModeFromISR (void);
  portBASE_TYPE xPMLockSleepModeFromISR ( unsigned portCHAR xMode);
  portBASE_TYPE xPMUnlockSleepModeFromISR (unsigned portCHAR xMode);
  portBASE_TYPE xPMUnlockSleepMode (unsigned portCHAR xMode);

  /* Internal function used inside by Power Manager to deal with sleep modes and ↩
      Tickless */
  void vPMIdleSleepMode ( void ) PRIVILEGED_FUNCTION;

  /* Functions used internally by FreeRTOS to allow a communication between GPMF ↩
      and Tasks */
  void vPMUpdateSleepModesInc ( unsigned portCHAR xMode );
  portBASE_TYPE xPMUpdateSleepModesDec ( unsigned portCHAR xMode );

#endif

#endif
```

Listing A.1: PowerManager.h

Here is shown the code in *PowerManager.c*.

```
#include "PowerManager.h"
#include "efm32.h"

#if ( configUSE_POWERMANAGER == 1 )

  /* Macro definitions */
  #define xPMCheckMode(mode) ((mode >= configSLEEPMODES ) || ( mode < 0) ? ↩
      errINSERT_NOT_VALID_SLEEPMODE : pdTRUE)

  /* This array keeps track about how many tasks want to stay in each sleep mode*/
  PRIVILEGED_DATA static unsigned portCHAR ucSleepTasks [configSLEEPMODES] = {0};

  /* This array keeps track about how many "locked ISRs" want to stay in each sleep↩
       mode*/
  PRIVILEGED_DATA static volatile unsigned portCHAR ucSleepLockISR [↩
      configSLEEPMODES] = {0};

  /* This variable save the highest sleep mode requested by ISRs */
  PRIVILEGED_DATA static volatile unsigned portCHAR ucSleepISR;

  /* Flag is 1 if an ISR done a request to move in a sleep mode. Otherwise flag is ↩
      0 */
  PRIVILEGED_DATA static volatile unsigned portCHAR ucFlagISR;

 /* This value is used to keep update the CMU status. Since when MCU goes in sleep
  * mode the value inside CMU->STATUS can suffer a misleading value. Then is a
  * responsability of PM to keep update this value.
  * This variable is only used for Energy Micro
  */
  PRIVILEGED_DATA static volatile portBASE_TYPE xPMStatus;

  /* ------ Internal functions used by Power Manager for porting EFM32 in FreeRTOS ↩
      ------ */
  static portBASE_TYPE xPMGoToSleepMode ( unsigned portCHAR xMode ) ↩
      PRIVILEGED_FUNCTION;
    static void vPMGoToSleepModeEM1( void ) PRIVILEGED_FUNCTION;
  static void vPMGoToSleepModeEM2( void ) PRIVILEGED_FUNCTION;
  static void vPMGoToSleepModeEM3( void ) PRIVILEGED_FUNCTION;
  static void vPMGoToSleepModeEM4( void ) PRIVILEGED_FUNCTION;
  static void vPMRestoreSleepMode( void ) PRIVILEGED_FUNCTION;

/******************************************************************************
 * @brief
 *  Initialise Power Management Framework.
 *
 * @details
    This function initialise internal registers used to manage efficiently GPMF.
 * @note
```

```
     This function must be invoked before the scheduler starts. For EFM32 it is
   necessary to use it.
 ******************************************************************************/

  void vPMInit ( void )
  {
      /*Lock EMU function*/
      EMU->LOCK = EMU_LOCK_LOCKKEY_LOCK;

  }
/******************************************************************************
 * @brief
 *  Set a sleep mode in tasks' body.
 *
 * @details
     This functions provides to set a sleep mode during task execution. If the
     sleep mode passed as input it is not a valid value, then the function returns
     an error value errINSERT_NOT_VALID_SLEEPMODE.
   Otherwise it returns a valid value pdTRUE.
 *
 * @note
     This function can only be invoked in tasks body and not inside ISRs.

 * @par
     Get in input the value of sleep mode to go in.
 ******************************************************************************/
  portBASE_TYPE xPMSetSleepMode ( unsigned portCHAR xMode)
  {
  portBASE_TYPE xReturn;

  portENTER_CRITICAL();
  /*Check if the value given in input is valid*/
  if ( xPMCheckMode( xMode ) == pdTRUE){
     ucSleepTasks[ucTaskGetCurrentTCB()]--;
           vTaskPutCurrentTCB(xMode);
           ucSleepTasks[xMode]++;
     xReturn = pdTRUE;
  }else{
           xReturn = errINSERT_NOT_VALID_SLEEPMODE;
  }
  portEXIT_CRITICAL();

  return xReturn;
  }

/*****************************************************************************
 * @brief
 *   Set a sleep mode in ISRs' body.
 *
 * @details
    This functions provides to set a sleep mode during ISR execution. If the
    sleep mode passed as input it is a not valid value, then the function returns
    an error value errINSERT_NOT_VALID_SLEEPMODE.
  Otherwise it returns a valid value pdTRUE.
 *
 * @note
    The sleep mode passed as input is valid only until the system meets the first
    idle task. For the following idle tasks the sleep mode is being discarded.
    This function can only be invoked in ISR's body.
 * @par
    Get as input the value of sleep mode to go in
 ******************************************************************************/
  portBASE_TYPE xPMSetSleepModeFromISR ( unsigned portCHAR xMode )
  {
  portBASE_TYPE xReturn;

  /*Check if the value given in input is valid*/
  if ( xPMCheckMode( xMode ) == pdTRUE){
```

```
            ucSleepISR = xMode;
      ucFlagISR = 1;
      xReturn = pdTRUE;
   }else
            xReturn = errINSERT_NOT_VALID_SLEEPMODE;

   return xReturn;
   }

/***************************************************************************
 * @brief
 *  Get the current sleep mode where the MCU can be moved in.
 *
 * @details
    According to requests done with functions like xPMSetSleepModeFromISR or
    xPMSetSleepMode the function aims to know in which sleep mode the MCU can be
    moved in. The function returns the value of current sleep mode.
 *
 * @note
    This function can be invoked only in tasks' body.

