Norwegian University of
Science and Technology

Operating system directed power reduction on EFM32

Martin Tverdal

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Lasse Natvig, IDI
Co-supervisor: Marius Grannæs, Energy Micro

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

Problem Description

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. EFM is short for energy friendly microcontrollers. FreeRTOS is a small and free open source OS targeted for embedded devices. The goal of this master thesis project is to get FreeRTOS to run on an EFM32 micro controller with as low power consumption as possible. Central subtasks are:

Implementing/Exploring a tickless scheduler for FreeRTOS.
Implementing/Exploring a peripheral driver structure for low-power.

Assignment given: 15. January 2010
Supervisor: Lasse Natvig, IDI

Abstract

Power consumption has become a major concern of embedded systems. Currently FreeRTOS wastes a power waking up regularly to keep track of time. In this work FreeRTOS is modified to sleep when there is no work for the CPU to be done. Timekeeping while sleeping is done by a low frequency oscillator, consuming very little power. Drivers for peripherals have been developed, in order to optimise power consumption even more. Battery life time has been increased from 56 hours to 1867 hours for a simple self made benchmark. The goal is to get the changes into the official FreeRTOS distribution, but it has not been accepted yet. However, a customer of Energy Micro has started to develop an application based on this design.

Acknowledgments

I want to thank my supervisor, Lasse Natvig for his guidance and encouragement throughout this spring. Marius Grannæs deserves my best, for always being sincerely helpful and supportive. Last but not least I want to thank my friends in office 443b for all the help, and for making university work fun. Especially Kjetil Wathne Oftedal for helping me in practical matters.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Energy Micro . . . 1
- 1.3 IAR Embedded Workbench . . . 1
- 1.4 Goals . . . 2
- 1.5 Contributions . . . 2
- 1.6 Structure of this report . . . 3

2 Background **5**
- 2.1 ARM Cortex M3 . . . 5
- 2.2 EFM32 . . . 5
  - 2.2.1 Interrupts and Sleep modes . . . 5
  - 2.2.2 Peripherals . . . 8
  - 2.2.3 Timing peripherals . . . 10
  - 2.2.4 HFXO and HFRCO . . . 11
- 2.3 FreeRTOS . . . 11
  - 2.3.1 Tasks . . . 11
  - 2.3.2 Co routines . . . 12
  - 2.3.3 FreeRTOS and systick . . . 14
  - 2.3.4 Interrupts . . . 14
- 2.4 Dynamic Power Management . . . 15
  - 2.4.1 Linux power management governors . . . 15
- 2.5 Timekeeping . . . 16

3 FreeRTOS on EFM32 **19**
- 3.1 IAR Project . . . 19
- 3.2 Demo application . . . 19

4 Tickless idle **21**
- 4.1 Difference in power consumption . . . 22
- 4.2 Chosen solution . . . 23
- 4.3 Interrupt handling . . . 23
- 4.4 Time until next event . . . 24
- 4.5 Calculating how long to sleep . . . 26
- 4.6 Calculating how long was slept . . . 27
  - 4.6.1 Storing reminder . . . 28
  - 4.6.2 xTickCount following RTC . . . 29
- 4.7 Consequences for interrupt latency . . . 29

5 Managing EFM32 energy modes **31**
- 5.1 #1 Clocks enabled . . . 31
- 5.2 #2 Explicit control . . . 31
- 5.3 Chosen solution . . . 33

6 Methodology **35**
6.1 Testing 35
6.2 Benchmark 35
6.2.1 Effect of load 37
6.3 Versions of FreeRTOS tested 37
6.3.1 Versions with DMA driver 37
6.4 Power measurement 38

7 Results **39**
7.1 Testing 39
7.2 Power consumption 40

8 Conclusion **47**
8.1 Future work 47

A Demo application **51**
A.1 main.c 51
A.2 lcdtest.c 52
A.3 ledtest.c 53
A.4 ParTest.c 54
A.5 startup_efm32.s 55

B Code **61**
B.1 energymode.c 61
B.2 checktiming.c 65
B.3 Application 66
B.3.1 main.c 66
B.3.2 measurement.c 69
B.3.3 adc.c 72
B.4 Diff for task.c in FreeRTOS 73
B.5 C++ program simulating GPS 75
B.6 Drivers 77
B.7 Serial 77
B.7.1 Serial with DMA 80
B.8 I2C 85
B.8.1 Modified driver from EFMLIB 85
B.8.2 My own I2C Driver 87
B.8.3 Script to process power csv files 91

C Figures **92**

List of Figures

1 Schematic of the Cortex M3 core. 6
2 Energy Modes on EFM32 7
3 Peripherals on EFM32 7
4 Peripherals used for timekeeping in EFM32 10
5 Shows how a delayed list may look like. 13
6 Interrupt priorities in FreeRTOS. 14
7 Power state machine for StrongARM SA-1100 16
8 Code handling overflow 22
9 vTaskTickoverflow() 22
10 Sequence diagram of code 24
11 xTaskNextTick function in task.c 25
12 xCoRoutineNextTick function in croutine.c 26
13 xCoRoutineNextTick in croutine.c 27
14 Cause of error in timekeeping 28
15 Modified idle task 30
16 Benchmark application 36
17 Drift on clock 39
18 Drift on clock 40
19 Drift on clock 41
20 Power consumption with different version of FreeRTOS 43
21 Power consumption with different versions of FreeRTOS 44
22 Battery life time using 220mAh battery 45
23 Battery life time using 220mAh battery 46
24 Bit assignments in HFPERCLKEN0. 92
25 Bit assignments in HFCORECLKEN0. 93

List of Tables

1 Lists used for task management in FreeRTOS 12
2 Sub tasks, benchmark 35
3 Parameters to load benchmark 37

Abbreviations

ADC Analog to Digital Converter

AEM Advanced Energy Monitor

CMU Clock Management Unit

DMA Direct Memory Access

DPM Dynamic Power Management

EM Energy Mode

GCC GNU Compiler Collection

GPL GNU General Public License

HFRCO High Frequency Resistor Capacitor Oscillator

HFXO High Frequency Crystal Oscillator

IAR IAR Embedded Workbench

ISR Interrupt Service Routine

LETIMER Low Energy Timer. Clocked by low frequency clocks. Remains active in EM2

NVIC Nested Vector Interrupt Controller

MPU Memory Protection Unit

LCD Liquid Crystal Display

RTC Real Time Clock

SCB System Control Block

SRAM Static Random Access Memory

TCB Task Control Block

UART Universal Asynchronous Receiver/Transmitter

USART Universal Synchronous Asynchronous Receiver/Transmitter

WFE Wait For Event

WFI Wait For Interrupt

xTickCount Variable used in FreeRTOS to count number of tick interrupts.

1 Introduction

1.1 Motivation

Reduced power consumption prolongs the battery life time of embedded systems, and other systems running on battery. This is especially important on certain embedded devices which need expensive recharging/replacement of batteries. It can be costly, or even impossible to access the device when the battery runs out. For some devices the life time is over when the battery is exhausted. Examples of this can be motion sensors embedded into the concrete of buildings, medical equipment(implants), equipment on satellites or sensors on the sea bed. Portable energy sources such as kinetic energy, or solar panels produce little power and are expensive. Low power allows the use of smaller, lower cost solar panels. Reducing the power consumption also reduces the need for cooling, which lowers the cost of electrical equipment.

Other reasons to reduce the power consumption of microprocessors is that power expensive, Rivoire et al. [Rivoire et al., 2007] reports that in data centres it can potentially exceed the cost of purchasing hardware. Using less energy reduces the emission of greenhouse gasses which, needless to say in 2010 is good for the environment. Reduced power consumption also reduces the need for cooling, which aside from reducing cost is also attractive because it makes computer systems make less noise. This can be an important selling point in home electronics. Less cooling requirement makes computer systems take less space, which is important in every segment of the computer market, from big computer centres to small embedded systems.

1.2 Energy Micro

This project is done in cooperation with Energy Micro. Energy Micro is a Norwegian semiconductor company focusing on 32 bit microcontrollers with ultra low energy consumption. The founders was also the founders of Chipcon AS which was acquired by Texas Instruments for approximately 200M USD. The EFM32 Gecko (Energy Friendly Microcontroller) has a core based on the ARM Cortex-M3. It was announced on the 21. October 2009 with prototype chips available. The chips were made available for ordinary distribution in February 2010. The chip used in this work is an early engineering sample.

1.3 IAR Embedded Workbench

The tool chosen for this project is IAR Embedded Workbench (IAR). It is a development environment from IAR Systems, a Swedish computer technology company. It includes a C/C++ compiler, and it got support for EFM32 early in my project. The alternative was GNU Compiler Collection (GCC). Since IAR is expensive, some customers of Energy Micro demanded GCC support. However, at the start of my project there was no support for EFM32 in GCC. In order to get started on work relevant to my thesis IAR was chosen. In IAR everything worked right out of

the box, interrupt routines, single step debugging and uploading my programs to the development kit worked well. If GCC was to be used, much time would have spent too much time on tasks not relevant for my thesis.

1.4 Goals

Goal 1. Introduce basic EFM32 support in FreeRTOS, without any support for energy modes.

Goal 1 is to get a basic port to the EFM32 introduced officially in FreeRTOS. Initially without any support for tickless idle or any other form of taking advantage of the different energy modes.

A basic port was made as a first step, partly to make it easier to get acceptance for a bigger change of the kernel itself at a later point if a port is already accepted in the main distribution, and partly because customers of Energy Micro were expecting RTOS support as soon as possible.

Goal 2. Implement FreeRTOS support for tickless idle on EFM32.

Implement a way for FreeRTOS to be totally idle while sleeping. This means that if all tasks are to sleep for 1 second, the core should also sleep for the whole second, without waking up to execute instructions. The kernel should also support keeping track of time while the EFM32 is in Energy Mode (EM)2. The complication here is that in EM2 the high frequency clock used by the core is turned off. The timekeeping has to be done with the low frequency clock.

Goal 3. Implement FreeRTOS management for different energy modes.

Implement support for FreeRTOS to manage the Energy Modes on EFM32. This means finding a way for FreeRTOS to determine what Energy Mode to go to when the core is idle.

Goal 4. Get changes into official FreeRTOS.

Get the changes developed in order to achieve goal 2 and 3 included officially in FreeRTOS. Since FreeRTOS is licensed under a Modified GNU General Public License (GPL) license, I am free to make any changes I want to the kernel as long as they are made open source. This means that Energy Micro AS is free to distribute my changes to their customers. It is however a goal to get the changes submitted to the official FreeRTOS. If the modifications gets included officially in FreeRTOS, customers of Energy Micro would be more assured that the quality is high.

1.5 Contributions

The contributions of this work is mainly a fulfilling of goals 1-3 in the previous section. In addition an application has been developed to illustrate how to use the modified FreeRTOS in order to reduce power consumption. This has involved

using the I2C bus, serial bus and Analog to Digital Converter (ADC) in a way such that the EFM32 can enter the lowest possible sleep state while these modules do work.

A serial driver has been developed which enables the serial bus to stay in EM while the Direct Memory Access (DMA) copies data to to/from the transmit/receive buffer. When a transmission is complete an interrupt is issued waking up the core. When a linefeed is received or the DMA buffer is full, the core is also woken up by an interrupt. When the ADC is set up to perform a conversion, since ADC needs the high frequency oscillator, FreeRTOS is instructed not to enter any Energy Mode below EM1. The I2C is used to read the temperature. When an I2C transfer is ongoing FreeRTOS is instructed not to enter any energy mode below EM1.

A customer of Energy Micro from the US has started using my code in a project developing a new product where battery life time of multiple months is important. Because of competition the customer wants not to be mentioned by name in this thesis.

1.6 Structure of this report

Section 2 explains some background information which can be useful to read in order to understand the rest of the report better. Section 3 deals with the first part of my thesis, porting FreeRTOS to the EFM32 architecture. Section 4 deals with implementing tickless idle in FreeRTOS. Section 5 deals with making FreeRTOS able to go to correct energy mode. Section 6.2 explains the benchmark used, while section 6.4 explains how power consumption has been measured. Section 7 presents the results of tests of power consumption and tests for drift on the clock.

2 Background

2.1 ARM Cortex M3

This paragraph is a small extension to the one in my project thesis [Tverdal, 2009]. Much of the information was found in Cortex-M3 Technical Reference Manual [arm, 2005]. The EFM32 has a CPU core based on the ARM Cortex-M3. This core implements the Thumb-2 instruction set, which has both 16 and 32 bit instructions. It has a 3 stage pipeline, fetch, decode and execute. It has Harvard architecture, with separate data and instruction memory. Data memory is Static Random Access Memory (SRAM), while instruction memory is flash. Figure 1 shows a block diagram of the core of the Cortex M3. The ALU supports 32 bit integer multiplication in one cycle. It also has a hardware divider which can perform integer divides in 2-12 cycles, depending on the operands. The division is completed faster if the dividend and divisor is closer in size. It also includes a *Nested Vector Interrupt Controller (NVIC)* interface (see section 2.2.1) and an optional *ETM* (Embedded Trace Macrocell) interface. The ETM is optional and provides debug and trace facilities. *DAP* (Debug Access Port) is implemented with Serial Wire Debug, using only 2 pins (clock and data).

2.2 EFM32

Figure 3 is a schematic overview of the peripherals on the EFM32. Figure 2 is an overview of the energy modes. There are in total 5 Energy Mode (EM)s. The most relevant ones for this thesis is EM0, EM1 and EM2. EM0 is the run mode. Everything is turned on and the core is executing instructions. In EM1 the core and the Memory Protection Unit (MPU) is shut down. Both the high frequency clock and the low frequency clock is running, meaning that the high frequency peripherals and the low frequency can be active. In EM2 the high frequency clock is turned off, but the low frequency clock is running. Which means that only the low frequency peripherals can be used. In EM3 the low frequency clock is also disabled, leaving nothing to keep track of time, the EFM32 can only be woken by the events listed in Figure 2. The flash memory code size ranges from 8KB to 128KB while SRAM data memory size ranges from 2KB to 16KB. More information can be found in Reference Manual, EFM32G Microcontroller Family [EFM32 Manual,]

2.2.1 Interrupts and Sleep modes

The NVIC on the EFM32 supports 8 priority levels for interrupts. Priorities are from 0-7, with 0 being the highest priority while 7 is the lowest. There is three exception mask registers which can affect the handling of exceptions by the processor. The technical details in this sections were found in Cortex-M3 Technical Reference Manual [arm, 2005].

Priority Mask Register Writing one to bit 0 prevents activation of all interrupts with configurable priority.

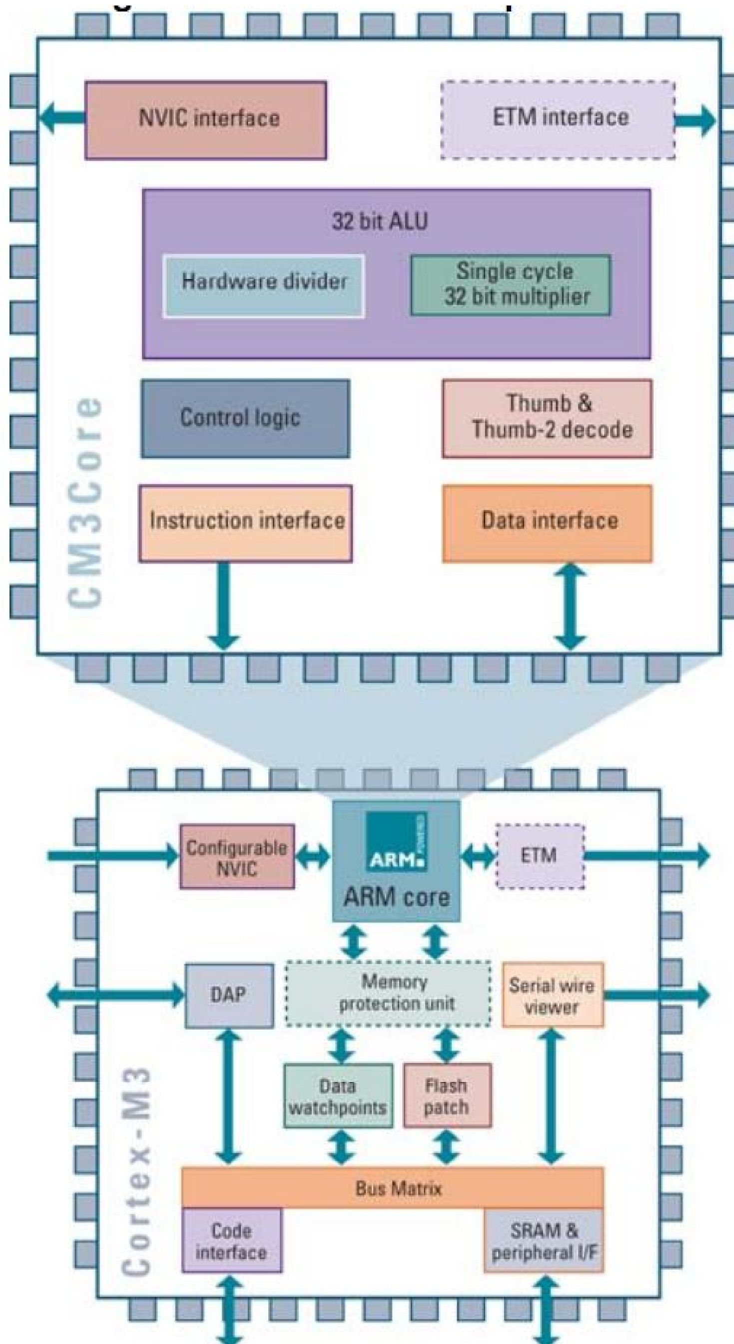

Figure 1: Schematic of the Cortex M3 core, taken from [Sadasivan, 2006]

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 IRQ, 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: Energy Modes on EFM32

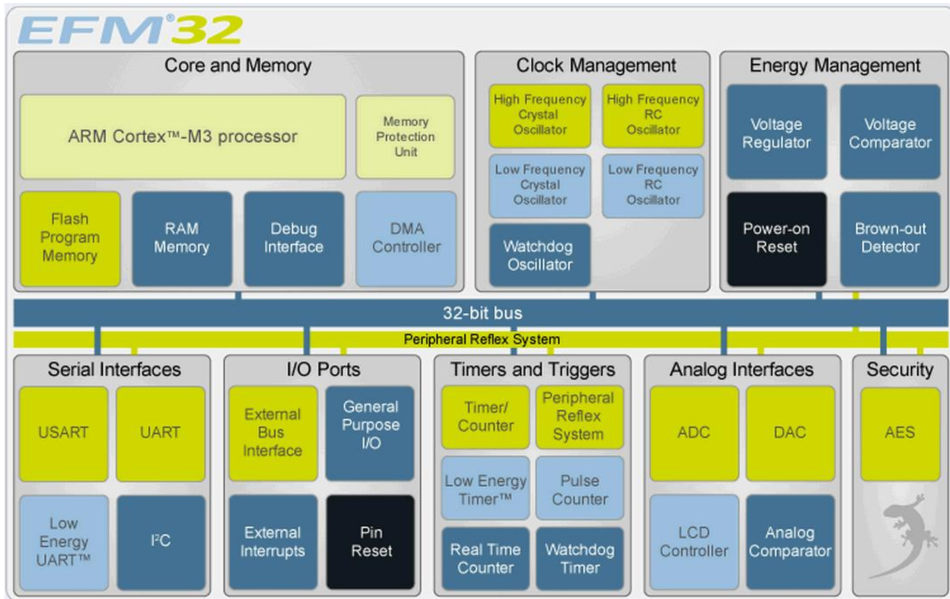

Figure 3: Peripherals on EFM32

Fault Mask Register Writing one to bit 0 prevents the activation of all exceptions except for Non-Maskable Interrupt.