 ****************************************************************************/

   unsigned portCHAR ucPMGetSleepMode ( void )
   {
     unsigned portCHAR i, ucReturn = configDEFAULT_SLEEPMODE;

     portENTER_CRITICAL();

    /*Check which is the suitable mode between tasks and LockISRs*/
     for (i=0; i<configSLEEPMODES; i++)
   if ( (ucSleepTasks[i] > 0) || (ucSleepLockISR[i] > 0)){
      ucReturn = i;
      break;
         }

    /*Check which is the suitable mode among simple ISR, tasks and LockISR */
   if ( ( ucFlagISR) && ( ucReturn > ucSleepISR))
      ucReturn = ucSleepISR;

     portEXIT_CRITICAL();

     return ucReturn;
   }

/***************************************************************************
 * @brief
 *  Get the current sleep mode where the MCU can be moved in (FromISR).
 *
 * @details
    According to requests done with functions like xPMSetSleepModeFromISR or
    xPMSetSleepMode the function aims to know in which sleep mode the MCU can be
    moved in. The function returns the value of current sleep mode.
 *
 * @note
    This function can be invoked only in ISRs' body.
 ****************************************************************************/

   unsigned portCHAR ucPMGetSleepModeFromISR (void)
   {
     unsigned portCHAR i, ucReturn = configDEFAULT_SLEEPMODE;

      /*Check which is the suitable mode between tasks and LockISRs*/
     for (i=0; i<configSLEEPMODES; i++)
       if ( (ucSleepTasks[i] > 0) || (ucSleepLockISR[i] > 0)){
         ucReturn = i;
         break;
     }
    /*Check which is the suitable mode among simple ISR, tasks and LockISR */
    if ( ( ucFlagISR) && ( ucReturn > ucSleepISR))
        ucReturn = ucSleepISR;

    return ucReturn;
  }

/*************************************************************************
 * @brief
 *  This function locks a sleep mode inside an ISR.
 *
 * @details
    The sleep mode is locked if there is a need to keep the value of sleep mode
    for a while. The way to unlock this value is to use the following functions
    xPMUnlockSleepModeFromISR or xPMUnlockSleepMode.
 *
 * @note
    This function can be invoked only in ISRs' body. Besides, this function
    can be used as an alterantive solution to xPMSetSleepModeFromISR.
   Since the use of this function can involve in a misleading use of
    sleep modes, then the function must be used only when it is necessary.
    The function can lock more sleep modes and also nested ones.
 * @par
    Get in input the value of sleep mode to lock.
 *************************************************************************/

  portBASE_TYPE xPMLockSleepModeFromISR ( unsigned portCHAR xMode)
  {
    /*Check if the value given in input is a valid one*/
    if ( xPMCheckMode( xMode ) == pdTRUE){
       ucSleepLockISR[xMode]++;
    return pdTRUE;
       }else
    return errINSERT_NOT_VALID_SLEEPMODE;
  }

  /*************************************************************************
 * @brief
 *  This function unlocks a sleep mode previously locked with ↩
     xPMLockSleepModeFromISR.
 *
 * @details
    This function unlocks a sleep mode only inside inside an ISR's body. If the ↩
         function
    tries to unlock a sleep mode not locked previously, then the function
    discards the request and returns an error code. Otherwise it returns a
    valid number pdTRUE.
 *
 * @note
    Function to call only inside ISR.
 * @par
    Get in input the value of sleep mode to unlock.
 *************************************************************************/

  portBASE_TYPE xPMUnlockSleepModeFromISR (unsigned portCHAR xMode)
  {
    /*Check if the value given in input is a valid one*/
     if ( xPMCheckMode( xMode ) == pdTRUE){
    /*If a request is issued and its value is equal zero, it means that there is no↩
         ISR to unlock*/
   if (ucSleepLockISR[xMode] == 0)
      return errCOULD_NOT_UNLOCK_SLEEPMODE;
   else{
            ucSleepLockISR[xMode]--;
            return pdTRUE;
   }
     }else
       return errINSERT_NOT_VALID_SLEEPMODE;
```

```c
  }

/************************************************************************
 * @brief
 *  This function unlocks a sleep mode previously locked with ↩
     xPMLockSleepModeFromISR
 *
 * @details
     This function unlocks a sleep mode only inside an task’s body. If the function
     tries to unlock a sleep mode not locked previously, then the function
     discards the request and returns an error code. Otherwise it returns a
     valid number pdTRUE.
 *
 * @note
     Function to call only inside task.
 * @par
     Get in input the value of sleep mode to unlock.
 ************************************************************************/

  portBASE_TYPE xPMUnlockSleepMode (unsigned portCHAR xMode)
  {
    portBASE_TYPE xReturn;

       portENTER_CRITICAL();