Base Priority Mask Register If for example a 5 is written to this register it prevents the activation of all exceptions with priority lower than or equal to 5 (5, 6 and 7).

There are two instructions used for entering sleep modes, Wait For Interrupt (WFI) and Wait For Event (WFE). If SLEEPDEEP bit in the System Control Block (SCB) is set to 0, EM1 is chosen, if it is set to 1, EM2 is chosen. EM3 is equal to EM2, except that the low frequency clocks are stopped in EM3. They have to be stopped by software explicitly. In EM3 there is no way to keep to track of time.

WFI When the WFI instruction is issued EFM32 enters a low energy mode. It is woken by an interrupt with high enough priority to preempt execution, disregarding the Priority Mask Register. This means that even if the Priority Mask Register is set to disable interrupts, the core will be woken up by interrupts which would have high enough priority to wake the core had it not been set to disable interrupts. This means that if the Base Priority Mask Register is set to mask out all interrupts with priority below 5 (5, 6 and 7), interrupts with these priorities will not wake up the core. An interrupt with priority of 4 and above, will wake up the core regardless of Priority Mask Register.

When the core is woken up execution proceeds at the instruction after the WFI instruction if Priority Mask Register is set. If the Priority Mask Register is not set execution proceeds in the Interrupt Service Routine (ISR).

WFE The other instruction that can be used to enter low energy mode is WFE. If the SEVONPEND bit in the System Control Register is set, the core is woken up by all interrupts entering the Pending state, even if they are disabled or has too low priority to cause ISR entry. When woken up, execution proceeds at the next instruction after WFE if the pending interrupt does not have high enough priority to preempt execution, or it proceeds in the ISR if it does.

2.2.2 Peripherals

ADC Analog to Digital Converter is available in EM0 and EM1. It is used to convert an analog signal to a digital representation.

AES Advanced Encryption Standard is available in EM0 and EM1. It is a hardware accelerator for encrypting and decrypting with 128 or 256 bit keys.

DAC Digital to Analog Converter is available in EM0 and EM1. It is used to convert a digital value to an analog signal.

Analog Comparator is available in EM0,EM1,EM2 and EM3. It is used to compare two analog signals. It can monitor a signal to see if it passes a certain threshold while the EFM32 is in EM3, consuming very little power.

Peripheral Reflex System is available in EM0 and EM1 and allows very simple communication between peripherals.

Timer/Counter is available in EM0 and EM1 and keeps track of timing and counting events.

UART is available in EM0 and EM1 and is used for Universal Asynchronous Receiver/Transmitter (UART) communication.

USART is available in EM0 and EM1 and is used for Universal Synchronous Asynchronous Receiver/Transmitter (USART) communication.

DMA Controller Direct Memory Access is available in EM0, EM1 and EM2 and can move data while the core either sleeps, or is busy.

External Bus Interface is available in EM0 and EM1 and is used to access external devices. They are mapped to the memory space of the core, making them easy to use.

General Purpose I/O is used to communicate with external devices and can wake up the EFM32 in EM1, EM2 and EM3.

I2C Inter-Integrated Circuit interface supports communication on I2C buses. I2C is a common bus protocol used in embedded systems. It can be configured to wake up the EFM in EM1, EM2 and EM3.

LCD Controller is able to drive a Liquid Crystal Display (LCD) display with up to 4x40 segments in EM0, EM1 and EM2.

Low Energy TIMER can keep track of time and output waveforms in EM0, EM1, EM2 and EM3.

Low Energy UART is available in EM0, EM1 and EM2 and provides UART communication.

Real Time Counter is available in EM0, EM1 and EM2. It is a 24 bit timer used to keep track of time.

Pulse Counter is available in EM0, EM1, EM2 and EM3. It count pulses and can wake up the core.

Watchdog Timer is available in EM0, EM1, EM2 and EM3 and resets the EFM32 when a fault condition is reached. If the core does not reset the watchdog within the configured timeout period the watchdog resets the EFM32.

Brown-out Detector monitors the supply voltage in EM0, EM1, EM2 and EM3. It resets the EFM32 if it drops below a safe value.

Power-on Reset is available in all energy modes and monitors the supply voltage and signals when it reaches the operating value.

2.2.3 Timing peripherals

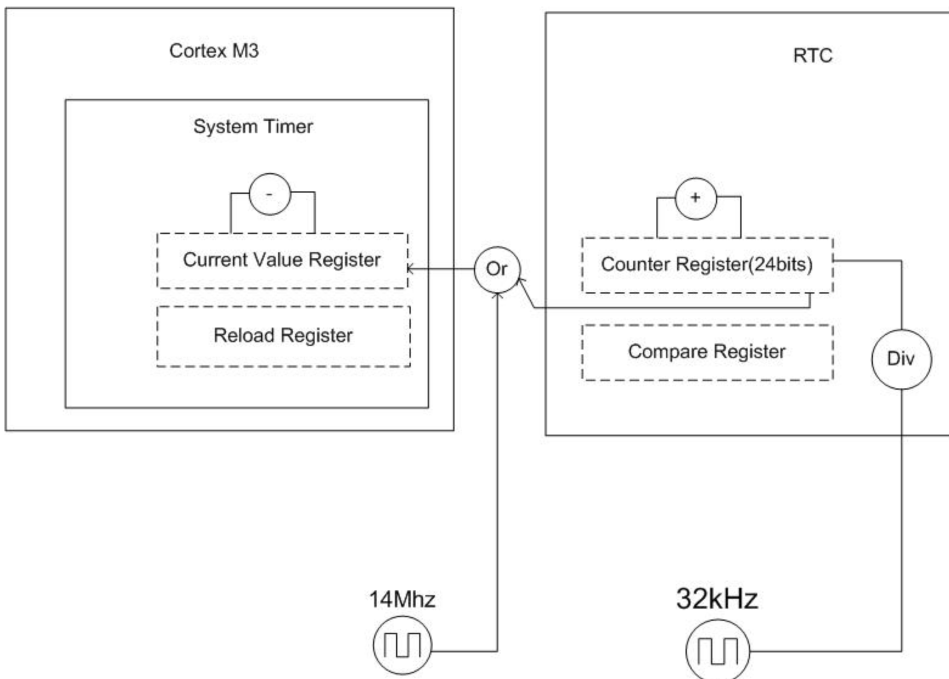

Figure 4: Peripherals used for timekeeping in EFM32

Figure 4 shows the peripherals relevant for this thesis when it comes to keeping track of time. The *Cortex M3* core has a built in *System Timer*, it works by decrementing *Current Value Register* every time the clock beats. When it reaches 0, an interrupt is issued to the core and it reloads the *Current Value Register* with the value set in the *Reload Register*. Then it continues to decrement the *Current Value Register*. This way a the System Timer can be used to generate a periodic interrupt. If for example 14'000'000 is written to the Reload Register, and the 14MHz clock is used, an interrupt occurs every second. The Current Value Register

can also be clocked by the least significant bit in the Real Time Clock (RTC), that way it can be clocked indirectly by the low frequency clock rather than the high frequency clock.

The RTC works in a similar way, it is however clocked by the low frequency 32kHz clock, which means it is available for timekeeping in EM2. The *Counter Register* (which is only 24 bits) is incremented every time the clock beats. The clock can be divided, to make it increment at a slower rate. When the value is equal to the value in the *Compare Register* an interrupt is issued to the Cortex M3 core, and starts to count up again from 0.

2.2.4 HFXO and HFRCO

The core clock can be generated either by High Frequency Crystal Oscillator (HFXO) or High Frequency Resistor Capacitor Oscillator (HFRCO). HFXO is more accurate but has a considerably longer wake-up time. According to [EFM32G890F128 Datasheet,] HFXO has a start up time of 400μs, while HFRCO has a start up time of only 0.6μs. When waking up from EM2 or EM3 when the high frequency oscillator is turned of, the EMF32 is always running with HFRCO. If HFXO is wanted, software has to enable it, then wait for it to start up before switching clock source. The Clock Management Unit (CMU) module of the EFM32 supports waiting for 8, 256, 1024 or 16384 cycles of the HFRCO. Since 1024 cycles of 14MHz is only 73μs, 16384 cycles has to be used. This equals to waiting for 1.1ms.

2.3 FreeRTOS

The information about FreeRTOS was found by reading the work of Sadasivan [Sadasivan, 2006], the FreeRTOS web site (`http://www.freertos.org/`) and the source code [FreeRTOS Code, 2010].

This paragraph is taken from my project thesis. There exists a vast amount of operating systems for microcontrollers, the decision to go with FreeRTOS came from Energy Micro. FreeRTOS is a small and free open source OS targeted for embedded devices. It contains approximately 4000 lines of code. It is a very simple OS, there is no support for complex memory management, no device drivers or any support networking. It lets the programmer create tasks with priorities and schedules them either cooperatively or preemptively. It uses a simple round robin algorithm within a priority, and does not schedule lower priorities as a long as tasks with higher priorities are ready. It provides mechanisms for tasks to communicate and share data safely (queues, semaphores and mutexes). FreeRTOS has been ported to many different architectures, including the Cortex-M3. The latest version also supports the MPU. on the Cortex-M3.

2.3.1 Tasks

Table 1 shows the lists used by the scheduler to manage lists. For every priority there is one list *ReadyTasksList[N]* of tasks ready to run. If configured in FreeR-

TOSConfig.h there is one list *TasksWaitingTerminiation* of tasks that is waiting to be deleted, and one list *SuspendedTaskList* with suspended tasks. Suspended tasks are tasks that are delayed indefinitely, and will not be made ready by the kernel itself no matter how long time elapses. They have asked to be delayed indefinitely by calling for example by calling vTaskDelay(MAX_DELAY).

The *PendingReadyList* list is used by the kernel for tasks that have been made ready while the scheduler has been suspended. It is needed since the scheduler can be suspended, meaning that it will not perform any tasks switches even if a task becomes ready. While the scheduler is suspended the ready list can not be modified. The task is instead added to the *PendingReadyList*, and moved to the Ready list when the scheduler is resumed.

The *DelayedTaskList* and *OverflowDelayedTaskList* is where delayed tasks are kept. They are sorted by wake up time. The reason for using two lists is that the wake up time might overflow. If at tick number 250 a task wants to sleep for 30 ticks, 250+30 will overflow and result in 25 (if a 8 bit counter is used). The task is then inserted into the overflowed list. When the Variable used in FreeRTOS to count number of tick interrupts. (xTickCount) variable overflows, the two lists are swapped. This implementation results in that the maximum time a task can sleep is 255 if a 8 bit counter is used. If a task was to sleep for 256 ticks when xTickCount was 150. 150+256 would result in 151.

List	Description
ReadyTaksList[N]	Tasks ready to run, one for each priority level.
TasksWaitingTermination	Tasks that has terminated, but not yet deleted
SuspendedTaskList	Suspended tasks will not get scheduled to run again by the scheduler.
PendingReadyList	Pending tasks will be put in the ReadyTaskList once the scheduler is re-enabled.
DelayedTaskList	Tasks that have been delayed. Sorted by wake up time.
OverflowDelayedTaskList	Due to arithmetic overflow, tasks delayed until after the tick count overflows are put here.

Table 1: Lists used for task management in FreeRTOS

Figure 5 shows how a delayed list looks like. *pxDelayedTaskList* points to a struct of type *xList*. The first variable in a *xList* tells how many items is in the list. The next variable *pxIndex* is used when traversing the list, to keep track of the next element to be returned. The *xListEnd* is the end marker. In this case it contains MAX_DELAY in *xItemValue*. The two other tasks in the list contain 24, and 45 in that position. Task x is delayed until xTickCount reaches 24, while task y is delayed until it reaches 45.

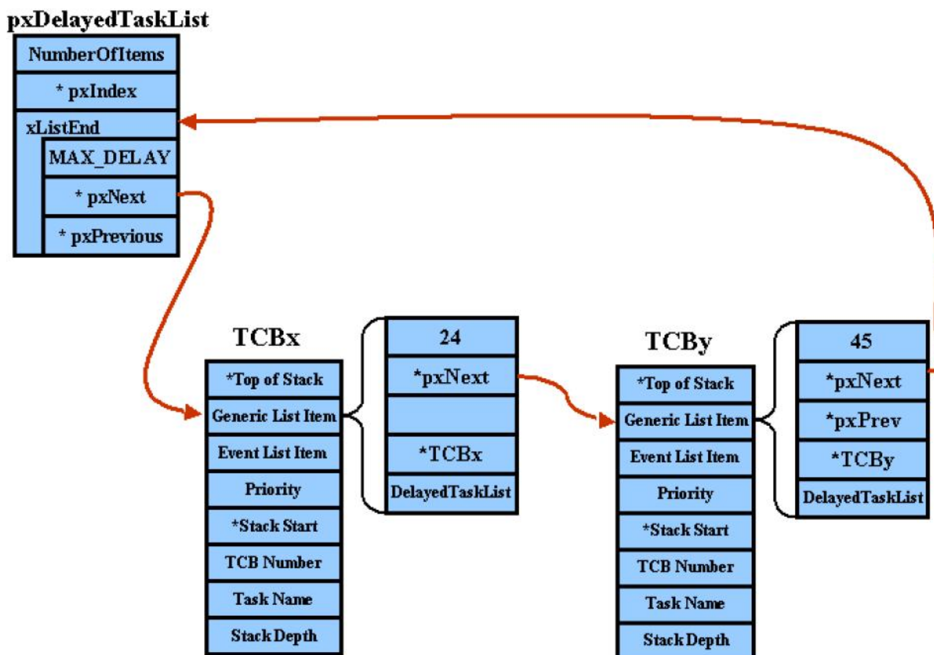

Figure 5: Shows how a delayed list may look like. Taken from [Sadasivan, 2006]

2.3.2 Co routines

FreeRTOS also has the concept of co routines. They are intended to be used on very small processors that have very little memory. To save memory, all co routines share stack. They are handled very similar to tasks, with a ready list for each priority, a pending ready list, a delayed list and an overflowed delayed list. The difference is that they all share a stack. The consequence of sharing a stack is that the variables declared on the stack can loose the value when the co routine is blocked. The way co routines are scheduled is worth noticing. They are scheduled by repeated calls to vCoRoutineSchedule(). The normal place to call this is in the application idle hook, which is a function repeatedly called by the idle task. This has the effect that co routine has a lower priority than tasks.

2.3.3 FreeRTOS and systick

FreeRTOS keeps track of time by incrementing a counter at a configurable rate, normally 1000Hz or 100Hz. The system timer of the Cortex M3 core is set up to issue an interrupt at one of these intervals. At every interrupt the xTickCount variable is incremented, and the scheduler also checks if it needs to perform any scheduler tasks.

2.3.4 Interrupts

Figure 6 illustrates how interrupts are handled in FreeRTOS. The macro configKERNEL_INTERRUPT_PRIORITY defines the priority all the kernel interrupts use, SysTick and PendSV. Systick is executed periodically in order to keep track of time, while PendSV is a software interrupt used for switching tasks. configKERNEL_INTERRUPT_PRIORITY is set to the lowest possible priority, which on EFM32 is 7 (The highest priority is 0). configMAX_SYSCALL_INTERRUPT_PRIORITY defines the highest priority an interrupt which uses the FreeRTOS API can have. An interrupt with priority below this value however only call API functions with names ending in FromISR. Interrupts with priority above configMAX_SYSCALL_INTERRUPT_PRIORITY is never delayed by anything the kernel does. They can not call any API functions in FreeRTOS.

2.4 Dynamic Power Management

Dynamic Power Management (DPM) is different techniques for turning off, or reducing the performance of components when they are idle. There is a lot of research on this topic. Benini et al. in [Benini et al., 2000] presents a survey of different techniques where figure 7 is used as an example. It shows the states of StrongARM SA-1100 has three states, which are summarized in the figure. RUN, IDLE and SLEEP. The power consumption and transition times between states are shown. Break-even time for a state is the minimum idle required to justify entering the state. A prediction of the idle period is needed in order to determine which state to enter. Timeout is the simplest prediction, wait for a fixed amount of time and if nothing has happened enter the low power state.

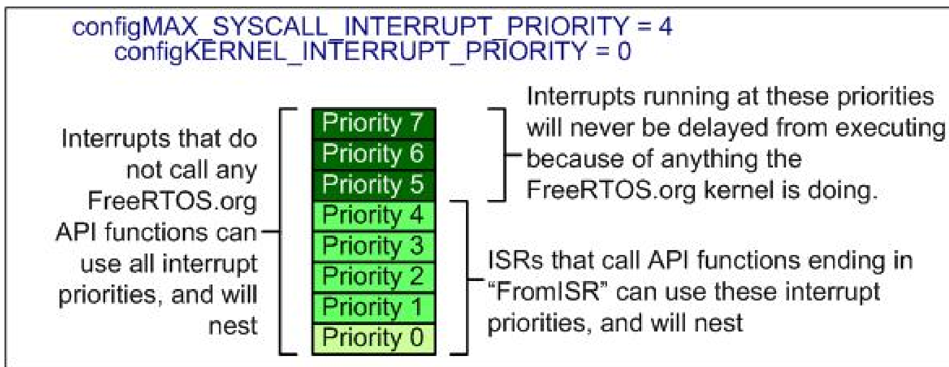

Figure 6: Figure illustrating how interrupt priorities are divided in FreeRTOS. Taken from www.FreeRTOS.com

Predictive shutdown techniques makes a decision of which state to go to as soon as the idle period starts based on previous idle and active periods.

To reduce the delay imposed when the device has to go from an idle state back to the active state, *Predictive wake up* techniques wake up the device before anything has happened. The technique proposed by [Hwang and Wu, 2000] predicts the length of the idle period as the weighted sum of the last idle period, and the last prediction. It wakes up the device when the predicted idle period has elapsed. If the idle period is lower the the break-even time, the device stays in the active state but the calculation done over again when the break-even time has elapsed. This is to avoid staying awake when a long idle period occurs.

2.4.1 Linux power management governors

The way Linux (version 2.6.34) [Lin, 2010] manages sleep states is interesting. The infrastructure used is called power management governors. There are two governors to choose from, ladder.c or menu.c. They can both found in drivers/cpuidle/governors/ in the Linux source code [Lin, 2010]. The ladder governor is the simplest one, it starts by entering the lightest sleep state. If that was successful it tries the next sleep state the next time.

The menu governor is more advanced. To estimate how long the idle period will be it uses the next scheduled event as a starting point. If timer is set to wake up the CPU in 500ms the idle period will not be longer than that. Since the idle period will rarely be that long, a correction factor which is based on previous behaviour is used. If for example all the previous idle periods was 50% of the time until the next timer event, the estimated idle period will be 50% of the time until next event. 12 independent factors used based on how long the expected idle period is, and if there is disk IO pending or not. If for example it is 50μs until the next timer event, a different factor is used than if it is 500μs until the next timer event. Another constraint used to limit the performance impact is a performance multiplier. It is used like this on line 237 in menu.c of the Linux kernel:

```
if (s->exit_latency * multiplier > data->predicted_us)
   break;
```

This means that if the exit latency of a sleep state multiplied with a multiplier is larger than the expected idle period, the state is not considered. This multiplier is based on current load of the system. The higher the load, the higher the multiplier.

Both the menu governor and the ladder governor lets device drivers register their latency constraints. For example an audio driver might know that it will get an interrupt when it has 200 μs of samples left in the DMA buffer. Then it could set a latency constraint of 150 μs. That way it can be sure it will have time enough to put more samples in the buffer before it runs out.

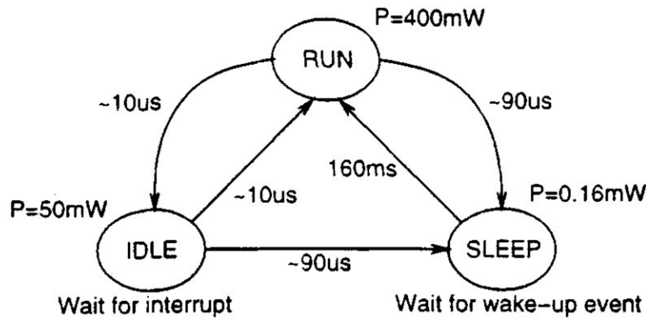

Figure 7: Power state machine for the StrongARM SA-1100 processor. Taken from [Benini et al., 2000].

2.5 Timekeeping

Traditionally Linux counted the number of timer interrupts in a variable called Jiffy. A hardware device was used to generate a periodic interrupt at a specific rate. Every time the interrupt occurs, a CPU increments the jiffy variable. This scheme has a number of shortcomings. One being that the rate of the timer interrupt limits the resolution timekeeping. This was the major concern of Srinivasan et al. [Srinivasan et al., 1998]. Other issues reported by Stultz et al. concerns correctness [J Stultz, 2005]. If a buggy driver blocks interrupts for too long, timer interrupts could be lost. Stultz et al. also argues that the code introduced in the Linux kernel to increase the resolution of timekeeping could also be buggy [J Stultz, 2005]. To increase the resolution time between ticks was interpolated by a high resolution hardware timer.

Corbert [Corbet, 2005] describes how tickless idle was first implemented on Linux. Linux had support for being tickless during idle periods on some architectures (at least on S390 and OMAP ARM) already in Kernel 2.6.6. When the CPU was idle, the hardware timer was set up to give an interrupt in time for the next event. When an interrupt woke up the CPU, the jiffy variable was updated before the interrupt handler was allowed to run. The jiffy variable was updated by calling

the do_timer_interrupt() function as many times as necessary. It is interesting to note that in kernel 2.6.13, a variable called modulo_count was used to store the reminder when updating jiffies. For example if 300 32768Hz ticks had elapsed in the idle period, the jiffy should be updated with 300 * 32768 / 1000 = 9.15. (With 1000Hz frequency on the timer interrupt). The jiffy variable (being an integer) can only be updated with 9. This way the reminder is always lost. In kernel 2.6.13 in arch/arm/mach-omap1/time.c [LiA,] this is compensated for by incrementing the modulo_count with the reminder every time, and when modulo_count gets big enough, increase the jiffy one extra time. In kernel 2.6.14 in the same file, this modulo compensation is removed because this rounding error is compensated for by the interpolation between ticks.

[Srinivasan et al., 1998] modifies the Linux kernel to use the hardware timer to interrupt the core when the next event is scheduled to occur (Rather than interrupting it periodically). This is done to achieve higher accuracy of timekeeping. Since other subsystems in the Linux kernel uses the jiffy variable and relies on it to be updated, it is kept up to date. [Gleixner and Niehaus., 2006] started as a fork of the work presented in [Srinivasan et al., 1998]. It is the code from this project that is used in the kernel today (2.6.34). The periodic tick is not used for timekeeping at all (when configured with the option CONFIG_NO_HZ). Timekeeping is not done with the periodic tick at all. There is still a periodic tick when the core is active though, but it is used for other tasks than timekeeping.

3 FreeRTOS on EFM32

3.1 IAR Project

Since FreeRTOS is already ported to Cortex-M3, getting FreeRTOS to run on EFM32 was really just a question about setting up a project in IAR Embedded Workbench, including the right files and setting up the interrupt vector table. In addition a demo application was created.

In the FreeRTOS directory there is a directory named Demo. Every vendor supported by FreeRTOS puts demo applications here. The demo includes a project that can be built without warnings or errors and run directly on the MCU from the vendor and work on a development kit. For example, STM as 4 demo applications for the STM32F103 which are in directories named: CORTEX_STM32F103_IAR, CORTEX_STM32F103_IAR, CORTEX_STM32F103_Primer_GCC and CORTEX_STM32F107_GCC_Rowley. This makes it easier for customers wanting to develop an application with FreeRTOS on the MCU. They can simply copy the project for the compiler they are using and have a working project they can start modifying.

A new directory called CORTEX_EFMG890F128_IAR was created, and inside an IAR project was configured to compile, upload, run and debug applications on the development kit from Energy Micro. In order to make it self-contained, the BSP[1] and CMSIS[2] files provided by Energy Micro was added to the directory. This makes it easier for new developers to download FreeRTOS and start developing applications on the development kit. It is however not ideal from a software development point of view since it leads to duplication of code if the CMSIS and BSP code is included inside different FreeRTOS projects. If there is a new release of either BSP or CMSIS, there would be several places to apply the updates. However, the requirement for the project in IAR to be self-contained and compilable as downloaded outweighed this consideration.

In order to set up the interrupt vector table, startup_efm32.s was copied from the CMSIS directory and modified. The modified version can be found in Appendix A.5. The interrupts changed are vPortSVCHandler, xPortPendSVHandler and xPortSysTickHandler.

3.2 Demo application

The demo application demonstrates how the LCD is used, and how the leds on the development kit can be used. This demo was added in the official FreeRTOS release 6.0.4 released March 14. 2010 [Changelog, 2010]. Richard Barry, the maintainer of FreeRTOS made some small changes to the application. The code can be found in appendix A. There is one task called LCDTask that prints out some text to the LCD display, and another that toggles leds 8-15 on the development kit. The demo application also uses a demonstration task which is included by FreeRTOS.

[1]Board Support Package is code code provided by Energy Micro to make it easy to develop applications on the development kit.

[2]Cortex Microcontroller Software Interface Standard contains name definitions, address definitions and helper functions to access registers and peripherals.

The task is called crflash and is common to demos from other vendors. It uses 8 co-routines to flash leds 1-7.

Every demo in FreeRTOS should implement the functions in partest.h. They are used to initialise and test the leds on the development kit. See appendix A.4 for ParTest.c. There is one function used to initialise the leds, which utilises the functions DVK_init and DVK_setLEDs from the BSP. There is also one function vParTestSetLED(uxLED,value) which sets led uxLed to either on or off according to value. The last function called vParTestToggleLED(uxLED) toggles led uxLED.

4 Tickless idle

In order for the EFM32 to take advantage of EM2 where the high frequency clock is turned off, a way to keep track of time with the low frequency clock is needed. It is also needed in order to stay in EM1 for longer periods than the period of the SysTick interrupt.

Several solutions has been tried. This subsection contains a short description of the ideas. The next subsection is a thorough description of the chosen solution. The main issue is keeping xTickCount up to date. When woken up by something, for example receiving something on the LEUART, the xTickCount has to be updated. One strategy is to modify FreeRTOS and make xTickCount point to a register which is updated when the core is sleeping. That way there is no need to update xTickCount when waking up, since the hardware has been updating it while the core was sleeping. The other conceptual idea is to update xTickCount according to the length of the period slept when the core is woken up. Care has to be taken to avoid that execution of the application is allowed to proceed before xTickCount is updated.

Idea #1: Cortex Systick count register as xTickCount The first idea tried out was to use the register used by the System Timer in the Cortex core. This register is used to count down to 0, and then give a systick interrupt. This register can be clocked by either the core clock (14MHz) or the last bit of the RTC counter value. The idea was to modify the xTickCount in FreeRTOS to point to this register. This was done with a #define, leading all references to xTickCount to dereference the memory location of the register. A prototype was implemented and seemed to work. A big problem however was that the register stopped counting when the MCU went to EM2, even though the RTC counter value continued to increment in EM2. The idea was discarded because of this.

Idea #2: RTC counter value as xTickCount The second idea tried is an adjustment of the first. The fact that the counter value register of the system timer in the Cortex core stops counting in EM2 led me to the idea to use the RTC counter value register instead. This has been implemented and seemed to work. The main reason for dropping this idea is the fact that it required bigger changes to the FreeRTOS code in order to work.

The code below shows how xTickCount was defined to be the value of the RTC count register (RTC-¿CNT).

```
#if configUSE_TICKLESSIDLE == 1
  #define xTickCount (portTickType)(RTC->CNT)
#else
```

The RTC CNT value is only 24 bits, which made it more complex to get FreeRTOS to handle using it as xTickCount . Figure 8 shows how the vTaskDelay() function in task.c was modified in order to get overflowing correct. Normally xTimeToWake is calculated as a normal addition. To check for overflow vTaskDelay checks if xTimeToWake is smaller then xTickCount. If that is the case, a overflow has

happened and the task is put into the overflowed delay list. If the 24 bit CNT register was to be used as xTickCount, this overflowing had to be done manually in software by using the modulo operator.

```
#if configUSE_TICKLESSIDLE == 1
    xTimeToWake = (xTickCount + xTicksToDelay)%0x1000000;
#else
    xTimeToWake = xTickCount + xTicksToDelay;
#endif
```

Figure 8: The code used in vTaskDelay to get overflow correct when using 24 bit register as xTickCount

When the RTC CNT overflows, the RTC gives an interrupt. The function called by the RTC interrupt routine can be found in figure 9. This interrupt however is asynchronous to the execution of the kernel. Meaning that there is no guarantee for what the kernel is doing when the RTC CNT overflows. The RTC CNT could overflow, without giving the kernel the opportunity to swap the delayed lists right away. In turn, this could lead to unwanted behaviour. This problem was not solved completely before abandoning this Idea.

```
#if configUSE_TICKLESSIDLE == 1
void vTaskTickoverflow(void){
    xList *pxTemp;

    /* Tick count has overflowed so we need to swap the delay lists.
    If there are any items in pxDelayedTaskList here then there is
    an error! */
    pxTemp = pxDelayedTaskList;
    pxDelayedTaskList = pxOverflowDelayedTaskList;
    pxOverflowDelayedTaskList = pxTemp;
    xNumOfOverflows++;
}
#endif
```

Figure 9: The function called by the RTC interrupt routine when the RTC CNT overflows

Idea #3: Sleeping and updating xTickCount in idle loop This is the chosen idea. Namely because it requires the least amount of changes to the FreeRTOS source code. Description follows in the rest of this section.

4.1 Difference in power consumption

The difference in power consumption between the ideas has not been investigated in depth, but is considered not to be significant enough to outweigh the advantages with the chosen solution.

A optimisation that could be applied to idea #2 is the need to wait for HFXO (if HFXO is used as core clock, and not HFRCO) to start up before proceeding. When woken up FreeRTOS could start HFXO but continue to run with HFRCO until HFXO was ready. That way the wake up time could be reduced with idea #1 and #2, which would reduce the power consumption, since the core could stay in low energy mode for longer. The order of magnitude here is milliseconds as explained in 2.2.4. This could result in a higher power consumption with idea #3 than the first two.

A drawback to this idea is the fact that the systick interrupt would for a small period occur at a different frequency than defined by the application. This could lead to unwanted behaviour. If the systick interrupt was still used as the source for periodically running the scheduler, tasks would end up getting an uneven time slice. But if this difference could be accepted, power could be saved. Even with idea #3 this method could be used if timing while active is not very crucial. Using HFRCO for timekeeping while FreeRTOS is configured to a higher frequency because it is set up to use HFXO would lead to a drift on the clock while active. This would be corrected by the scheme presented in section 4.6.2.

4.2 Chosen solution

The idle loop is scheduled to run by the scheduler every time there is no tasks with higher priority than the idle task ready to run. This makes the idle loop a good place to put the core to sleep, and wake it up again in time for the next task to run. The matter is complicated a bit by the fact that the co routines are scheduled by the idle loop making a call to vCoRoutineSchedule();

Figure 10 shows a sequence diagram for the common case when FreeRTOS puts the core to sleep. The idle task calls a function called sleepWhileIdle() which is implemented in in energymodes.c. sleepWhileIdle disables interrupts and calls xTaskGetTickCount() to get the number of ticks elapsed. Then it continues to call xCoRoutineNextTick(int) and xTaskNextTick() to determine the next event scheduled to happen. The RTC is then set up to wake up the core before the next event. When an interrupt is received, either from the RTC or something else, sleepWhileIdle() determines how long the core was sleeping by looking at the counter value in the RTC and returns the value. The idle task then updates the xTickCount value and enables interrupts. It is important that the xTickCount is updated before the interrupt is processed since the interrupt may wake up a task, and if a preemptive scheduler is used it may cause a task to run while xTickCount still has an old value for xTickCount .

4.3 Interrupt handling

It is considered imperative that the xTickCount variable is updated before interrupts are allowed to be processed. If an interrupt is allowed to be processed before xTickCount is updated to the correct value, a task might be woken up by the interrupt and start running while the xTickCount has an old value. This could

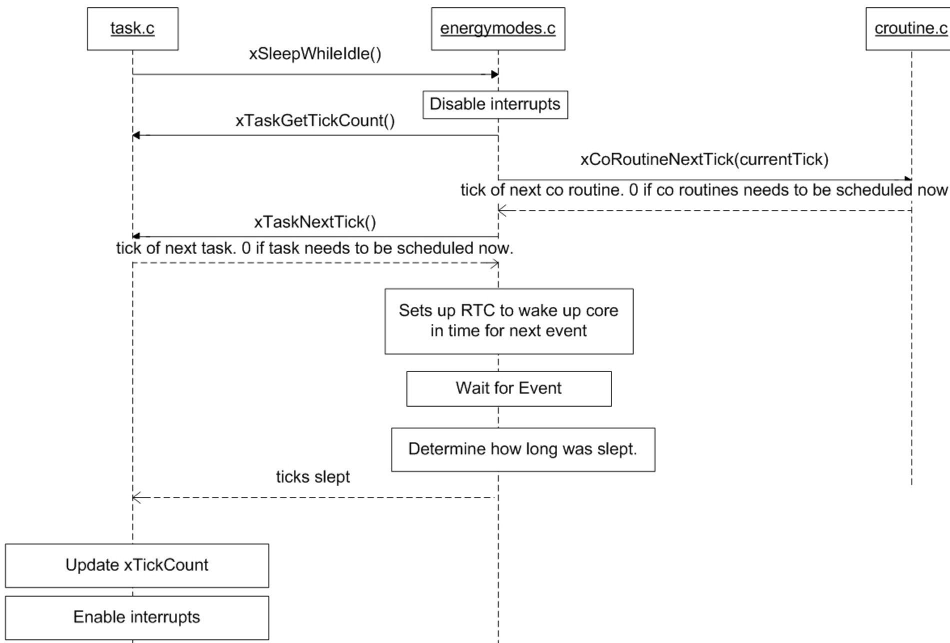

Figure 10: Sequence diagram showing interactions between files in my solution.

lead to unwanted behaviour, if for example the task calls vTaskDelay() while the xTickCount is old.

To make sure the xTickCount variable is updated before processing any interrupts, interrupts are disabled while sleeping. This is done by using FreeRTOS macro called portDISABLE_INTERRUPTS(). It disables all interrupts with priority lower than configMAX_SYSCALL_INTERRUPT_PRIORITY (see Section 2.3.4). Disabling all interrupts and then going to sleep would usually make the core sleep forever. However, if the SEVONPEND bit in the System Control Register is set the core is woken up from a WFE instruction when an interrupt goes to the pending state, even if it does not have a high enough priority to wake up the core.

If an interrupt has a higher priority than configMAX_SYSCALL_INTERRUPT_PRIORITY the interrupt service routine will get processed right away, without updating the xTickCount . Those interrupts are not allowed by to call any FreeRTOS API functions. This scheme ensures that those interrupts will not be delayed by code in this project. If the core is sleeping when such an interrupt occurs, the delay to start up the core (around 2 μs) will be encountered. The interrupt will in this case be processed with HFRCO even if HFXO is used by the application.