       /*Check if the value given in input is a valid one*/
       if ( xPMCheckMode( xMode ) == pdTRUE){
          /*If a request is issued and its value is equal zero, it means that there↩
                is no ISR to unlock*/
          if (ucSleepLockISR[xMode] == 0)
       xReturn = errCOULD_NOT_UNLOCK_SLEEPMODE;
    else{
       ucSleepLockISR[xMode]--;
       xReturn= pdTRUE;
    }
       }else
     xReturn = errINSERT_NOT_VALID_SLEEPMODE;

       portEXIT_CRITICAL();

    return xReturn;
  }

/************************************************************************
 * @brief
 *  This function is used in IDLE TASK and can be used:
  POWER MANAGER: It only pushes the MCU in sleep modes;
 *
 * @details
    This function is directly called inside the IDLE TASK. The user DOESN NOT
    have to call it in vApplicationIdleHook.
 *
 * @note
    If the user does not use the vPMInit, then ucPMIdleSleepMode does not
    work properly.
 ************************************************************************/

 portBASE_TYPE ucPMIdleSleepMode ( void )
  {
    unsigned portCHAR sleep;
    portBASE_TYPE xReturn;

      vTaskSuspendAll();
      portENTER_CRITICAL();

      /* Retrieve the sleep mode to use for idle task */
      sleep = ucPMGetSleepMode();
      xPMGoToSleepMode(sleep); /* Go in sleep mode */
      /* Reset ucFlagISR if SSM issues a request before idle task */
      ucFlagISR = 0;

      portEXIT_CRITICAL();
      xTaskResumeAll();

    return xReturn;
  }

/************************************************************************
 * @brief
 * This function is a porting for Energy Micro. Besides, it provides to move the
   MCU in the sleep mode received as input.
 *
 * @details
      According to the value received as input, the function enables the MCU in
      the right sleep mode. If the sleep mode got as input is not valid, then
      the function returns an error value (-1).
 *
 * @par
     Get in input the value of sleep mode.
 ************************************************************************/

 static portBASE_TYPE xPMGoToSleepMode ( unsigned portCHAR xMode )
 {
    switch(xMode)
  {
          case RUNNING:
            break;
          case EM1:
            vPMGoToSleepModeEM1();
            break;
          case EM2:
            vPMGoToSleepModeEM2();
            break;
          case EM3:
            vPMGoToSleepModeEM3();
            break;
          case EM4:
            vPMGoToSleepModeEM4();
            break;
          default:
            return errCOULD_NOT_GOTO_SLEEPMODE;
  }
 return pdTRUE;
 }

/************************************************************************
 * @brief
 *   This function is a porting for Energy Micro. Move the MCU in EM1
 *
 * @details
      The function before to go in EM1 it sets the CTRL register and it
      unlocks the EMU register locked in vPMInit.
 *
 * @note
      The MCU is awakened with an event (WFE)
 ************************************************************************/

static void vPMGoToSleepModeEM1( void )
{
  /*Unlock EMU registers*/
  EMU->LOCK = EMU_LOCK_LOCKKEY_UNLOCK;

  /*It is used to gurantee that the user can not change EMs in the rest of program↩
       */
  EMU->CTRL |= EMU_CTRL_EM2BLOCK;

  /*If EMU register was locked, then lock it again */
  EMU->LOCK = EMU_LOCK_LOCKKEY_LOCK;


  /* Enter Cortex-M3 sleep mode */
  SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk;
  __WFE();


}

/**************************************************************************
 * @brief
 *   This function is a porting for Energy Micro. Move the MCU in EM2
 *
 * @details
      The function before to go in EM2 it REMOVES the CTRL register and it
      unlocks the EMU register locked in vPMInit. Besides, it keeps updated the
      value of CMU->STATUS (explained above). When the MCU is woken up it enables
    all oscillators that were enabled before to go in a sleep mode.
 *
 * @note
      The MCU is awakened with an event (WFE)
 **************************************************************************/

static void vPMGoToSleepModeEM2( void )
{

    /*Save the content of STATUS before to move in EM2. This is useful because the ↩
        HW can alter
      the value of STATUS register in the meanwhile when the MCU is in EM2 */
    xPMStatus = (CMU->STATUS);

    /* Unlock EMU registers */
    EMU->LOCK = EMU_LOCK_LOCKKEY_UNLOCK;

    /*Remove the flag to block EM2 and lower (if any)*/
    EMU->CTRL &= ~EMU_CTRL_EM2BLOCK;

    /*If EMU register was locked, then lock it again */
    EMU->LOCK = EMU_LOCK_LOCKKEY_LOCK;

    /* Enter Cortex-M3 deep sleep mode */
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;
    __WFE();

}

/**************************************************************************
 * @brief
 *   This function is a porting for Energy Micro. Move the MCU in EM3
 *
 * @details
      The function before to go in EM1 it REMOVES the CTRL register and it also
      unlocks the EMU register locked in vPMInit. Besides, it keeps updated the
      value of CMU->STATUS (explained above). When the MCU is woken up it enables
    all oscillators that were enabled before to go in a sleep mode.
 *
 * @note
      The MCU is awakened with an event (WFE). The only difference
      with EM2 is that the function takes care about CMU->LOCK register, if this
      register was enabled before sleep, then it sets the register again with ↩
          LOCKED.
 **************************************************************************/

static void vPMGoToSleepModeEM3( void )
{
  portBASE_TYPE xPMStatusLockCMU;
```

```

    /* Save the content of STATUS before to move in EM2. This is useful because the↩
         HW can alter
    the value of STATUS register in the meanwhile when the MCU is in EM2 */
    xPMStatus = (CMU->STATUS);

    /* Unlock EMU registers */
    EMU->LOCK = EMU_LOCK_LOCKKEY_UNLOCK;

    /*Remove the flag to block EM2 and lower (if any)*/
    EMU->CTRL &= ~EMU_CTRL_EM2BLOCK;

    /*If EMU register was locked, then lock it again */
    EMU->LOCK = EMU_LOCK_LOCKKEY_LOCK;

    /* CMU registers may be locked */
    xPMStatusLockCMU = CMU->LOCK & CMU_LOCK_LOCKKEY_LOCKED;
    CMU->LOCK = CMU_LOCK_LOCKKEY_UNLOCK;