4.4 Time until next event

In order to determine the time until next event two functions has been added to FreeRTOS. xTaskNextTick() in task.c and xCoRoutineNextTick(currentTick). See

Figure 11 and Figure 12 for code. They both return the next tick count on which a task or co routine is ready to run. They both return 0 if execution of a task or co routine needs to be performed, and sleeping is not possible.

```
1  #if configUSE_TICKLESSIDLE == 1
2     portTickType xTaskNextTick(void){
3           if(uxTopReadyPriority>0
4              || xPendingReadyList.uxNumberOfItems>0
5              || pxReadyTasksLists[0].uxNumberOfItems>1){
6              return 0;
7           }
8           if(pxDelayedTaskList->uxNumberOfItems>0){
9              return (pxDelayedTaskList->xListEnd.pxNext->xItemValue);
10          }else{
11             return portMAX_DELAY;
12          }
13    }
14 #endif
```

Figure 11: xTaskNextTick function in task.c

Figure 11 shows the xTaskNextTick() function. In line 3 a check is made to see if there are any tasks ready to run right now. Since this is all done from the idle task, one should think that no tasks could be ready run when the idle task is running. However, tasks could be sharing the idle priority. That way tasks with the same priority as the idle task could be ready to run even if the idle task is running. Even if no tasks are sharing the idle priority, higher priority tasks could have been woken up by interrupts before interrupts where disabled in the idle task. This is only possible when a cooperative kernel is in use. Because if the kernel is preemptive, a task becoming ready to run would preempt the idle task and start running right away. If the kernel is cooperative the task that becomes ready would not be able to run before the idle task yields on its own. There could be a task in one of the xReadyTasksLists. If uxTopReadyPriority is set to something above 0, it means that a task is ready to run in one of the readyTask lists. If however it is 0, a task could still be ready to run at priority 0, which is why the *pxReadyTaksLists[0]* is checked. If any tasks are in the xPendingReadyList sleep mode can not be entered either.

If there are no tasks ready to run at this time, the next task can be found in the pxDelayedTaskList. If there are no tasks here, either no tasks have been created at all, or they are all in the pxOverflowedTaskList. Either way, *portMAX_DELAY* is returned. Meaning the core will wake up in time to handle the *xTickCount* overflow.

Figure 13 shows xCoRoutineNextTick(portTickType currentTick), it takes in the current tick count and is called when interrupts are disabled. croutine.c maintains its one tick count variable *xCoRoutineTickCount*, and it is not always up to date with the "original" *xTickCount* in task.c. This is why *xCoRoutineNextTick* differs from xTaskNextTick in that it returns 0 if currentTick has overflowed and *xCoRoutineTickCount* has not overflowed yet. If that is the case sleep mode will

```
1        #if configUSE_TICKLESSIDLE == 1
2           portTickType xCoRoutineNextTick(portTickType currentTick){
3           if(uxTopCoRoutineReadyPriority>0 ||xPendingReadyCoRoutineList
                .uxNumberOfItems>0 || pxReadyCoRoutineLists[0].
                uxNumberOfItems>0 ){
4              //there are co routines ready to run.
5              return 0;
6           }
7           if(xCoRoutineTickCount>currentTick){
8              //this means there has been an overflow on current tick,
                   which needs to updated.
9              return 0;
10          }
11          if(pxDelayedCoRoutineList != NULL && pxDelayedCoRoutineList->
                uxNumberOfItems>0){
12             return (pxDelayedCoRoutineList->xListEnd.pxNext->
                   xItemValue);
13          }else{
14            return portMAX_DELAY;
15          }
16       }
17       #endif
```

Figure 12: xCoRoutineNextTick function in croutine.c

not be entered now, but rather have FreeRTOS update the *xCoRoutineTickCount*. The only way *xCoRoutineTickCount* can be bigger is if *currentTick* overflowed while *xCoRoutineTickCount* did not.

4.5 Calculating how long to sleep

```
if((ticksUntillNextEvent-1)>=((0xFFFFFFFF)/RTCTICKFREQ))
{
  ticksUntillNextEvent=((0xFFFFFFFF)/RTCTICKFREQ);
}
unsigned int rtcWakeUpVal=(((ticksUntillNextEvent-1)*RTCTICKFREQ
    )/configTICK\_RATE\_HZ)+rtcCountBefore;
```

When calculating how many RTC cycles to sleep, the number of ticks until next event is multiplied with the frequency of the RTC (which is 32768 if no prescaling is used). This is divided by the frequency of the systick (configTICK_RATE_HZ). The current value of the RTC CNT register is added to the result.

Since ticksUntillNextEvent can be 32bits, multiplying it with 32768 might overflow. To guard against this a check is performed in advance. If the multiplication would overflow, tickUntillNextEvent is set so that it will not. This means that the maximum number of ticks that can be slept is $0xFFFFFFFF/32768 = 131071$. Meaning that if it is above 131071 ticks until next event, 131071 ticks will be tried to be slept. This is equal to 1310 seconds if 100Hz SysTick is used. The RTC would overflow in 512 seconds running at 32768Hz anyway, which means that this way of calculating how many RTC ticks to sleep is not the limitation in a normal

```
#if configUSE_TICKLESSIDLE == 1
   portTickType xCoRoutineNextTick(portTickType currentTick){
   if(uxTopCoRoutineReadyPriority>0 ||xPendingReadyCoRoutineList.
       uxNumberOfItems>0 || pxReadyCoRoutineLists[0].uxNumberOfItems>0
        ){
      return 0;
   }
   if(xCoRoutineTickCount>currentTick){
      return 0;
   }
   if(pxDelayedCoRoutineList != NULL && pxDelayedCoRoutineList->
       uxNumberOfItems>0){
      return (pxDelayedCoRoutineList->xListEnd.pxNext->xItemValue);
   }else{
     return portMAX_DELAY;
   }
}
#endif
```

Figure 13: xCoRoutineNextTick in croutine.c

use scenario.

The alternative would be to multiply *ticksUntillNextEvent* with RTCTICKFREQ/configTICK_RATE_HZ directly. This way ticksUntillNextEvent could be be bigger. However the RTC frequency would have to be evenly divisible by the tick frequency used by FreeRTOS. Essentially this would mean having the FreeRTOS systick as a power of 2. This disadvantage was to great. A very normal systick frequency are 100Hz or 1000Hz. Using this alternative they would have to be 128Hz or 1024Hz.

4.6 Calculating how long was slept

When the core is woken up, it is necessary to calculate how long it was sleeping. This is done by storing the RTC CNT value in a variable before going to sleep and comparing it to the RTC value when the core is woken up.

```
if(RTC->IF & RTC_IF_OF){
   rtcCounterValue = RTC->CNT;
   rtcTicksElapsed = (rtcCounterValue+0xFFFFFF-rtcCountBefore);
}else{
   rtcTicksElapsed=(rtcCounterValue - rtcCountBefore);
}

xTickCountIncrement=rtcTicksElapsed*configTICK_RATE_HZ/
    RTCTICKFREQ;
```

The code above shows how the number of RTC cycles the core was sleeping is calculated. If an overflow on the RTC CNT value has happened, 0xFFFFFF, has to be added in order to get it correct. In the next line RTC cycles is translated into xTickCount cycles. Because of the way the RTC is set up, this calculation can not overflow. Another issue is that the division when translating to xTickCount almost always result in truncation of bits, leading to a large rounding error. If for

example 50'000 RTC cycles has elapsed while sleeping. This division will result in $50000 * 1000/32768 = 1525.9$. Since integer arithmetic is used, the reminder will be truncated away, with 0.9 xTickCount cycles being lost.

Another problem which could lead to time drifting is illustrated in figure 14. When reading the RTC CNT value there is an error margin. The value could be read anywhere in the gray area. This means that in the worst case, the error could be up to 1 period of the RTC clock. With the RTC running at 32768Hz, this is an error of around 30μs. On average however one could suspect that the read operation is performed in the middle of the clock period, e.g. when the clock is falling. If this was true, the timing errors would even them self out. One problem however is that when the interrupts which wakes the core up is synchronous to the RTC clock, the RTC will be read at the same place every time. This could lead to errors being accumulated.

Section B.2 shows a task used when developing the tickless while idle feature. It is uses the LETIMER to keep track of time unaffected by FreeRTOS. Periodically the task examines the FreeRTOS systick, and compares it with the value from the LETIMER. The difference is sent over the LEUART.

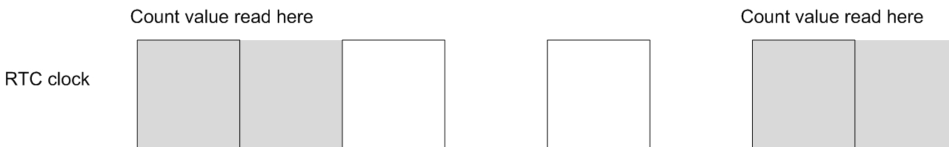

Figure 14: The RTC CNT value could be read anywhere in the grey area, possibly resulting in error.

4.6.1 Storing reminder

One way to solve the first problem in the previous subsection is to store the reminder in a variable called rounding.

```
rounding+=(rtcTicksElapsed*configTICK_RATE_HZ) % RTCTICKFREQ;
```

In the following lines of code, a check is performed to see if there is room for incrementing xTickCountIncrement by one more than the time slept this time actually dictates. It is unwanted to increase the xTickCount by any more than until one cycle before the next event is scheduled to happen. Since this might result in undefined behaviour in FreeRTOS.

```
if(rounding>=RTCTICKFREQ){
  if(xTickCountIncrement+1<ticksUntillNextEvent){
   xTickCountIncrement+=1;
   rounding-=RTCTICKFREQ;
  }
}
```

When the rounding variable is larger than the frequency of the RTC, the xTick-Count is rounded one up. This is very similar to what Linux did in 2.6.13, (see section 2.5).

4.6.2 xTickCount following RTC

To compensate for the error introduced by truncating the reminder, and the uncertainty in illustrated in 14 a different scheme was developed. The idea is to keep track of the time using the RTC, and make sure xTickCount is kept in sync with the RTC. To keep track of time with the RTC, the number of overflows is stored in a variable. When compensating for rounding errors, the time derived from the RTC is compared to the time derived from xTickCount, if they differ, xTickCount is rounded up by one.

The code below shows how this is done. First, the tick from both the RTC and from FreeRTOS (xTickCount in task.c) is calculated. Currently 64 bits is used to avoid problems with overflow. Using a 64 bit counter at 32768kHz, which is the worst case, the counter will overflow in over 17 million years. If the tick from the RTC is higher than the one in xTickCount, xTickCountIncrement is increased by one. xTickCountIncrement is the value returned to task.c, and is the value xTickCount is incremented by. There is also a check to make sure xTickCount is not incremented to past the next event scheduled to happen in FreeRTOS.

The reason this works is that in the calculation of xTickCountIncrement a truncating division is performed. This way the time slept is always rounded down, making xTickCount run slower than the RTC. When the xTickCount gets behind the tick count from the RTC, xTickCount is rounded up by one.

```
uint64_t currentTickFromRTC = ((uint64_t) rtcOverflows<<24 |
    rtcCounterValue)* (uint64_t)configTICK_RATE_HZ/RTCTICKFREQ;
uint64_t currentTickInXTickCount = (uint64_t)((uint64_t)
    ltaskGetNumberOfOverflows()<<32|currentTick)+
    xTickCountIncrement;
if( currentTickFromRTC>currentTickInXTickCount){
   if(xTickCountIncrement+1<ticksUntillNextEvent){
      xTickCountIncrement+=1;
   }
}else if( currentTickFromRTC<currentTickInXTickCount){
    if(xTickCountIncrement>0){
      xTickCountIncrement--;

    }
}
```

4.7 Consequences for interrupt latency

The interrupt latency is impacted by my changes to the kernel. In the worst case an interrupt occurring right after interrupts are disabled, would have to wait until the entire xSleepWhileIdle() function executes. It would skip sleeping since the core would terminate the WFE instruction right away because of the pending interrupt. But still, all the instructions in the xSleepWhileIdle would have to be executed. In non optimised code (meaning compiler optimisations turned off), this was measured to around 300 CPU cycles. This corresponds to around 10μs, and should be acceptable. If a lower latency is required by the application, an interrupt priority above configMAX_SYSCALL_INTERRUPT_PRIORITY could be used. Interrupt

latency with such a priority would be unaffected by the FreeRTOS kernel. They can not call any FreeRTOS API functions.

```
static portTASK_FUNCTION( prvIdleTask, pvParameters )
{
   /* Stop warnings. */
   ( void ) pvParameters;
   for( ;; )
   {
      prvCheckTasksWaitingTermination();

      #if ( configUSE_PREEMPTION == 0 )
      {
         taskYIELD();
      }
      #endif

      #if ( ( configUSE_PREEMPTION == 1 ) && ( configIDLE_SHOULD_YIELD
           == 1 ) )
      {
         if( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[
             tskIDLE_PRIORITY ] ) ) > ( unsigned portBASE_TYPE ) 1 )
         {
            taskYIELD();
         }
      }
      #endif

      #if ( configUSE_IDLE_HOOK == 1 )
      {
         extern void vApplicationIdleHook( void );
         vApplicationIdleHook();
      }
      #endif
      #if (configUSE_TICKLESSIDLE == 1)
         extern portTickType sleepWhileIdle();
         portTickType tickSlept=xSleepWhileIdle();
         xTickCount += tickSlept;
         portENABLE_INTERRUPTS();
      #endif
   }
}
```

Figure 15: idle task, I added the part after configUSE_TICKLESSIDLE

5 Managing EFM32 energy modes

This section handles the issue of which energy mode FreeRTOS should put the EFM32 in when all the tasks are delayed. Two approaches has been implemented, and both works well. Which one is best is really more a question about which one is most convenient for the programmer of an application or device driver.

5.1 #1 Clocks enabled

The first solution is for the xSleepWhileIdle() function to look at which clock is enabled in the CMU register. The CMU has two registers which needs to be examined, HFCORECLKEN0 and HFPERCLKEN0. Figure 24 and 25 in appendix C shows the bit assignments of these register. The idea of this implementation is to look at these registers in order to determine which energy mode to go to. A peripheral module can not possibly be in use if it is clock is disabled in these registers. If all high frequency clocks are turned of in the CMU, then it is safe to go to EM2, where the high frequency clock is turned off. In the HFCORECLKEN0 register the only peripheral that needs to be checked is AES. The other modules need not be checked, since the DMA will keep the core awake automatically if it is in use. The PRS can only be accessed by the core or DMA. In the HFPERCLKEN0 register GPIO clock bit can be disregarded. Even if the GPIO clock is enabled energy mode 2 can still be entered.

The implementation of this becomes very simple.

```
if((CMU->HFCORECLKEN0 & CMU_HFCORECLKEN0_AES) || (CMU->HFPERCLKEN0
    & ~CMU_HFPERCLKEN0_GPIO)){
   //Go to EM1
}else{
   //Go to EM2
}
```

5.2 #2 Explicit control

This scheme is to have tasks tell which energy mode they can go to. If one tasks needs to stay in EM1, it will issue a function call vTaskCanGoToEM(1); Then FreeRTOS will not go to an energy mode lower than 1 until the task calls vTaskCanGoToEM(int) with a parameter with a lower value than 1. If one task demands to stay in a particular energy mode, FreeRTOS will not go to a lower energy mode before the tasks calls vTaskCanGoToEM again. This scheme is easy to expand to also include energy mode 0 and 3. If a task needs to, it can call vTaskCanGoToEM(0) and FreeRTOS will not go to any sleep state until the task lets it. If all tasks agree to to EM3 (they all called vTaskCanGoToEM(3), FreeRTOS can put the core in EM3.

Several ways of implementing this was considered. One idea was to have a variable for each energy mode, with one bit for each task indicating that the task needs to stay in the respective energy mode. A difficulty with this mode was to map tasks to position in array, since no obvious way of mapping a task to a bit

in the array exists in FreeRTOS. They do not have their own numerical ID for example. Another problem is that the number of tasks changes dynamically. The number of bits needed in the variables would not be known at compile time.

Another approach explored was to store which energy mode a task can go to in the Task Control Block (TCB) of the Task. A traversal through all the tasks would be needed in order to determine which energy mode to go to. As FreeRTOS has several lists where tasks could be located, the complexity of this approach, both in run time and in coding complexity was considered to high.

A simpler, yet just as powerful approach was conceived. In the TCB of every task, it is stored which energy mode the task needs the core to stay in. The calculation of which energy mode FreeRTOS can go to if facilitated by keeping the count of how many tasks can go to which energy mode in an array. An array which contains an integer for every energy mode is kept updated with how many tasks can enter the respective energy mode. For example, pucTaksAbleToGoToEM[0] contains the number of tasks that needs the core to stay in EM0.

```
#if configUSE_TICKLESSIDLE == 1
   PRIVILEGED_DATA static unsigned char pucTaksAbleToGoToEM[
       configENERGYMODES];
#endif
```

When a task is created, it is initialized to stay in a value which can be set by the programmer in FreeRTOSConfig.h. This is shown below. The function prvInitialiseTCBVariables() is used by FreeRTOS every time a new task is created.

```
static void prvInitialiseTCBVariables(.....){
   (....)
   #if (configUSE_TICKLESSIDLE == 1)
      pxTCB->ucEmAbleToGoTo = confgiDEFAULT_EM_FOR_NEW_TASK;
      pucTaksAbleToGoToEM[confgiDEFAULT_EM_FOR_NEW_TASK]++;
   #endif
```

The code below shows how pucTaksAbleToGoToEM is handled when a task is deleted. One less task needs to stay in the energy mode found in the tasks TCB.

```
void vTaskDelete( xTaskHandle pxTaskToDelete )
{
   (...)
   #if configUSE_TICKLESSIDLE == 1
      pucTaksAbleToGoToEM[(pxTCB->ucEmAbleToGoTo)]--;
   #endif
```

The function below is called when a task wants to tell FreeRTOS that it needs to stay in an energy mode. For example if it calls vTaskCanGoToEM(1), the core will not go to a deeper sleep than EM1, until the core again calls vTaskCanGoToEM() with a higher value than 1.

```
vTaskCanGoToEM(unsigned char em){
   vPortEnterCritical();
   pucTaksAbleToGoToEM[pxCurrentTCB->ucEmAbleToGoTo]--;
   pucTaksAbleToGoToEM[em]++;
   pxCurrentTCB->ucEmAbleToGoTo = em;
   vPortExitCritical();
}
```

The function bellow is called by xSleepWhileIdle() to figure out which energy mode to go to. Simply iterate through the pucTaksAbleToGoToEM array, until a non zero value is found, and return that number. If no value is found,

```
unsigned char ucTaskEmAllowed(unsigned char em){
    for(int i = 0; i < configENERGYMODES;i++){
        if(pucTaksAbleToGoToEM[i] > 0){
            return i;
        }
    }
    return configENERGYMODES; //The current implementation will
        never reach here. Because every task will be located in one
        of the fields in the array.
}
```

5.3 Chosen solution

Solution #2 gives more flexibility. It is quite generic, as what it really does is simply return the highest number (energy mode) among all tasks. It was easy to also implement a way for the application to stay in EM0. Support for EM3 also comes with solution #2. #2 is also easier for the programmer to get correct behaviour. This has been experienced while developing driver. It is awkward and little intuitive to remember to turn of the clocks before delaying a task. The disadvantage of #2 over #1 is difficult to spot. The run time cost is negligible. The only disadvantage is that it needs changes to the FreeRTOS kernel.