    /* Disable LF oscillators */
    CMU->OSCENCMD = CMU_OSCENCMD_LFXODIS | CMU_OSCENCMD_LFRCODIS;

    /*If CMU register was locked, then lock it again */
    if ( xPMStatusLockCMU )
      CMU->LOCK = CMU_LOCK_LOCKKEY_LOCK;

    /* Enter Cortex-M3 deep sleep mode */
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;
    __WFE();

    vPMRestoreSleepMode();

}

/*****************************************************************************
 * @brief
 *   This function is a porting for Energy Micro. Move the MCU in EM4
 *
 * @details
      The function moves the MCU directly in EM4. Only a reset can awake the MCU.
 *
 * @note

 ******************************************************************************/
static void vPMGoToSleepModeEM4( void )
{
  unsigned portCHAR i;

   /* Unlock EMU registers */
   EMU->LOCK = EMU_LOCK_LOCKKEY_UNLOCK;

   for (i = 0; i < 4; i++)
   {
     EMU->CTRL = (2 << _EMU_CTRL_EM4CTRL_SHIFT);
     EMU->CTRL = (3 << _EMU_CTRL_EM4CTRL_SHIFT);
   }
   EMU->CTRL = (2 << _EMU_CTRL_EM4CTRL_SHIFT);
}
/*****************************************************************************
 * @brief
 *   This function is used to restore the oscillators after EM2 or EM3
 *
 * @details
      The functions restore those oscillators were enabled before to move the
      MCU in sleep mode. This operation is accomplished thanks to xPMStatus that
      keeps track which oscialltors were enabled before to go in sleep mode.
 *
 * @note
```

```
      The HFRCO core is automatically restored by the system.
 ******************************************************************************/
static void vPMRestoreSleepMode( void )
{
  portBASE_TYPE xPMStatusLockCMU;

  /* CMU registers may be locked */
  xPMStatusLockCMU = CMU->LOCK & CMU_LOCK_LOCKKEY_LOCKED;
  CMU->LOCK = CMU_LOCK_LOCKKEY_UNLOCK;

  /* Restore oscillators used before to drop in EM2/EM3 */
  CMU->OSCENCMD = xPMStatus & (CMU_STATUS_AUXHFRCOENS | CMU_STATUS_HFXOENS | ↩
       CMU_STATUS_LFRCOENS | CMU_STATUS_LFXOENS);

  /* If the oscillator used for clocking core is not HFRCO, then I restore the ↩
       right one */
  switch (xPMStatus & (CMU_STATUS_HFXOSEL | CMU_STATUS_HFRCOSEL | ↩
       CMU_STATUS_LFXOSEL | CMU_STATUS_LFRCOSEL))
  {
      case CMU_STATUS_LFRCOSEL:
        /* Wait for LFRCO to stabilize */
        while (!(CMU->STATUS & CMU_STATUS_LFRCORDY)) ;
        CMU->CMD = CMU_CMD_HFCLKSEL_LFRCO;
        break;

      case CMU_STATUS_LFXOSEL:
        /* Wait for LFXO to stabilize */
        while (!(CMU->STATUS & CMU_STATUS_LFXORDY)) ;
        CMU->CMD = CMU_CMD_HFCLKSEL_LFXO;
        break;

      case CMU_STATUS_HFXOSEL:
        /* Wait for HFXO to stabilize */
        while (!(CMU->STATUS & CMU_STATUS_HFXORDY)) ;
        CMU->CMD = CMU_CMD_HFCLKSEL_HFXO;
        break;

      default: /* CMU_STATUS_HFRCOSEL */
        /* If core clock was HFRCO core clock, it is automatically restored by ↩
            system */
        break;
  }

  /* Restore CMU register locking */
  if ( xPMStatusLockCMU )
    CMU->LOCK = CMU_LOCK_LOCKKEY_LOCK;

}

/*****************************************************************************
 * @brief
 *   This function is internally used by Power Manager and can not be used in
     application level.
 *
 * @details
     This function updates the number of tasks that stay in the sleep mode
     given as input. It increments the number of tasks in ucSleepTasks array.
 *
 ******************************************************************************/
void vPMUpdateSleepModesInc (unsigned portCHAR xMode )
{
   ucSleepTasks[xMode]++;
}

/*****************************************************************************
 * @brief
 *   This function is internally used by Power Manager and can not be used in
     application level.
   @ret
     Return pdTRUE if the referred sleep mode has been decremented succesfully
     Return pdFAIL if the referred sleep mode did not contain any task
 * @details
     This function updates the number of tasks that stay that sleep mode
     given as input. It decrements the number of tasks in ucSleepTasks array.
 *
 ******************************************************************************/
portBASE_TYPE xPMUpdateSleepModesDec ( unsigned portCHAR xMode )
{
  if (ucSleepTasks[xMode] == 0)
      return pdFAIL;
  else{
      ucSleepTasks[xMode]--;
      return pdTRUE;
  }
}
#endif
```

Listing A.2: PowerManager.c

Appendix B

Tickless Framework

Here are reported those functions discussed in chapter 7.

B.1 vPMIdleSleepMode()

This function is implemented inside `PowerManager.c`.

```c
void vPMIdleSleepMode ( void )
{
    unsigned portCHAR sleep;
    portTickType TickNextTask;
    portTickType TickNextRoutine = portMAX_DELAY;

  #if ((configTICKLESS == 1)  && (configUSE_POWERMANAGER == 1))

      if ( xTickPass == 0){

          /* Make sure the eventregister is set to 0. Otherwiswe core keep running↩
              */
          __SEV();
          __WFE();

          /* Suspend either tasks and Interrupts */
          vTaskSuspendAll();
          portENTER_CRITICAL();

          /* Tickless mode is starting*/
          xPMTicklessENB = TICKRUNNING;
          CounterBefore = 0;
          Counter = 0;

          /* Retrieve the sleep mode to use in idle state */
          sleep = ucPMGetSleepMode();

          /* RTC does not work beyond EM2. So, I have to force EM2 as max sleep ↩
              mode */
          if (sleep > EM2)
            sleep = EM2;

          TickNextTask = xTaskNextTick();
          #if (configUSE_CO_ROUTINES == 1)
              TickNextRoutine = xCoRoutineNextTick();
          #endif

          /* Break even checks if it is worthwhile to use tickless mode */
          if ( (xPMBreakEvenTime(TickNextTask,TickNextRoutine) == pdPASS) ){

        #if (configUSE_CO_ROUTINES == 1)
          if ( TickNextTask <= TickNextRoutine )
             TickToWakeup = (TickNextTask - xTaskGetTickCountTickless()) - 1;
          else
             TickToWakeup = (TickNextRoutine - xTaskGetTickCountTickless()) - 1;
        #else
          TickToWakeup = (TickNextTask - xTaskGetTickCountTickless()) - 1;
        #endif

        #if (configPRESC == 0)
          /* No prescaling */
          TickToWakeup = TickToWakeup * (RTC_FREQ/configTICK_RATE_HZ);
        #endif

                if ( TickToWakeup > MAXRTCOUNT )
                        TickToWakeup = MAXRTCOUNT;

        vPMSetRTC(TickToWakeup);
        xPMGoToSleepMode(sleep);  /* Go in sleep mode */
        /* Tickless starts right now */
        xPMTicklessENB = TICKLESS;
          }else{
        /* No tickless mode, I stay in sleep mode until next tick occurs */
        vPMSetRTC(((RTC_FREQ/PRESC)/configTICK_RATE_HZ));
        RTC_IntClear(RTC_IFC_COMP0);
        NVIC_ClearPendingIRQ(RTC_IRQn);
        xTickPass = 1;
        xPMGoToSleepMode(sleep);  /* Go in sleep mode */
          }

          /* Reset ucFlagISR if SSM issues a request before idle task */
          ucFlagISR = 0;

          /* Re-enable either interrupts and tasks */
          portEXIT_CRITICAL();
          xTaskResumeAll();
      }
  #endif

  #if ((configTICKLESS == 0)  && (configUSE_POWERMANAGER == 1))

      vTaskSuspendAll();
      portENTER_CRITICAL();

      /* Retrieve the sleep mode to use for idle task */
      sleep = ucPMGetSleepMode();
      xPMGoToSleepMode(sleep); /* Go in sleep mode */
      /* Reset ucFlagISR if SSM issues a request before idle task */
      ucFlagISR = 0;

      portEXIT_CRITICAL();
      xTaskResumeAll();
  #endif

}
```

Listing B.1: vPMIdleSleepMode()

B.2 xPMBreakEvenTime()

This function is implemented inside `PowerManager.c`.

```c

static portBASE_TYPE xPMBreakEvenTime ( portTickType xNextTickTask, portTickType ↩
     xNextTickCoRoutine){

  portCHAR NumTickBreakEven;

  /* If xNextTickTask or xNextTickCoRoutine is equal 0. It means that a task
   * or co-routine needs to run with same priority of idle task (or pending).
   * Thus, we cannt go in sleep mode. Break even time is not satisfied.
   */

        #if configUSE_CO_ROUTINES == 1
            if (( xNextTickTask == 0) || ( xNextTickCoRoutine == 0 ))
                return pdFAIL;
        #else
            if ( xNextTickTask == 0 )
                return pdFAIL;
        #endif

        NumTickBreakEven = MinTICK; /* Mimimum ticks required  to use tickless mode↩
              */

        /* If the number ticks for a task or a co-routine is less than break even
         * condition (sum of all factors) the MCU does not use Tickless mode.
         */
        #if configUSE_CO_ROUTINES == 1
            if (( NumTickBreakEven >=  ( xNextTickTask - xTaskGetTickCountTickless↩
                 ())) || ( NumTickBreakEven >= ( xNextTickCoRoutine - ↩
                 xTaskGetTickCountTickless())))
                return pdFAIL;
            else
                return pdPASS;
        #else
            if ( NumTickBreakEven >= ( xNextTickTask - xTaskGetTickCountTickless())↩
                 )
                return pdFAIL;
            else
                return pdPASS;
        #endif

    return pdFAIL;
}
```

Listing B.2: xPMBreakEvenTime()

B.3 RTC_Handler()

This function is implemented inside `PowerManager.c`.

```c
void RTC_IRQHandler  ( void )
{
  unsigned long ulDummy;

     if ( RTC->IF & RTC_IF_COMP0 ){

         if (( xPMTicklessENB == TICKLESS )){

            #if configPRESC == 1
        vTaskUpdateTickCountFromISR(TickToWakeup);
      #else
        vTaskUpdateTickCountFromISR(TickToWakeup/(RTC_FREQ/configTICK_RATE_HZ));
      #endif

      #if (configUSE_CO_ROUTINES == 1)
         /* I keep update the coroutine tick. Thus, I avoid to perform
        * (waste time and energy) all missed ticks when coroutine-
        * scheduling is being called.
        */
        prvCheckDelayedListTickless();
      #endif
         }

         /* Set next tick time */
         RTC_IntClear(RTC_IFC_COMP0);
         RTC_CompareSet(0,((RTC_FREQ/PRESC)/configTICK_RATE_HZ));

         /* If using preemption, also force a context switch. */
         #if configUSE_PREEMPTION == 1
                *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
         #endif

         if (xPMTicklessENB != TICKLESSNOPREEMPTED)
         {
             ulDummy = portSET_INTERRUPT_MASK_FROM_ISR();
             {
                   vTaskIncrementTick();
             }
             portCLEAR_INTERRUPT_MASK_FROM_ISR( ulDummy );
         }

         /* Reset the tickless variable in running mode */
         xPMTicklessENB = TICKRUNNING;
         xTickPass = 0;
     }

}
```

Listing B.3: RTC IRQHandler()

B.4 vTaskUpdateTickCountFromISR()

This function is implemented inside `task.c`.