6 Methodology

6.1 Testing

One test used is based on having the LETIMER keep track of time on its own, unaffected by FreeRTOS. Then the time reported by FreeRTOS can be compared to the time reported by LETIMER. See appendix B.2. Since the LETIMER is only 16 bits, it overflows quite often. If 32768Hz is used, it would overflow every 2 seconds. To reduce overflows 1024Hz is used. Overflow happens every 64 seconds.

The time reported by FreeRTOS is plotted against the time reported by the LETIMER. If the difference is more than what can be expected due to normal variation something is wrong. Example of normal variation is variation due to the issue described in 4.6.

6.2 Benchmark

Figure 16 visualizes the benchmark used. 3 tasks are used. One task receives on RS232 simulating receiving GPS information. It does not do anything with the data, just receive it. One task uses the ADC and the light sensor on the development kit and measure the light level. After doing 10 conversions, it does some calculations on them, then sends data over the RS232 bus. This is to simulate sending them over GSM. Another task uses the I2C bus to get the temperature from a temperature sensor on the development kit. It sends the temperature over the RS232 bus on every temperature reading.

Task name	Description	Frequency	Bytes transmitted/received
vTaskRxLeuart	GPS(RX LEUART)	1Hz	65 Bytes Rx
vMeasurementLight	Measure Light(ADC)	1Hz	0
vMeasurementLight	Calculate and send light data	0.1Hz	125 Bytes Tx
vMeasurementTemp	Measure Temp(I2C) and TX it on LEUART	1Hz	157 Bytes Tx

Table 2: Sub tasks performed by the benchmark application

Figure 2 shows at which rate the sub tasks of the benchmark application is run. The task named vTaskRxLeuart performs a receive on the LEUART once every second, receiving 65 bytes. vMeasurementLight performs two sub tasks. It measures the the light level and stores it in an array once every second. Every ten seconds it calculates and sends light data over RS232.

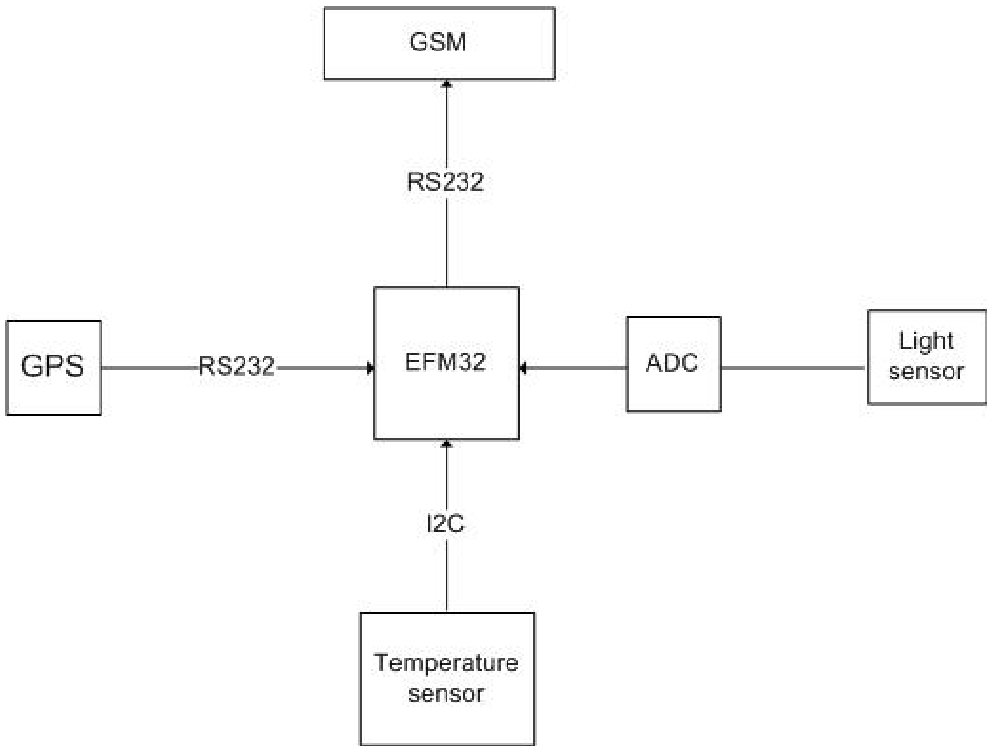

Figure 16: Benchmark application

6.2.1 Effect of load

How the load affects the power saving of my solution was measured. A task was used, which is awake for given time and then sleeps for a given amount of time. This task was run with the lowest priority along with all the other tasks in the application. The task has a period of 400 ms. The tests are shown in Table 3. For example in the 3rd test, the task does work for 40ms then it sleeps for 360ms. Giving 10% load time and 90% idle time.

Test	Load per cent	Idle per cent
1	0(0ms)	100 (400ms)
2	1(4ms)	99 (396 ms)
3	10 (40ms)	90 (360 ms)
4	25 (100ms)	75 (300 ms)
5	50 (200ms)	50 (200 ms)
6	75 (400ms)	25 (100 ms)
7	100 (400ms)	0 (0 ms)

Table 3: Distribution of idle and load time for task.

6.3 Versions of FreeRTOS tested

Vanilla is the standard FreeRTOS which now is included in the standard FreeRTOS source. *Simple EM1* is a very simple extension of FreeRTOS taking advantage of EM1. The IDLE task enters EM1, keeping the SysTick active and waking up to increment xTickCount. *EM2 No DMA* is my extension to FreeRTOS described in section 4 with a LEUART driver not utilizing the DMA.

6.3.1 Versions with DMA driver

EM2 With DMA is the same as *EM2 No DMA*, but with a LEUART driver which uses the DMA while transmitting and receiving. *EM2 With DMA Bug* is the same as *EM2 With DMA* but with a very small difference. It uses a different signal for the LEUART to request a new character for transmission. *EM2 With DMA* uses TXBL, which means that the LEUART will request a new character as soon as it has room in its transmit buffer. *EM2 With DMA Bug* uses TXEMPTY, which means that the LEUART will request a new character when the transmit buffer is empty.

The EFM32 should be able to stay in EM2 when using DMA to receive on LEUART. But because of what is suspected to be an errata to the chip at hand, the RXDATAV interrupt has to be turned on in the LEUART. With the result that the core is woken up from sleep when a byte is received, even if it is copied to memory with DMA. An errata for this is not listed in the errata history of EFM32 [EFM32 Errata,]. A similar errata for transmitting is listed. A bug report has been submitted to my supervisor at Energy Micro. My EFM32 chip is an early engineering example.

6.4 Power measurement

The Dev kit has a built in way to measure the power consumption of the MCU. It is called AEM(Advanced Energy Monitor), and samples the current consumption and the voltage at a rate of 60Hz when the current is bellow 200μA and at 120Hz when the current is above. A rate of 120Hz means one sample every 8.3ms.[Dev kit Manual,].

An application (energyAware Profiler version 0.92) developed by my supervisor at Energy Micro was used to get these values in CSV (comma separated values) format. These CSV files where then processed in order to plot them with gnuplot. The python script used for this can be found in B.8.3. energyAware Profiler was also has a feature to give the average current the application has used over a time interval. This feature was used to calculate the expected battery lifetime of the different applications.

7 Results

7.1 Testing

Figure 17 shows the result of the timing test on two versions of the code. The difference between the time in the FreeRTOS xTickCount variable, and the one in Low Energy Timer. Clocked by low frequency clocks. Remains active in EM2 (LETIMER) is plotted. The error was 1111ms after the application went to sleep and woke up to update the xTickCount variable 6350 times. In the worst case there should only be 30μs error each time a sleep mode is entered (see section 4.6.1). The error should, in worst case, only be roughly $6350 * 30\mu s = 194$ms. Something was clearly wrong. The error was only observed when using HFXO as the core clock, and not while using HFRCO. The main difference between the two is start up time. HFXO has a start up time of 400μs, while HFRCO has a start up time of only 0.6μs. See section 2.2.4. The bug was due to the fact that when waking up from EM2, the routine has to wait for 1.1ms for HFXO to become ready. This resulted in that there was never room to counter act the rounding error introduced by integer arithmetic. See section 4.6.

Figure 18, shows the same application as in figure 17, but with only for the correct application, and with a different scale. After 6336 sleep periods, the difference is 41ms. This small enough to be within the concerns expressed in 4.6.1.

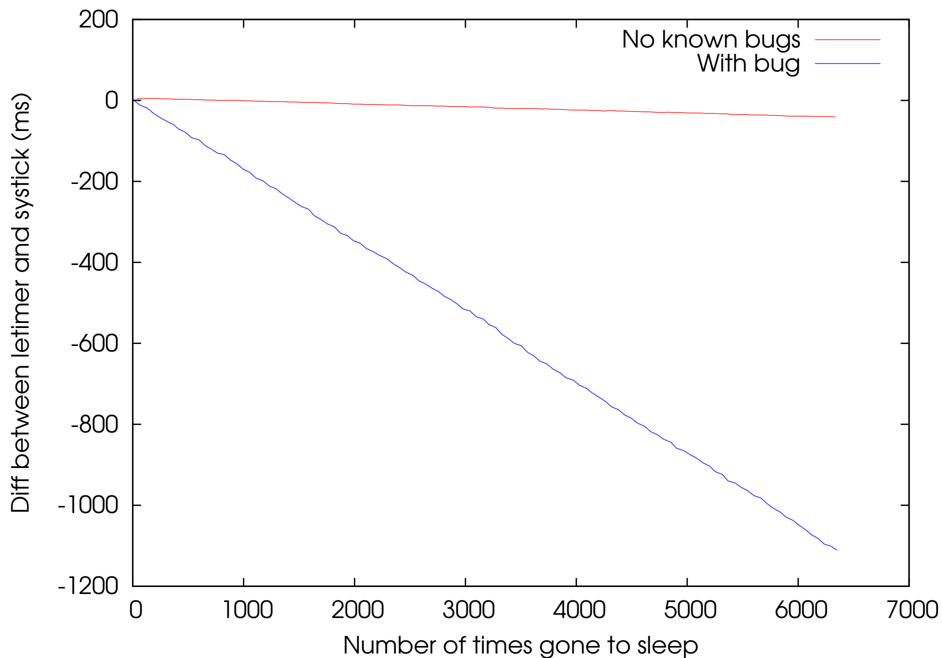

Figure 17: X-axis is the number of time the application enters EM2. Y-axis is the difference between the time in the systick variable, and the time from LETIMER

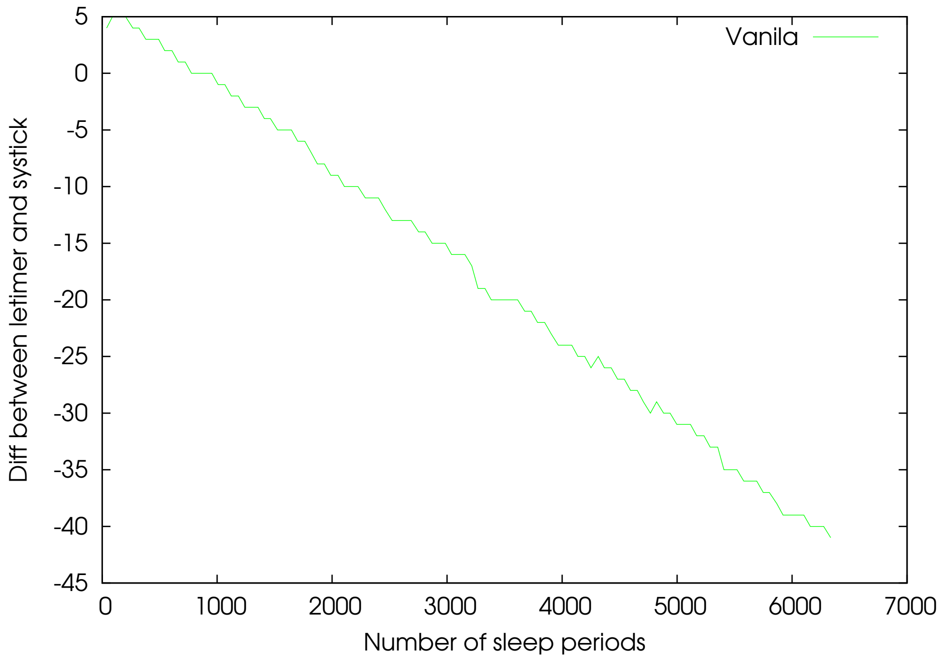

Figure 18: X-axis is the number of time the application enters EM2. Y-axis is the difference between the time in the systick variable, and the time from LETIMER

"No known bugs" in figure 19 shows the drift when the rounding scheme in 4.6.1 is applied. "No drift" is when the scheme in 4.6.2 is used. As can be seen, the clock drift has been eliminated with the later scheme.

7.2 Power consumption

Figure 20 shows power consumption of the application in 6.2 running on different versions of FreeRTOS. The lines represent the power consumption of different versions of FreeRTOS. The different versions are described in section 6.3. The Y axis is the power consumption in mW on a logarithmic scale. The X axis is time in ms. The snapshot is 9 seconds into the application.

The spikes marked "1" is when the application is measuring and transmitting the the temperature over LEUART. The spikes marked "2" is when the application is receiving data from the "GPS". The spikes marked "3" is when the application is measuring the light level using the ADC. The spike marked "4" is when the application does the calculations on the measured light values and then transmits them over the LEUART.

It is obvious that utilising the EM2 enabled version of FreeRTOS saves a lot of power. The power consumption while idle is only 0.02mW when EM2 is used compared to 13mW when the core stays in EM0. In the version where EM1 is used, 6mW is consumed.

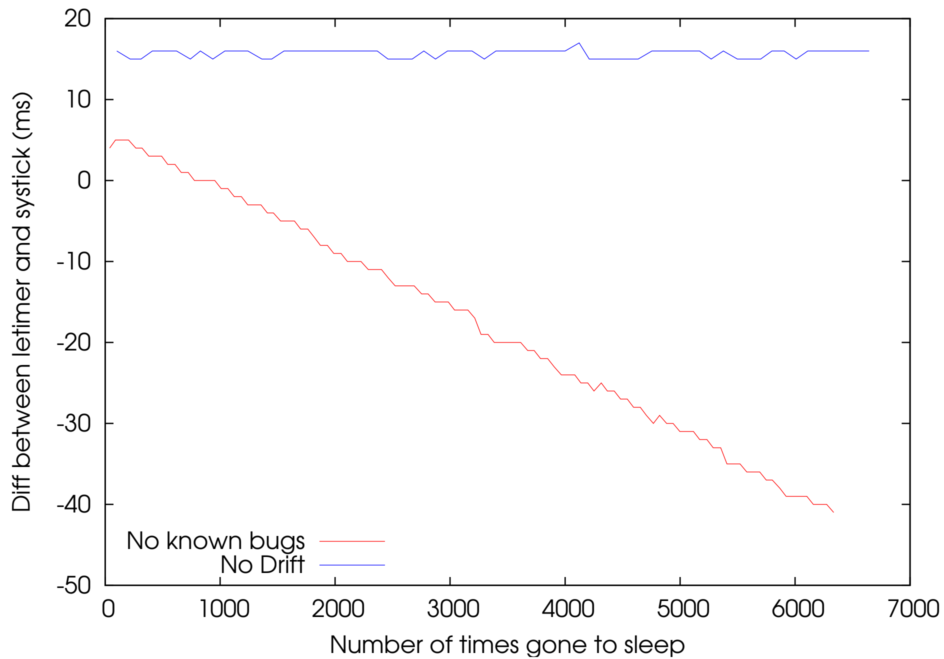

Figure 19: X-axis is the number of time the application enters EM2. Y-axis is the difference between the time in the systick variable, and the time from LETIMER

More observations can be made in the graph is that DMA saves power. This can be seen in the spikes marked "1". Using DMA EFM32 can enter EM2 while transmitting the temperature. Using DMA for receiving does not save as much power as one should expect.

The spikes marked "2" shows that when the application is receiving on the LEUART the version with DMA uses only slightly less power than the version without DMA. The DMA should use significantly lower power, but because of the issue explained in 6.3.1, the core is woken up from EM2 every time a byte is received.

The spikes marked "3" are roughly the same both with and without DMA. This is expected because no transmission or receiving performed. The only thing done at this spike is a measurement of the light level.

The spike marked "4" is a longer more complex calculation on the values collected at spikes "3". As can be seen on the graph both with and without DMA uses roughly the same power when calculating. When the data is transmitted over the LEUART at the end of the spike, the DMA version uses less power.

Figure 21 is almost the same as figure 20 *EM2 With DMA Bug* (described in 6.3.1) is added to the plot. As can be seen in the graph, the bug version of the DMA driver uses more power when transmitting. This is due to the fact that it is woken up every time the DMA transfers a byte to the LEUART TX register.

Figure 22 shows the expected battery lifetime of the different versions of the application. As expected the version which utilises EM2 will last a lot longer with a standard 220mAh battery. The Vanilla version lasts 56 hours, EM1 lasts roughly twice as long, 118 hours. While the version which utilises EM2, lasts for 883 hours. The benefit of also utilising DMA with the EM2 version is also significant. The DMA version lasts roughly twice as long, 1867 hours. As expected the buggy DMA version makes things worse and lasts shorter the the version without DMA, 768 hours.

Figure 23 shows the result of the tests described in 6.2.1. As expected the benefit of taking advantage of EM2 decreases as the load increases. Already at 1 per cent load time time, the batter lifetime decreased to 69% of what it was. From 2040 hours to 1417 hours.

Figure 20: Power consumption with different version of FreeRTOS

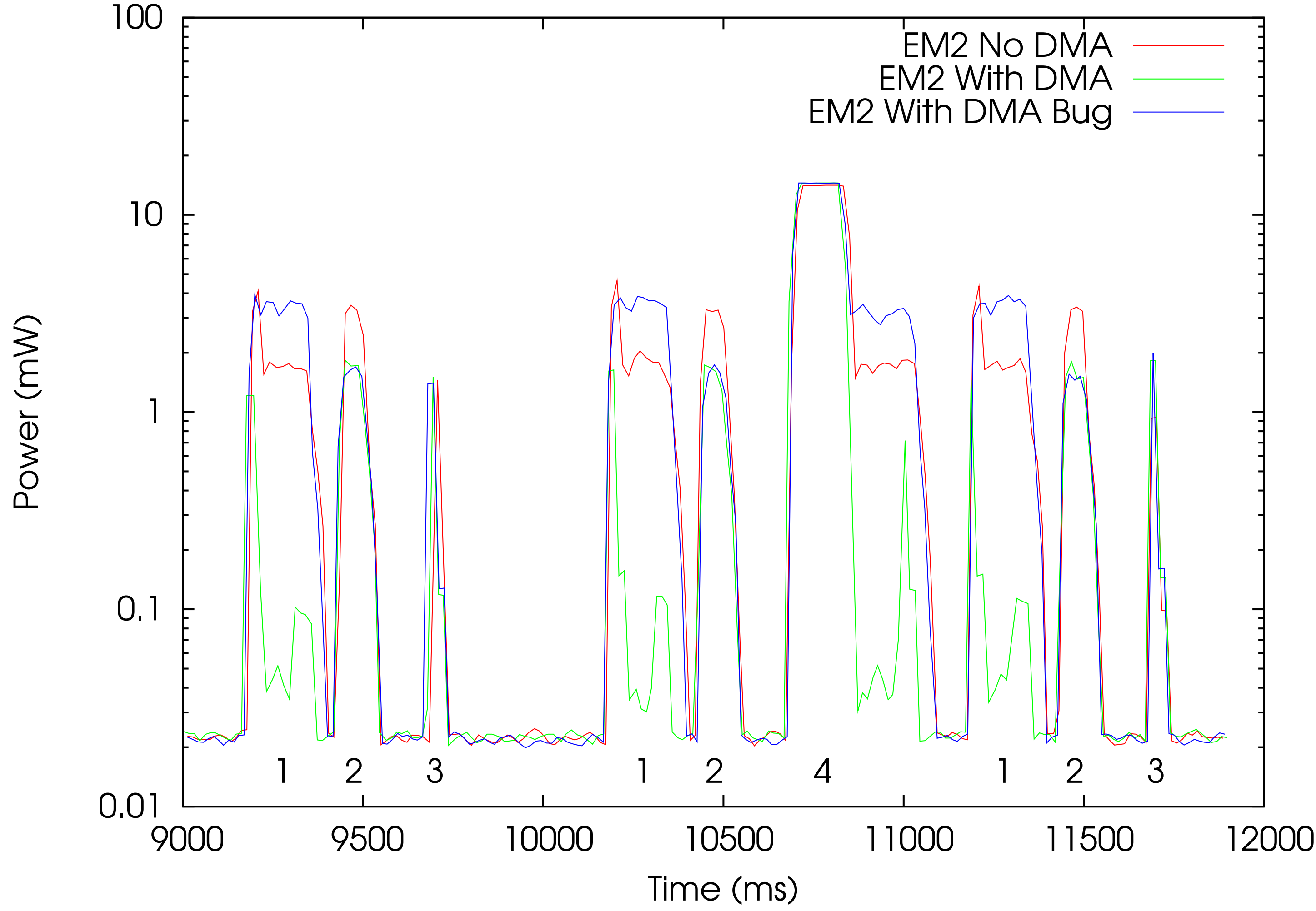

Figure 21: Power consumption with different versions of FreeRTOS

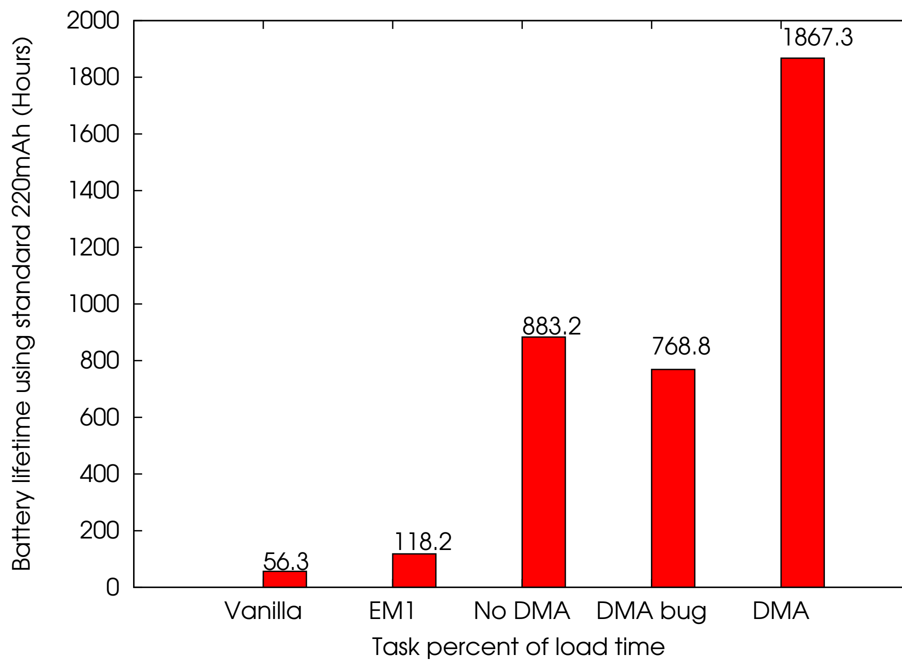

Figure 22: Expected battery life time for applications using a standard 220mAh battery

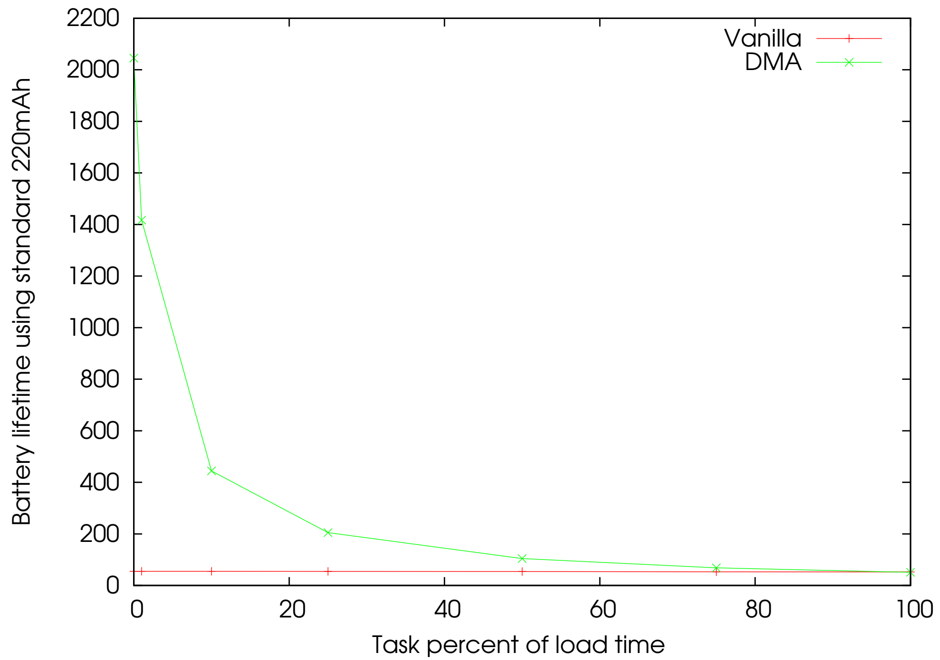

Figure 23: Expected battery life time for applications using a standard 220mAh battery. X axis is per cent of ticks the additional calculation task keeps the core awake.

8 Conclusion

In this work FreeRTOS has been modified to stay idle for long periods while there are no tasks ready to run. Avoiding the time to drift has been a major issue. In the end the chosen solution has been to have time follow the low frequency oscillator. This is considered the best option since this is the oscillator that is on when the high frequency oscillator is off to save power. And hence probably the oscillator used most of the time in the workloads of the targeted customers of Energy Micro.

An interface has been implemented which lets the application/driver tell FreeRTOS which energy mode it can put the EFM32 in. A different solution where FreeRTOS puts the EFM32 in energy modes automatically based on how peripherals has been configured was also implemented, but abandoned. It was abandoned mostly because of the constraint on how the peripherals had to be configured for the scheme to work correctly.

It has been a goal to get the changes made included in the official FreeRTOS release. This has been a consideration in all design decisions. Minimal changes are needed to the FreeRTOS source code in order for the solution to work.

A demonstration of how the modified FreeRTOS can be configured in order save power has also been implemented. This has also included implementing a way for the drivers to work in an energy efficient manner. For example using DMA to transfer/receive on RS232. Getting this to work correctly has included days of debugging, both due to my own bugs, but also due to not reading the errata of chip at hand carefully enough.

8.1 Future work

Future work depends on what the maintainer of FreeRTOS thinks of the code. If he says some modifications has to be made in order to get the code into the main FreeRTOS release, those modification should be made. It is suspect that the changes made in order to effectively manage the energy modes is hardest for him to accept. Those changes are much more intrusive as they alter the process descriptor. On the other hand, my changes only alter the FreeRTOS if it is compiled for EFM32.

Using a register which is updated while sleeping as the reference for time in FreeRTOS would also be an improvement to my solution. (This is like idea 1 and 2 in section 4.) Not having to calculate how long was slept when woken up from deep sleep would make the clock drifting issue disappear. It would reduce the wake up time, and the core could then sleep longer. This should be considered on the next generation of EFM32 where a 32 (or 64) bit register will be used to count clock edges on the low power clock source. On the other hand, this kind of change would require a lot more intrusive changes to FreeRTOS.

Another very interesting project would be to make FreeRTOS *totally tickless*. This would involve a major redesign of the FreeRTOS scheduler, as it would have to look at the next scheduled task and set up a hardware interrupt at that time. In contrast to at every timer interrupt check if an event is scheduled now. Having a totally tickles operating system, makes entering sleep mode trivial. This is what

Linux does, and as a side note I want to mention that what I have implemented is very similar to what Linux did before they implemented a *totally tickless* kernel.

A more complex way of doing dynamic power management ($\approx$ entering sleep state) would also be interesting to look into. One of the algorithms described by Benini et al. [Benini et al., 2000] could be looked into. However intuition tells me that as long as the wake up time is as small as it is on the EFM32 ($\approx 2\mu$s) it is not worth using clock cycles to predict how long an idle period will be. Embedded programmers might want control of when and when not to enter sleep states. Predictability and determinism is often much more important in embedded and real time application.

References

[LiA,] Linux source. http://lxr.linux.no/linux+v2.6.13/arch/arm/mach-omap1/time.c.

[arm, 2005] (2005). Cortex m3 technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf.

[Lin, 2010] (2010). Linux source 2.6.34. http://lxr.free-electrons.com/source/.

[Benini et al., 2000] Benini, L., Bogliolo, A., and De Micheli, G. (2000). A survey of design techniques for system-level dynamic power management. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(3):299–316.

[Changelog, 2010] Changelog (2010). Freertos changelog. http://www.freertos.org/History.txt.

[Corbet, 2005] Corbet, J. (2005). The dynamic tick patch. Available: https://lwn.net/Articles/138969/ Viewed: 1. March 2010.

[Dev kit Manual,] Dev kit Manual. User manual development kit efm32-g8xx-dk. http://www.energymicro.com/downloads/tools-documents.

[EFM32 Errata,] EFM32 Errata. Efm32 errata history. http://downloads.energymicro.com/devices/pdf/errata/d0028_efm32g890_errata_history.pdf.

[EFM32 Manual,] EFM32 Manual. Reference manual, efm32g microcontroller family. http://www.energymicro.com/downloads/reference-manuals.

[EFM32G890F128 Datasheet,] EFM32G890F128 Datasheet. Gecko datasheet efm32g890f128/efm32g890f64/efm32g890f32. http://downloads.energymicro.com/devices/pdf/d0010_efm32g890_datasheet.pdf.

[FreeRTOS Code, 2010] FreeRTOS Code (2010). Freertos source code. http://sourceforge.net/projects/freertos/files/FreeRTOS/.

[Gleixner and Niehaus., 2006] Gleixner, T. and Niehaus., D. (2006). Hrtimers and beyond: Transforming the linux time subsystems. in proceedings of the ottawa linux symposium (ols'06).

[Hwang and Wu, 2000] Hwang, C.-H. and Wu, A. C.-H. (2000). A predictive system shutdown method for energy saving of event-driven computation. *ACM Trans. Des. Autom. Electron. Syst.*, 5(2):226–241.

[J Stultz, 2005] J Stultz, N Aravamudan, D. H. (2005). We are not getting any younger: A new approach to time and timers. in proceedings of the ottawa linux symposium (ols'05).

[Rivoire et al., 2007] Rivoire, S., Shah, M. A., Ranganathan, P., Kozyrakis, C., and Meza, J. (2007). Models and metrics to enable energy-efficiency optimizations. *Computer*, 40(12):39–48.

[Sadasivan, 2006] Sadasivan, S. (2006). An introduction to the arm cortex-m3 processor.

[Srinivasan et al., 1998] Srinivasan, B., Pather, S., Hill, R., Ansari, F., and Niehaus, D. (1998). A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *RTAS '98: Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, page 112, Washington, DC, USA. IEEE Computer Society.

[Tverdal, 2009] Tverdal, M. (2009). Power saving features in microprocessors. tdt4592 fall 2009.

A Demo application

This file was developed as a part of this thesis.

A.1 main.c

```
/*
 * This demo application creates 8 coroutines which flashes the 8 leds
     0-7.
 * It creates one task which toogles the leds 8-16, and it creates one
 * task that uses the LCD.
 *
 *
 */

#include "FreeRTOS.h"
#include "croutine.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"

#include "partest.h"
#include "crflash.h"

#include "lcdcontroller.h"
#include "ledtest.h"
#include "lcdtest.h"

// The prioritys of the tasks
#define mainLCD_TASK_PRIORITY (tskIDLE_PRIORITY + 1)
#define mainLED_TASK_PRIORITY (mainLCD_TASK_PRIORITY + 1)

/*
 * Sets up the hardware used in the demo
 */
static void prvSetupHardware( void );

/*
    ***************************************************************************
    *//**
 * @brief  Main function
 ***************************************************************************
     */

int main(void)
{
  prvSetupHardware();
  /*Start the standard coroutines that flashes and set them to flash
      the
   8 lower order leds*/
  vStartFlashCoRoutines(8);

  /*Start the task that animates and writes to the display*/
  xTaskCreate( vLedtestLedTask, "LedTask", configMINIMAL_STACK_SIZE,
      NULL,mainLCD_TASK_PRIORITY, NULL );
```

```
  /*Start the task that animates the 8 uper order leds*/
  xTaskCreate( vLcdtestLcdTask, "LCDTask", configMINIMAL_STACK_SIZE,
      NULL,mainLED_TASK_PRIORITY, NULL );

  /*Start the scheduler*/
  vTaskStartScheduler();
  return 0;
}

/*
 * Application Idle Hook is only used to schedule the coroutines.
 * As long as preemtion is used, we can repedetly
 */
void vApplicationIdleHook( void )
{
  vCoRoutineSchedule();
}



static void prvSetupHardware( void )
{
   /*Initiaise the LEDS*/
  vParTestInitialise();

  /*Set up the LCD*/
  LCD_Init(LCD);
}
```

A.2 lcdtest.c

This file was modified to use the FreeRTOS API.

```
#include "lcdtest.h"
/*
    ************************************************************************
    *//**
 * @brief LCD Test Routine, shows various text and patterns
 ****************************************************************************
      */
void vLcdtestLcdTask(void *pvParameters)
{
  int i;
  LCD_TypeDef *lcd = LCD;
  char *stext = "FreeRTOS Energy Micro       ";

  /* Loop through funny pattern */
  while (1)
  {
    LCD_ScrollText(lcd,stext);
    LCD_AllOff(lcd);

    /*Count down from 100 on the number section of the LCD display*/
    for (i = 100; i > 0; i--)
    {
      LCD_Number(lcd, i);
      vTaskDelay(10);
    }
```

```
    LCD_NumberOff(lcd);

    /*Turn on gecko and efm32 symbol*/
    LCD_Symbol(lcd, LCD_SYMBOL_GECKO, 1);
    LCD_Symbol(lcd, LCD_SYMBOL_EFM32, 1);
    LCD_Write(lcd, " Gecko ");
    vTaskDelay(1000);

    LCD_AllOn(lcd);
    vTaskDelay(1000);

    LCD_AllOff(lcd);
    LCD_Write(lcd, "OOOOOOO");
    vTaskDelay(62);
    LCD_Write(lcd, "XXXXXXX");
    vTaskDelay(62);
    LCD_Write(lcd, "+++++++");
    vTaskDelay(62);
    LCD_Write(lcd, "@@@@@@@");
    vTaskDelay(62);
    LCD_Write(lcd, "ENERGY ");
    vTaskDelay(250);
    LCD_Write(lcd, "@@ERGY ");
    vTaskDelay(62);
    LCD_Write(lcd, " @@RGY ");
    vTaskDelay(62);
    LCD_Write(lcd, " M@@GY ");
    vTaskDelay(62);
    LCD_Write(lcd, " MI@@Y ");
    vTaskDelay(62);
    LCD_Write(lcd, " MIC@@ ");
    vTaskDelay(62);
    LCD_Write(lcd, " MICR@@");
    vTaskDelay(62);
    LCD_Write(lcd, " MICRO@");
    vTaskDelay(62);
    LCD_Write(lcd, " MICRO ");
    vTaskDelay(250);
    LCD_Write(lcd, "-EFM32-");
    vTaskDelay(250);
  }
}
```

A.3 ledtest.c

This file was developed as a part of this thesis.

```
#include "ledtest.h"

void vLedtestLedTask( void *pvParameters )
{
      /*ledOn is used toogle leds*/
      portBASE_TYPE ledOn=pdTRUE;

      for( ;; )
       {
```

```
            for(int i = 8;i<16;i++){
              /*Depending on if ledOn is true or false, turn on or off
                  led number i*/
              vParTestSetLED(i,ledOn);

              /*Delay for 1000 ms*/
              vTaskDelay(1000/portTICK_RATE_MS);
            }
            /*After the for loop, we flip ledOn. On the next run through
                the
            for loop above, the leds will be flipped.*/
            ledOn=~ledOn;
        }
}
```