```
void vTaskUpdateTickCountFromISR( portTickType xNumTicks )
  {
  unsigned portBASE_TYPE uxSavedInterruptStatus;
  portTickType temp,i;

    uxSavedInterruptStatus = portSET_INTERRUPT_MASK_FROM_ISR();

    /* I update the variable immediately with N-1 ticks. After that, on the last ↩
         tick
         * I run vTaskIncrementTick to check if the timeout of some tasks is ↩
              expired.
     */
    if ( xNumTicks  > 1 )
        {
      temp = xTickCount + (xNumTicks - 1);

      /*An overflow occurs, I manage it swapping the two delayed lists*/
      if ( xTickCount >  (xTickCount + xNumTicks) )
            {
          xTickCount = portMAX_DELAY;
          vTaskIncrementTick();  /* Exchange two delayed list */
          if ( (xTickCount + xNumTicks) > 0)
          for ( i = 0; i < temp; i++)
            vTaskIncrementTick(); /* Update all ticks after 0 */
      }else
        {
          xTickCount = temp;
          vTaskIncrementTick();
        }
      }

    /* I update the variable with only 1 tick*/
    if ( xNumTicks == 1 )
      vTaskIncrementTick();

    portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptStatus );

  }
```

Listing B.4: vTaskUpdateTickCountFromISR()

B.5 xTaskNextTick()

This function is implemented inside `task.c`.

```
  portTickType xTaskNextTick ( void ) {

    if (uxTopReadyPriority > tskIDLE_PRIORITY || xPendingReadyList.uxNumberOfItems ↩
        > 0 || pxReadyTasksLists[tskIDLE_PRIORITY].uxNumberOfItems > 1)
      return 0;

    /*If there are no tasks to schedule, then I return the time of first task to ↩
        wake up (delayed list) */
        return xNextTaskUnblockTime;
  }
```

Listing B.5: xTaskNextTick()

B.6 xCoRoutineNextTick()

This function is implemented inside `routine.c`.

```
  portTickType xCoRoutineNextTick ( void ) {

      /* If xCoRoutineTickCount is not synchronized with xTickCount, then the next ↩
          two functions update either xCoRoutineTickCount and all lists*/
    prvCheckPendingReadyListTickless();

    prvCheckDelayedListTickless();

    /* Now I can check if there is a co-routine ready to be executed. If any, ↩
        return 0*/
    if( (uxTopCoRoutineReadyPriority > 0) || (xPendingReadyCoRoutineList.↩
        uxNumberOfItems > 0) || (pxReadyCoRoutineLists[0].uxNumberOfItems > 0) )
      return 0;

    if( (pxDelayedCoRoutineList != NULL) && (pxDelayedCoRoutineList->↩
         uxNumberOfItems > 0))
      return (pxDelayedCoRoutineList->xListEnd.pxNext->xItemValue);
    else
      return portMAX_DELAY;
  }
```

Listing B.6: xCoRoutineNextTick()

B.7 prvCheckPendingReadyListTickless() and prvCheck-DelayedListTickless()

Thes functions are implemented inside `routine.c`.

```
void prvCheckDelayedListTickless( void )
{
  corCRCB *pxCRCB;
  portTickType temp;

  temp = xTaskGetTickCountTickless();

  /* I update variable when an overflow occurs */
  if ( xLastTickCount > temp )
  {
    xCoRoutineTickCount = portMAX_DELAY - 1; /* Remove delayed tasks (if any) ↩
         before to overflow */
    xPassedTicks = 2 + temp;  /* I update two ticks before portMAX_DELAY and all ↩
         ticks beyond 0 */
  }

  /* I update variable in normal case */
  if ( xLastTickCount < temp  )
  {
    xPassedTicks = 1;
    xCoRoutineTickCount += ( temp - xLastTickCount ) - 1;
  }

  while( xPassedTicks )
  {
    xCoRoutineTickCount++;
    xPassedTicks--;

    /* If the tick count has overflowed we need to swap the ready lists. */
    if( xCoRoutineTickCount == 0 )
    {
      xList * pxTemp;

      /* Tick count has overflowed so we need to swap the delay lists.  If there ↩
           are
      any items in pxDelayedCoRoutineList here then there is an error! */
      pxTemp = pxDelayedCoRoutineList;
      pxDelayedCoRoutineList = pxOverflowDelayedCoRoutineList;
      pxOverflowDelayedCoRoutineList = pxTemp;
    }

    /* See if this tick has made a timeout expire. */
    while( listLIST_IS_EMPTY( pxDelayedCoRoutineList ) == pdFALSE )
    {
      pxCRCB = ( corCRCB * ) listGET_OWNER_OF_HEAD_ENTRY( pxDelayedCoRoutineList );

      if( xCoRoutineTickCount < listGET_LIST_ITEM_VALUE( &( pxCRCB->xGenericListItem ) ) )
      {
        /* Timeout not yet expired. */
        break;
      }

      portENTER_CRITICAL();
      {
        /* The event could have occurred just before this critical
        section.  If this is the case then the generic list item will
        have been moved to the pending ready list and the following
        line is still valid.  Also the pvContainer parameter will have
        been set to NULL so the following lines are also valid. */
        vListRemove( &( pxCRCB->xGenericListItem ) );

        /* Is the co-routine waiting on an event also? */
        if( pxCRCB->xEventListItem.pvContainer )
        {
          vListRemove( &( pxCRCB->xEventListItem ) );
        }
      }
      portEXIT_CRITICAL();

      prvAddCoRoutineToReadyQueue( pxCRCB );
    }
  }

  xLastTickCount = xCoRoutineTickCount;
}

static void prvCheckPendingReadyListTickless( void )
{
  /* Are there any co-routines waiting to get moved to the ready list?  These
  are co-routines that have been readied by an ISR.  The ISR cannot access
  the ready lists itself. */
  while( listLIST_IS_EMPTY( &xPendingReadyCoRoutineList ) == pdFALSE )
  {
    corCRCB *pxUnblockedCRCB;

    /* The pending ready list can be accessed by an ISR. */
    portENTER_CRITICAL();
    {
      pxUnblockedCRCB = ( corCRCB * ) listGET_OWNER_OF_HEAD_ENTRY( (&xPendingReadyCoRoutineList) );
      vListRemove( &( pxUnblockedCRCB->xEventListItem ) );
    }
    portEXIT_CRITICAL();

    vListRemove( &( pxUnblockedCRCB->xGenericListItem ) );
    prvAddCoRoutineToReadyQueue( pxUnblockedCRCB );
  }
}
```

Listing B.7: prvCheckPendingReadyListTickless() prvCheckDelayedListTickless()

B.8 vPMTicklessFromISR()

This function is implemented inside `PowerManager.c`.

```
void vPMTicklessFromISR( portBASE_TYPE xHigherPriorityTaskWoken){

  portTickType Add = 0, temp;
```

```

/* If tickless is running I need to check if context switched is required */
if ( xPMTicklessENB != TICKRUNNING){

    /* Context switching is required */
    if  (xHigherPriorityTaskWoken == pdTRUE){
        /* This section calculates the time elapsed since the first ISR occured.
           Time is stored in Add. This value is immediately added in xTickCount.
        */
        switch (xPMTicklessENB){

            case TICKLESS:
          #if configPRESC == 1
          Add = RTC_CounterGet();
          #else
          Add = (RTC_CounterGet()/(RTC_FREQ/configTICK_RATE_HZ));
          #endif
                    break;

            case TICKLESSNOPREEMPTED:
          #if configPRESC == 1
          Add = RTC_CounterGet() - CounterBefore;
          #else
                      Add = (RTC_CounterGet()/(RTC_FREQ/configTICK_RATE_HZ)) - ↩
                           CounterBefore;
          #endif
                    CounterBefore = 0;
                    break;

            case TICKLESSPREEMPTED: /* Number of ticks elapsed since last check*/
          #if configPRESC == 1
          Add = RTC_CounterGet() - CounterBefore;
          #else
                      Add = (RTC_CounterGet()/(RTC_FREQ/configTICK_RATE_HZ)) - ↩
                           CounterBefore;
          #endif
                    CounterBefore = 0;
                    break;

            default: Add = 0;
        }

        /* If using preemption, also force a context switch. */
        #if configUSE_PREEMPTION == 1
              *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
        #endif

        /* Now the RTC behaves as a normal tick timer. It does not work in a
           tickless mode anymore.
         */
        if ( ((xPMTicklessENB == TICKLESS) || (xPMTicklessENB == ↩
             TICKLESSNOPREEMPTED)) ){
            xPMTicklessENB = TICKLESSPREEMPTED;  /* Now the system runs in ↩
                 preempted mode */

            /* Reset Counter of RTC again */
            RTC->CTRL &= ~(RTC_CTRL_EN_DISABLE);
            while (RTC->SYNCBUSY & RTC_SYNCBUSY_CTRL);

            /* Set RTC as tick timer */
            RTC_CompareSet(0,((RTC_FREQ/PRESC)/configTICK_RATE_HZ));

            /* Start Counter of RTC again */
            RTC->CTRL |= RTC_CTRL_EN;
            while (RTC->SYNCBUSY & RTC_SYNCBUSY_CTRL);
        }

        /* I clear all pending interrupt because I have already updated the
```

```
            variable of xTickCount. */
         RTC_IntClear(RTC_IFC_COMP0);
         NVIC_ClearPendingIRQ(RTC_IRQn);
     }

     /* If context switching is not required I only have to keep xTickCount ↩
          updated */
     if ( xHigherPriorityTaskWoken == pdFALSE){

         /* I check the number of ticks lost during execution of ISRs */
         if ( (xPMTicklessENB == TICKLESS)||(xPMTicklessENB == TICKLESSNOPREEMPTED↩
              ) ){

         #if (configPRESC == 1)
        temp = RTC_CounterGet();
       #else
        temp = (RTC_CounterGet()/(RTC_FREQ/configTICK_RATE_HZ));
       #endif
             if ( Counter == 0)
                Add = temp;
             else
                Add = temp - CounterBefore;

             /* Right now the system runs in nopreempted tickless mode */
             xPMTicklessENB = TICKLESSNOPREEMPTED;
             Counter ++;
             CounterBefore = temp;
         }

         /* If a previous ISR was running in PREEMPTED TICKLESS MODE, I keep
            counting the number of ticks lost after last executed ISR.
          */
         if ( xPMTicklessENB == TICKLESSPREEMPTED ){
       #if configPRESC == 1
       Add = RTC_CounterGet() - CounterBefore;
       #else
       Add = (RTC_CounterGet()/(RTC_FREQ/configTICK_RATE_HZ)) - CounterBefore;
       #endif
             CounterBefore = 0;
         }
     }
     /* Update the number of 'ticks lost' in xTickCount variable*/
     vTaskUpdateTickCountFromISR(Add);
 }else
   {
    /* This section issues a context switching when we use this function as
      portEND_SWITCHING_ISR.
     */
       if ((xHigherPriorityTaskWoken != pdFALSE)){
           /* If using preemption, also force a context switch. */
           #if configUSE_PREEMPTION == 1
               *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET;
           #endif
       }
 }