A.4 ParTest.c

This file was developed as a part of this thesis.

```
#include "FreeRTOS.h"
#include "partest.h"
#include "task.h"
#include "dvk.h"
int j=0;
void vParTestInitialise( void ){
  DVK_init();
  DVK_setLEDs(0);
}
void vParTestSetLED( unsigned portBASE_TYPE uxLED, signed
    portBASE_TYPE xValue ){
  // Suspend all other tasks, in order to make sure no
  // other tasks excecutes this code at the same time
  vTaskSuspendAll();
  portBASE_TYPE leds =DVK_getLEDs();
  if(xValue==pdTRUE){
    //If xValue is set to True, we are turning on a led.
    //We do that by oring 1 into the correct position.
    leds=leds|(1<<uxLED);
  }else{
    //If xValue is set to False, we are turning a led off.
    //We do that by anding 0 into the correct position.
    leds&=~(1<<uxLED);

  }
  DVK_setLEDs(leds);
  xTaskResumeAll();
}
void vParTestToggleLED( unsigned portBASE_TYPE uxLED ){
  vTaskSuspendAll();
  portBASE_TYPE leds =DVK_getLEDs();
  //Use XOR to toogle led since xoring a bit with one toogles the bit,
  //and xoring with 0 leves the bit alone.
  leds=leds^(1<<uxLED);
  DVK_setLEDs(leds);
  xTaskResumeAll();
}
```

A.5 startup_efm32.s

This file was only slightly modified in order to set up the interrupts needed by FreeRTOS.

```
        MODULE  ?cstartup

        ;; Forward declaration of sections.
        SECTION CSTACK:DATA:NOROOT(3)

        SECTION .intvec:CODE:NOROOT(2)

        EXTERN  __iar_program_start
        EXTERN  SystemInit
        PUBLIC  __vector_table
        PUBLIC  __vector_table_0x1c
        PUBLIC  __Vectors
        PUBLIC  __Vectors_End
        PUBLIC  __Vectors_Size

        DATA

__vector_table
        DCD     sfe(CSTACK)
        DCD     Reset_Handler

        DCD     NMI_Handler
        DCD     HardFault_Handler
        DCD     MemManage_Handler
        DCD     BusFault_Handler
        DCD     UsageFault_Handler
__vector_table_0x1c
        DCD     0
        DCD     0
        DCD     0
        DCD     0
        DCD     vPortSVCHandler
        DCD     DebugMon_Handler
        DCD     0
        DCD     xPortPendSVHandler
        DCD     xPortSysTickHandler

        ; External Interrupts
        DCD DMA_IRQHandler  ; 0: DMA Interrupt
        DCD GPIO_EVEN_IRQHandler  ; 1: GPIO_EVEN Interrupt
        DCD TIMER0_IRQHandler  ; 2: TIMER0 Interrupt
        DCD USART0_RX_IRQHandler  ; 3: USART0_RX Interrupt
        DCD USART0_TX_IRQHandler  ; 4: USART0_TX Interrupt
        DCD ACMP0_IRQHandler  ; 5: ACMP0 Interrupt
        DCD ADC0_IRQHandler  ; 6: ADC0 Interrupt
        DCD DAC0_IRQHandler  ; 7: DAC0 Interrupt
        DCD I2C0_IRQHandler  ; 8: I2C0 Interrupt
        DCD GPIO_ODD_IRQHandler  ; 9: GPIO_ODD Interrupt
        DCD TIMER1_IRQHandler  ; 10: TIMER1 Interrupt
        DCD TIMER2_IRQHandler  ; 11: TIMER2 Interrupt
        DCD USART1_RX_IRQHandler  ; 12: USART1_RX Interrupt
        DCD USART1_TX_IRQHandler  ; 13: USART1_TX Interrupt
        DCD USART2_RX_IRQHandler  ; 14: USART2_RX Interrupt
```

```
        DCD USART2_TX_IRQHandler  ; 15: USART2_TX Interrupt
        DCD UART0_RX_IRQHandler  ; 16: UART0_RX Interrupt
        DCD UART0_TX_IRQHandler  ; 17: UART0_TX Interrupt
        DCD LEUART0_IRQHandler  ; 18: LEUART0 Interrupt
        DCD LEUART1_IRQHandler  ; 19: LEUART1 Interrupt
        DCD LETIMER0_IRQHandler  ; 20: LETIMER0 Interrupt
        DCD PCNT0_IRQHandler  ; 21: PCNT0 Interrupt
        DCD PCNT1_IRQHandler  ; 22: PCNT1 Interrupt
        DCD PCNT2_IRQHandler  ; 23: PCNT2 Interrupt
        DCD SYSTICCK_IRQHandler;DCD RTC_IRQHandler  ; 24: RTC
            Interrupt
        DCD CMU_IRQHandler  ; 25: CMU Interrupt
        DCD VCMP_IRQHandler  ; 26: VCMP Interrupt
        DCD LCD_IRQHandler  ; 27: LCD Interrupt
        DCD MSC_IRQHandler  ; 28: MSC Interrupt
        DCD AES_IRQHandler  ; 29: AES Interrupt

__Vectors_End
__Vectors       EQU   __vector_table
__Vectors_Size  EQU     __Vectors_End - __Vectors


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Default interrupt handlers.
;;
        THUMB

        PUBWEAK Reset_Handler
        SECTION .text:CODE:REORDER(2)
Reset_Handler
        LDR     R0, =SystemInit
        BLX     R0
        LDR     R0, =__iar_program_start
        BX      R0

        PUBWEAK NMI_Handler
        SECTION .text:CODE:REORDER(1)
NMI_Handler
        B NMI_Handler

        PUBWEAK HardFault_Handler
        SECTION .text:CODE:REORDER(1)
HardFault_Handler
        B HardFault_Handler

        PUBWEAK MemManage_Handler
        SECTION .text:CODE:REORDER(1)
MemManage_Handler
        B MemManage_Handler

        PUBWEAK BusFault_Handler
        SECTION .text:CODE:REORDER(1)
BusFault_Handler
        B BusFault_Handler

        PUBWEAK UsageFault_Handler
```

```
        SECTION .text:CODE:REORDER(1)
UsageFault_Handler
        B UsageFault_Handler

        PUBWEAK vPortSVCHandler
        SECTION .text:CODE:REORDER(1)
vPortSVCHandler
        B vPortSVCHandler

        PUBWEAK DebugMon_Handler
        SECTION .text:CODE:REORDER(1)
DebugMon_Handler
        B DebugMon_Handler

        PUBWEAK xPortPendSVHandler
        SECTION .text:CODE:REORDER(1)
xPortPendSVHandler
        B xPortPendSVHandler

        PUBWEAK SYSTICCK_IRQHandler
        SECTION .text:CODE:REORDER(1)
SYSTICCK_IRQHandler
        B SYSTICCK_IRQHandler
        ; EFM32G specific interrupt handlers

        PUBWEAK DMA_IRQHandler
        SECTION .text:CODE:REORDER(1)
DMA_IRQHandler
        B DMA_IRQHandler

        PUBWEAK GPIO_EVEN_IRQHandler
        SECTION .text:CODE:REORDER(1)
GPIO_EVEN_IRQHandler
        B GPIO_EVEN_IRQHandler

        PUBWEAK TIMER0_IRQHandler
        SECTION .text:CODE:REORDER(1)
TIMER0_IRQHandler
        B TIMER0_IRQHandler

        PUBWEAK USART0_RX_IRQHandler
        SECTION .text:CODE:REORDER(1)
USART0_RX_IRQHandler
        B USART0_RX_IRQHandler

        PUBWEAK USART0_TX_IRQHandler
        SECTION .text:CODE:REORDER(1)
USART0_TX_IRQHandler
        B USART0_TX_IRQHandler

        PUBWEAK ACMP0_IRQHandler
        SECTION .text:CODE:REORDER(1)
ACMP0_IRQHandler
        B ACMP0_IRQHandler

        PUBWEAK ADC0_IRQHandler
        SECTION .text:CODE:REORDER(1)
```

```
ADC0_IRQHandler
        B ADC0_IRQHandler

        PUBWEAK DAC0_IRQHandler
        SECTION .text:CODE:REORDER(1)
DAC0_IRQHandler
        B DAC0_IRQHandler

        PUBWEAK I2C0_IRQHandler
        SECTION .text:CODE:REORDER(1)
I2C0_IRQHandler
        B I2C0_IRQHandler

        PUBWEAK GPIO_ODD_IRQHandler
        SECTION .text:CODE:REORDER(1)
GPIO_ODD_IRQHandler
        B GPIO_ODD_IRQHandler

        PUBWEAK TIMER1_IRQHandler
        SECTION .text:CODE:REORDER(1)
TIMER1_IRQHandler
        B TIMER1_IRQHandler

        PUBWEAK TIMER2_IRQHandler
        SECTION .text:CODE:REORDER(1)
TIMER2_IRQHandler
        B TIMER2_IRQHandler

        PUBWEAK USART1_RX_IRQHandler
        SECTION .text:CODE:REORDER(1)
USART1_RX_IRQHandler
        B USART1_RX_IRQHandler

        PUBWEAK USART1_TX_IRQHandler
        SECTION .text:CODE:REORDER(1)
USART1_TX_IRQHandler
        B USART1_TX_IRQHandler

        PUBWEAK USART2_RX_IRQHandler
        SECTION .text:CODE:REORDER(1)
USART2_RX_IRQHandler
        B USART2_RX_IRQHandler

        PUBWEAK USART2_TX_IRQHandler
        SECTION .text:CODE:REORDER(1)
USART2_TX_IRQHandler
        B USART2_TX_IRQHandler

        PUBWEAK UART0_RX_IRQHandler
        SECTION .text:CODE:REORDER(1)
UART0_RX_IRQHandler
        B UART0_RX_IRQHandler

        PUBWEAK UART0_TX_IRQHandler
        SECTION .text:CODE:REORDER(1)
UART0_TX_IRQHandler
        B UART0_TX_IRQHandler
```

```
        PUBWEAK LEUART0_IRQHandler
        SECTION .text:CODE:REORDER(1)
LEUART0_IRQHandler
        B LEUART0_IRQHandler

        PUBWEAK LEUART1_IRQHandler
        SECTION .text:CODE:REORDER(1)
LEUART1_IRQHandler
        B LEUART1_IRQHandler

        PUBWEAK LETIMER0_IRQHandler
        SECTION .text:CODE:REORDER(1)
LETIMER0_IRQHandler
        B LETIMER0_IRQHandler

        PUBWEAK PCNT0_IRQHandler
        SECTION .text:CODE:REORDER(1)
PCNT0_IRQHandler
        B PCNT0_IRQHandler

        PUBWEAK PCNT1_IRQHandler
        SECTION .text:CODE:REORDER(1)
PCNT1_IRQHandler
        B PCNT1_IRQHandler

        PUBWEAK PCNT2_IRQHandler
        SECTION .text:CODE:REORDER(1)
PCNT2_IRQHandler
        B PCNT2_IRQHandler

        PUBWEAK xPortSysTickHandler
        SECTION .text:CODE:REORDER(1)
xPortSysTickHandler
        B xPortSysTickHandler

        PUBWEAK CMU_IRQHandler
        SECTION .text:CODE:REORDER(1)
CMU_IRQHandler
        B CMU_IRQHandler

        PUBWEAK VCMP_IRQHandler
        SECTION .text:CODE:REORDER(1)
VCMP_IRQHandler
        B VCMP_IRQHandler

        PUBWEAK LCD_IRQHandler
        SECTION .text:CODE:REORDER(1)
LCD_IRQHandler
        B LCD_IRQHandler

        PUBWEAK MSC_IRQHandler
        SECTION .text:CODE:REORDER(1)
MSC_IRQHandler
        B MSC_IRQHandler

        PUBWEAK AES_IRQHandler
```

```
        SECTION .text:CODE:REORDER(1)
AES_IRQHandler
        B AES_IRQHandler

        END
```

B Code

B.1 energymode.c

This file was developed as a part of this thesis.

```
#include "energymodes.h"

/* If this flag is set we will disconnect the debug interface to allow
 * going down when debugger is connected. However this will disable
    debug
 * access, and thus can be a potential hazard if we're locked out from
 * debug completly */
#undef DISCONNECT_DEBUG_INTERFACE

#define RTCTICKFREQ (configRTCCLOCKFREQUENCY/((1<<configRTCDIVIDER)))

void EM_Enter(unsigned char em)
{
  int dmaclk, auxclk;
  /* Disable AUXHFRCO (debug clock prevents E Modes) */
  if (CMU->STATUS & CMU_STATUS_AUXHFRCOENS)
  {
    auxclk        = 1;
    CMU->OSCENCMD = CMU_OSCENCMD_AUXHFRCODIS;
  }

  /* Make sure DMA clock is running to enter EM2 (see chip errata) */
  if (CMU->HFCORECLKEN0 & CMU_HFCORECLKEN0_DMA)
  {
    dmaclk = 1;
  }
  else
  {
    CMU->HFCORECLKEN0 |= CMU_HFCORECLKEN0_DMA;
    dmaclk             = 0;
  }
  /* Disconnect debug interface (disable pull-up/down) */
#if defined(DISCONNECT_DEBUG_INTERFACE)
  GPIO->ROUTE = 0;
#endif

  if(em==1){
   //EM 1
   SCB->SCR &= ~(1 << SCB_SCR_SLEEPDEEP_Pos);
  }else if(em==2) {
   //EM2
   SCB->SCR |= (1 << SCB_SCR_SLEEPDEEP_Pos);
  }else if(em ==3){
   SCB->SCR |= (1 << SCB_SCR_SLEEPDEEP_Pos);
   CMU->OSCENCMD = CMU_OSCENCMD_LFXODIS | CMU_OSCENCMD_LFRCODIS;

  }

  __WFE();
  if(em==3){
   CMU->OSCENCMD = CMU_OSCENCMD_LFXOEN | CMU_OSCENCMD_LFRCOEN;
```

```
  }
   #if configUSE_HFXO_AS_HFCLK == 1
      CMU->OSCENCMD = CMU_OSCENCMD_HFXOEN;
      while (!(CMU->STATUS & CMU_STATUS_HFXORDY)) ;
      CMU->CMD=CMU_CMD_HFCLKSEL_HFXO;
   #endif
  /* Renenable AUXHFRCO */
  if (auxclk == 1)
  {
    CMU->OSCENCMD = CMU_OSCENCMD_AUXHFRCOEN;
  }
  /* Restore DMA clock (see chip errata) */
  if (!dmaclk)
  {
    CMU->HFCORECLKEN0 &= ~CMU_HFCORECLKEN0_DMA;
  }

}

/*Used to compensate for rounding errors.*/
unsigned int rtcOverflows=0;
unsigned int numberOfSleeps = 0;
portTickType xSleepWhileIdle(){
      portTickType xTickCountIncrement=0;

      /*Make sure the eventregister is set to 0 before we interrupts
          are disabled.*/
      __SEV();
      __WFE();
      portENTER_CRITICAL();
      /*Get tickcount, cant use xTaskGetTickCount() since it enables
          interrupts*/
      portTickType currentTick = xTaskGetTickCount();
      unsigned char emToGoTo = ucTaskEmAllowed();
      if(emToGoTo==0)
      {
         return 0;
      }
      /*If the RTC is syncing the previously written value of COMP0,
          it is
      not safe to go to sleep now*/
      if(RTC->SYNCBUSY){
         return 0;
      }

      /* tickNextEvent is set to portMAX_DELAY. We will not try to
          sleep
       longer than that. If the tick count is going to overflow before
            the next
       event, we will wake up in time to handle it.*/
      portTickType tickNextEvent = portMAX_DELAY;

      #if configUSE_CO_ROUTINES == 1
         portTickType nextCoRoutine= xCoRoutineNextTick(currentTick);
         if(   nextCoRoutine == 0 || nextCoRoutine < currentTick){
            return 0;
         }else{
```

```
        tickNextEvent=nextCoRoutine;
    }
#endif
portTickType nextTask = xTaskNextTick();
if(nextTask==0 || nextTask < currentTick){
    return 0;
}else{
    if(nextTask<tickNextEvent){
        tickNextEvent=nextTask;
    }
}
portTickType ticksUntillNextEvent = tickNextEvent - currentTick;

if(ticksUntillNextEvent<2){
    return 0;
}

/*Check against overflow in calculation of rtcWakeUpVal */
if((ticksUntillNextEvent-1)>=((0xFFFFFFFF)/RTCTICKFREQ))
{
  ticksUntillNextEvent=((0xFFFFFFFF)/RTCTICKFREQ);
}

unsigned int  rtcCountBefore = RTC->CNT;
SysTick->CTRL &= ~(1 << SysTick_CTRL_ENABLE_Pos); //Stop the
     systick counter

unsigned int rtcTickcUntillWakeup = (((ticksUntillNextEvent-1)*
     RTCTICKFREQ)/configTICK_RATE_HZ);

#if configUSE_HFXO_AS_HFCLK == 1
    /* If HFXO is used as core clock, compensate for the long
         startup time of the clock.
     * By default it takes 16384 cycles to start it up.
     *
     * This could be changed to read what startup time is
         confgured for HFXO in the CMU
     */
    if(rtcTickcUntillWakeup > 16384*RTCTICKFREQ/14000000){
        rtcTickcUntillWakeup -= 16384*RTCTICKFREQ/14000000;
    }else{
        SysTick->CTRL |= (1 << SysTick_CTRL_ENABLE_Pos); //Start
             the systick counter
        return 0;
    }
#endif
unsigned int rtcWakeUpVal=rtcTickcUntillWakeup+rtcCountBefore;

/*
 * Guard against overflow in the RTC CNT register.
 * Needed because we can not guarentee if it overflowed before
     or after we read it.
 *
 * If it is less then 4 rtc ticks untill we have to wake up, we
     can not go to sleep.
 * since it can take up to 3 cycles to synchronize the value
     into RTC->COMP0
```

```
 */
if(RTC->IF & RTC_IF_OF || rtcTickcUntillWakeup <4 ){
   SysTick->CTRL |= (1 << SysTick_CTRL_ENABLE_Pos); //Start the
        systick counter
   return 0;
}

if(rtcWakeUpVal>0xFFFFFF){
   RTC->COMP0=0xFFFFFF;
}else{
   RTC->COMP0=rtcWakeUpVal;
}

if(rtcWakeUpVal<=rtcCountBefore){
   while(1);
}
if(tickNextEvent<=currentTick){
   while(1);
}

EM_Enter(emToGoTo);
unsigned int rtcCounterValue = RTC->CNT;
SysTick->CTRL |= (1 << SysTick_CTRL_ENABLE_Pos); //Start the
     systick counter
unsigned int rtcTicksElapsed;

if(RTC->IF & RTC_IF_OF){
   rtcCounterValue = RTC->CNT;
   rtcTicksElapsed = (rtcCounterValue+0xFFFFFF-rtcCountBefore);
}else{
   rtcTicksElapsed=(rtcCounterValue - rtcCountBefore);
}
RTC->COMP0=0x00FFFFFF;
xTickCountIncrement=rtcTicksElapsed*configTICK_RATE_HZ/
     RTCTICKFREQ;
/*
 * These next lines are to compensate for rounding errors.
 * Since the xTickCountIncrement calculation above is done with
     a integer division, it will
 * always round down. Over time, this leads to a big drift on
     the time.
 *
 * Another source of error is the fact that we do know where on
     the RTC clock the CNT value is read.
 * This leads to a error of up to one period of the RTC clock
     each time we co to sleep.
 * Over time, this also leads to an error.
 *
 * To compensate for this xTickCountIncrement is incremented by
     1 if the time in xTickCount (in task.c)
 * is behind the time reported by the number of RTC ticks.
 *
 * To make it simpler to handle overflows, 64 bit arithmetic is
     used at the moment.
 *
 * The effect of this scheme of correcting rounding errors, is
     that the xTickCount (in task.c)
```

```
         * follows the RTC clock.
         */
        uint64_t currentTickFromRTC = ((uint64_t) rtcOverflows<<24 |
            rtcCounterValue)* (uint64_t)configTICK_RATE_HZ/RTCTICKFREQ;
        uint64_t currentTickInXTickCount = (uint64_t)((uint64_t)
            lTaskGetNumberOfOverflows()<<32|currentTick)+
            xTickCountIncrement;
        if( currentTickFromRTC>currentTickInXTickCount){
           if(xTickCountIncrement+1<ticksUntillNextEvent){
              xTickCountIncrement+=1;
           }
        }else if( currentTickFromRTC<currentTickInXTickCount){
             if(xTickCountIncrement>0){
              xTickCountIncrement--;

             }
        }
        numberOfSleeps++;
        if(xTickCountIncrement>=ticksUntillNextEvent){
          while(1);//wtf happenend?
        }
        return xTickCountIncrement;
}
```