}
```

Listing B.8: vPMTicklessFromISR()

Bibliography

[1] The Climate Savers Computing Initiative is a nonprofit group of eco-conscious consumers, businesses and conservation organizations dedicated to reducing the energy consumption of computers. http://www.climatesaverscomputing.org/

[2] State of the Union 2011: WINNING THE FUTURE. Conference hold 25 January 2011. http://www.whitehouse.gov/state-of-the-union-2011

[3] Martin Tverdal. Operating system directed power reduction on EFM32. Master thesis project.

[4] EFM32 Introduction White Paper. http://downloads.energymicro.com/pdf/efm32 Available at: _introduction_white_paper.pdf (Accessed 17 Febraury 2011)

[5] Energy Micro "Reference Manual EFM32G Microcontroller Family". Available at: http://www.energy.com/downloads (Last Access 17 June 2011)

[6] An Introduction to the ARM Cortex-M3 Processor Shyam Sadasivan 2006. http://www.energymicro.com/images/stories/technlogy/arm_introtocortex-m3.pdf

[7] Angelo Spalluto. Green Computing: Energy Aware RTOS for EFM32. TDT4592 fall semester 2010.

[8] Joseph Yiu. The definitive guide to the ARM-Cortex-M3.

[9] Application note: EFM32_dvk_manual. Energy Micro. Available at: http://www.energymicro.com/downloads/tools-documents

[10] FreeRTOS website. www.freertos.org

[11] Chi-Hong Hwang and Allen C.-H. Wu. 2000. A predictive system shutdown method for energy saving of event-driven computation. ACM Trans. Des. Autom. Electron. Syst. 5, 2 (April 2000), 226-241. DOI=10.1145/335043.335046 http://doi.acm.org/10.1145/335043.335046

[12] Srinivasan, B.; Pather, S.; Hill, R.; Ansari, F.; Niehaus, D.; , "A firm real-time system implementation using commercial off-the-shelf hardware and free software," Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE , vol., no., pp.112-119, 3-5 Jun 1998 doi: 10.1109/RTTAS.1998.683194

[13] Giovani Gracioli, Danillo Moura Santos, Roberto de Matos, Lucas Francisco Wanner, and Ant&#244;nio Augusto Fr&#246;hlich. 2008. One-Shot Time Management Analysis in EPOS. In Proceedings of the 2008 International Conference of the Chilean Computer Science Society (SCCC '08). IEEE Computer Society, Washington, DC, USA, 92-99. DOI=10.1109/SCCC.2008.13 http://dx.doi.org/10.1109/SCCC.2008.13

[14] J Stultz, N Aravamudan, D.H. (2005) We Are Not Getting Any Younger: A New Approach to Time and Timers: Ottawa Linux Symposium Proceedings, OLS2005 Proceedings

[15] T. Gleixner, D. Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems: Ottawa Linux Symposium Proceedings, OLS2006 Proceedings.

[16] Dan Tsafrir. 2007. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In Proceedings of the 2007 workshop on Experimental computer science (ExpCS '07). ACM, New York, NY, USA, , Article 4 . DOI=10.1145/1281700.1281704 http://doi.acm.org/10.1145/1281700.1281704

[17] C. Michael Olsen and Chandra Narayanaswami. 2006. PowerNap: An Efficient Power Management Scheme for Mobile Devices. IEEE Transactions on Mobile Computing 5, 7 (July 2006), 816-828. DOI=10.1109/TMC.2006.103 http://dx.doi.org/10.1109/TMC.2006.103

[18] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN 04). IEEE Computer Society, Washington, DC, USA, 455-462. DOI=10.1109/LCN.2004.38 http://dx.doi.org/10.1109/LCN.2004.38

[19] http://www.tinyos.net/

[20] http://tiros.sourceforge.net/

[21] http://www.quasarsoft.com/

[22] AVIX-RT, The Hybrid RTOS. Available at: http://www.avix-rt.com/ (Accessed 18 March 2011)

[23] CMX Systems, Inc. Available at: http://www.cmx.com/index.htm (Accessed 18 March 2011)

[24] RTOS-the heart of good power management. Available at: http://www.iar.com/website1/1.0.1.0/1197/1/ (Accessed 18 March 2011)

[25] Application note: Temperature Compensated Calendar. Energy Micro Available at: http://cdn.energymicro.com/dl/an/pdf/an0006_efm32_calendar.pdf (Accessed 28 May 2011)

[26] Application note 3566: Timekeeping Accuracy, Automatic and Affordable. Available at: http://pdfserv.maxim-ic.com/en/an/AN3566.pdf (Accessed 28 May 2011)

[27] Understanding TCXOs. MPD Microwave Product Digest. Ramon M. Cerda, Crystek Crystals Corp. Available at: http://www.mpdigest.com/issue/Articles/2005/apr2005/crystek/Default.asp (Accessed 28 May 2011)

[28] Datasheet and Manual: EFM32 Cortex-M3 Reference Manual. Energy Micro Available at: http://cdn.energymicro.com/dl/devices/pdf/d0002_efm32_cortex-m3_reference_manual.pdf (Accessed 28 May 2011)

[29] Buy or roll your own OS? Neither with FreeRTOS. Available at: http://www.eetimes.com/discussion/ guest-editor/4027638/Buy-or-roll-your-own-OS-Neither-with-FreeRTOS- (Last Access 17 June 2011)

[30] The results for 2010 are in!. Available at: http://www.eetimes.com/design/embedded/4008920/The-results-for-2010-are-in- (Last Access 17 June 2011)