B.2 checktiming.c

This file was developed as a part of this thesis.

```
#include "checktiming.h"

#define RTCTICKFREQ (configRTCCLOCKFREQUENCY/((1<<configRTCDIVIDER)))

void LETIMER_init(){
   CMU->LFAPRESC0 &= ~_CMU_LFAPRESC0_LETIMER0_MASK;
   CMU->LFAPRESC0 |= _CMU_LFAPRESC0_LETIMER0_DIV32 <<
       _CMU_LFAPRESC0_LETIMER0_SHIFT;
   CMU->LFACLKEN0 |= CMU_LFACLKEN0_LETIMER0;
   while (CMU->SYNCBUSY) ;
   NVIC_DisableIRQ( LETIMER0_IRQn );
   NVIC_ClearPendingIRQ( LETIMER0_IRQn );
   NVIC_EnableIRQ(LETIMER0_IRQn);
   NVIC_SetPriority(LETIMER0_IRQn,7);

   LETIMER0->IEN=LETIMER_IEN_UF;
   LETIMER0->COMP0 =0; //Count down to 0, at frequency 256Hz. Should
       take 511 seconds.
   LETIMER0->CMD = LETIMER_CMD_START;
   LETIMER0->IFC=LETIMER_IFC_UF;
   while(LETIMER0->SYNCBUSY);
}
extern unsigned int numberOfSleeps;
int underflows=0;
char pcOutput3[100];
int maxDiff=0;//Maximum change between two diffs
int previous=6;
extern unsigned int rtcOverflows;
```

```
void vCheckTiming( void *pvParameters )
{

  LETIMER_init();

  for(;;){
      vTaskDelay(5000/portTICK_RATE_MS);
      portTickType currentTick=xTaskGetTickCount();
      unsigned int tickFromLetimer = underflows<<16|(0xFFFF-LETIMER0->
          CNT);
      tickFromLetimer = ((uint64_t) tickFromLetimer * (uint64_t)
          configTICK_RATE_HZ ) / (uint64_t) (configRTCCLOCKFREQUENCY
          /32);
      int currentDiff = currentTick-tickFromLetimer;
      int diffFromPrev = currentDiff - previous;
      if(abs(diffFromPrev)>maxDiff){
         maxDiff = abs(diffFromPrev);
      }
      previous = currentDiff;
      int currentTickFromRTC =((uint64_t) rtcOverflows<<24 | RTC->CNT)
          * (uint64_t) configTICK_RATE_HZ/RTCTICKFREQ;
      sprintf(pcOutput3,"timer: %d, %d, %d, %d, %d\r\n\0",currentTick,
          tickFromLetimer,currentDiff,maxDiff,numberOfSleeps);
      vSerialPutString(pcOutput3,strlen(pcOutput3));
  }
}

void LETIMER0_IRQHandler(){
   if(LETIMER0->IF & LETIMER_IF_UF){
      underflows++;
      LETIMER0->IFC = LETIMER_IFC_UF;
   }
}
```

B.3 Application

B.3.1 main.c

This file was developed as a part of this thesis.

```
#include <stdio.h>
#include <stdlib.h>

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "croutine.h"

#include "rtc.h"
#include "dvk.h"
#include "crflash.h"
#include "serial.h"
#include "i2cdrv.h"
#include "checktiming.h"
#include "inithw.h"
```

```
#include "measurement.h"
#include "lcdtest.h"
#include "serial.h"

#if configUSE_TICKLESSIDLE == 0
   #define vTaskCanGoToEM(x)
#endif
#define mainMEASUREMENT_PRIORITY (tskIDLE_PRIORITY + 1)
#define mainCHECKTIMING_PRIORITY (mainMEASUREMENT_PRIORITY)
#define mainRXLEUART_PRIORITY    (mainMEASUREMENT_PRIORITY)

#define mainSTACK_SIZE_FOR_TASK_USING_SPRINTF   (
    configMINIMAL_STACK_SIZE + 50)
xSemaphoreHandle bsemLetimerFlashLed;
xComPortHandle * portHandle;

/*
 * vTaskRxLeuart continuously reads characters from the LEUART.
 * It provides the xSerialGetString() function with a buffer big
     enough
 * to store the maximum size of the receive buffer.
 *
 * When it recives a string, it sends the string back to the LEUART,
     adding
 * text before and a line feed after the string.
 */
static void vTaskRxLeuart( void *pvParameters );

/*
    ************************************************************************
    *//**
 * @brief  Main function
 ***************************************************************************
     */
static void vTaskCalc( void *pvParameters )
{
  portTickType time;
  vTaskCanGoToEM(2);
  for(;;){
    vTaskDelay(40);
    time = xTaskGetTickCount();
    while(xTaskGetTickCount()<time+360);
  }
}

int main(void)
{

   inithw();
   vSemaphoreCreateBinary(bsemLetimerFlashLed);
   //xTaskCreate( vLCDTask, "LCD", configMINIMAL_STACK_SIZE, NULL, 2,
       NULL );
   xTaskCreate( vMeasurementTemp, "Temperature",
       mainSTACK_SIZE_FOR_TASK_USING_SPRINTF, NULL,
       mainMEASUREMENT_PRIORITY, NULL );
   xTaskCreate( vTaskRxLeuart, "RXLEUART", configMINIMAL_STACK_SIZE,
       NULL,mainRXLEUART_PRIORITY, NULL );
```

```
    xTaskCreate( vMeasurementLight, "Light",
        mainSTACK_SIZE_FOR_TASK_USING_SPRINTF, NULL,tskIDLE_PRIORITY,
        NULL );
    //xTaskCreate( vTaskCalc, "Calc",
        mainSTACK_SIZE_FOR_TASK_USING_SPRINTF, NULL,tskIDLE_PRIORITY,
        NULL );
    //xTaskCreate( vCheckTiming, "Check",
        mainSTACK_SIZE_FOR_TASK_USING_SPRINTF , NULL,
        mainCHECKTIMING_PRIORITY, NULL );
    #if configUSE_CO_ROUTINES == 1
       vStartFlashCoRoutines(2);
    #endif

    vTaskStartScheduler();
    return 0;
}

/*
 * This task listens to the RS223 bus, if the first character read is
     [0-3] it
 * set this as the energy mode used.
 *
 * It also echos back the string read.
 */
static void vTaskRxLeuart( void *pvParameters )
{

/*
 * Buffer used to to give to xSerialGetString, it is important that it
 * has roomfor all the chars the function might write.
 */
    static char rxBuffer[RXBUFFERSIZE];


/*
 * Buffer used to send over the LEUART.
 */
    static char txBuffer[RXBUFFERSIZE+25];
    vTaskCanGoToEM(2);
    for( ;; )
    {
       xSerialGetString(rxBuffer);
       switch (rxBuffer[0])
       {
       case '0': vTaskCanGoToEM(0);
          break;
       case '1': vTaskCanGoToEM(1);
          break;
       case '2': vTaskCanGoToEM(2);
          break;
       case '3': vTaskCanGoToEM(3);
          break;
       default:
          break;
       }
       sprintf(txBuffer,"Received: \"%s\" \r\n",rxBuffer);
       vSerialPutString(txBuffer,strlen(txBuffer));
```

```
   }
}

void vApplicationIdleHook( void )
{
#if configUSE_CO_ROUTINES == 1
   vCoRoutineSchedule();
#endif
}

vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR *
    pcTaskName ){
   while(1);
}
```

B.3.2 measurement.c

This file was developed as a part of this thesis.

```
#include "measurement.h"

#if configUSE_TICKLESSIDLE == 0
   #define vTaskCanGoToEM(x)
#endif
/*Three functions to to some calculation on the Light values*/
static double prvMin(double * array);
static double prvMax(double * array);
static double prvAvg(double * array);
#define measurmentPOWERDELAY 1000
static char* createTemperatureString(TEMPSENS_Temp_TypeDef *);
/*Queue with ADC result*/
extern xQueueHandle xQueueADC0Result;

/*Array of ADC values*/
static double prvValues[10];

/*Used by booth tasks to store the string they want to send over
    LEUART*/
char pcOutput[150];

void vMeasurementLight( void *pvParameters )
{
        int receive = 0;
        portTickType xLastWakeTime;
   CMU_ClockEnable(cmuClock_ADC0,1);
        vTaskDelay(750);
        xLastWakeTime = xTaskGetTickCount();
   for( ;; )
   {
      for(int i =0;i<10;i++){
         vTaskCanGoToEM(1);
         ADC0->CMD=ADC_CMD_SINGLESTART;
         xQueueReceive( xQueueADC0Result, &receive, portMAX_DELAY );
                        prvValues[i]= receive;
         vTaskCanGoToEM(2);
         vTaskDelayUntil(&xLastWakeTime,measurmentPOWERDELAY/
             portTICK_RATE_MS);
```

```
      }
                  for(int i = 0;i<60;i++){
                    sprintf(pcOutput,"Light Level: max: %f, min: %f, avg
                        : %f

                        \r\n",prvMax(prvValues),prvMin(prvValues),prvAvg
                        (prvValues));
                  }
      vSerialPutString(pcOutput,strlen(pcOutput));

   }
}
void vMeasurementTemp( void *pvParameters )
{
         portTickType xLastWakeTime;
   int stack = uxTaskGetStackHighWaterMark(NULL);
   TEMPSENS_Temp_TypeDef temp;
   int returnValue;
         vTaskDelay(250);
         xLastWakeTime = xTaskGetTickCount();
   for( ;; )
   {
      vTaskCanGoToEM(1);
                returnValue= TEMPSENS_TemperatureGet(I2C0,
                    TEMPSENS_DVK_ADDR,&temp);
      vTaskCanGoToEM(3);
      if(returnValue<0){
         sprintf(pcOutput,"Temperature: invalid:\r\n");
      }else{
         sprintf(pcOutput,"Temperature: %s

             \r\n",createTemperatureString(&temp));
      }
      vSerialPutString(pcOutput,strlen(pcOutput));
      vTaskDelayUntil(&xLastWakeTime,measurmentPOWERDELAY/
          portTICK_RATE_MS);
   }
}


static double prvMin(double * array){
   double min = 10E37;
   for(int i = 0; i<10;i++){
      if(array[i]<min){
         min=array[i];
      }
   }
   return min;
}

static double prvMax(double * array){
   double max = 0;
   for(int i = 0; i<10;i++){
      if(array[i]>max){
         max=array[i];
      }
   }
```

```
    return max;
}

static double prvAvg(double * array){
    double sum = 0;
    for(int i = 0; i<10;i++){
        sum+=array[i];
    }
    return sum/10;
}

    char text[8];
static char* createTemperatureString(TEMPSENS_Temp_TypeDef * temp){
    int showFahrenheit = 0;
  /* Work with local copy in case conversion to Fahrenheit is required
       */
    TEMPSENS_Temp_TypeDef dtemp;
    dtemp = *temp;

    memset(text, ' ', sizeof(text) - 1);
    text[sizeof(text) - 1] = 0;
    text[7] = '\0';
    if (showFahrenheit)
    {
      text[6] = 'F';

    TEMPSENS_Celsius2Fahrenheit(&dtemp);
    }
    else
    {
    text[6] = 'C';
    }

    /* Round temperature to nearest 0.5 */
    if (dtemp.f >= 0)
    {
    dtemp.i += (dtemp.f + 2500) / 10000;
    dtemp.f = (((dtemp.f + 2500) % 10000) / 5000) * 5000;
    }
    else
    {
    dtemp.i += (dtemp.f - 2500) / 10000;
    dtemp.f = (((dtemp.f - 2500) % 10000) / 5000) * 5000;
    }

    if ((dtemp.i < 0) || (dtemp.f < 0))
    {
    text[0] = '-';
    }
    else
    {
    text[0] = '+';
    }
    /* 100s */
    if (abs(dtemp.i) >= 100)
    text[1] = '0' + (abs(dtemp.i) / 100);
```

```
    /* 10s */
    if (abs(dtemp.i) >= 10)
    text[2] = '0' + ((abs(dtemp.i) % 100) / 10);

    /* 1s */
    text[3] = '0' + (abs(dtemp.i) % 10);
    text[4] = '.';
    /* 0.1s */
    text[5] = '0' + (abs(dtemp.f) / 1000);
    return text;
}
```

B.3.3 adc.c

This file was developed as a part of this thesis.

```
#include "adc.h"

xQueueHandle xQueueADC0Result;

/*
    ************************************************************************
    *//**
 * @brief ADC0 interrupt handler
 ***************************************************************************
     */
void ADC0_IRQHandler(void)
{
    ADC_TypeDef *adc = ADC0;
    uint16_t cRx;
    portBASE_TYPE xHigherPriorityTaskWoken=pdFALSE;
    if (adc->IF & ADC_IF_SINGLE)
    {
        cRx = adc->SINGLEDATA;
        xQueueSendFromISR(xQueueADC0Result,&cRx,&
            xHigherPriorityTaskWoken);
        adc->CMD=ADC_CMD_SINGLESTOP;
        adc->IFC = 0xFF;
    }
    portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
}

void initADC(){
/* Amient Light sensor gives out values between 0.1 to 2V
 * Refference of 2.5V is used.
 */
        DVK_enablePeripheral(DVK_AMBIENT);
    xQueueADC0Result = xQueueCreate( 1, ( unsigned portBASE_TYPE )
        sizeof( uint16_t ) );
    CMU->HFPERCLKEN0 |= CMU_HFPERCLKEN0_ADC0;
    ADC_TypeDef *adc= ADC0;
    /* Clear all pending interrupts */
    adc->IFC = _ADC_IFC_MASK;
    adc->IEN = ADC_IEN_SINGLE;
    NVIC_SetPriority(ADC0_IRQn,7);
    /* Enable interrupt in Cortex Core */
    NVIC_ClearPendingIRQ(ADC0_IRQn);
```

```
   NVIC_EnableIRQ(ADC0_IRQn);

   DVK_enablePeripheral(DVK_I2C);
   ADC0->SINGLECTRL = ADC_SINGLECTRL_AT_256CYCLES|
       ADC_SINGLECTRL_INPUTSEL_CH5 | ADC_SINGLECTRL_REF_2V5;
   //CMU_ClockEnable(cmuClock_ADC0,0);
}
```

B.4 Diff for task.c in FreeRTOS

This shows the changes made to task.c as a part of this thesis.

```
--- FreeRTOS6040rignal/FreeRTOS/Source/tasks.c  Sun Mar 14 12:38:14
    2010
+++ martintv-master.git/FreeRTOS/Source/tasks.c Wed May 26 10:01:09
    2010
@@ -113,6 +113,9 @@
   #if ( configGENERATE_RUN_TIME_STATS == 1 )
      unsigned long ulRunTimeCounter;     /*< Used for calculating how
           much CPU time each task is utilising. */
   #endif
+  #if ( configUSE_TICKLESSIDLE == 1)
+     unsigned portBASE_TYPE ucEmAbleToGoTo;
+  #endif

 } tskTCB;

@@ -136,6 +139,9 @@
 PRIVILEGED_DATA static xList * volatile pxDelayedTaskList ;
        /*< Points to the delayed task list currently being used. */
 PRIVILEGED_DATA static xList * volatile pxOverflowDelayedTaskList;
        /*< Points to the delayed task list currently being used to
     hold tasks that have overflowed the current tick count. */
 PRIVILEGED_DATA static xList xPendingReadyList;
     /*< Tasks that have been readied while the scheduler was
     suspended.  They will be moved to the ready queue when the
     scheduler is resumed. */
+#if configUSE_TICKLESSIDLE == 1
+  PRIVILEGED_DATA static unsigned portBASE_TYPE pucTaksAbleToGoToEM[
    configENERGYMODES];
+#endif

 #if ( INCLUDE_vTaskDelete == 1 )

@@ -553,6 +559,9 @@
         scheduler for the TCB and stack. */
         vListRemove( &( pxTCB->xGenericListItem ) );

+     #if configUSE_TICKLESSIDLE == 1
+        pucTaksAbleToGoToEM[(pxTCB->ucEmAbleToGoTo)]--;
+     #endif
         /* Is the task waiting on an event also? */
         if( pxTCB->xEventListItem.pvContainer )
         {
@@ -1826,9 +1835,56 @@
         vApplicationIdleHook();
```

```
      }
      #endif
+
+     #if (configUSE_TICKLESSIDLE == 1)
+        extern portTickType xSleepWhileIdle();
+        portTickType tickSlept=xSleepWhileIdle();
+        xTickCount += tickSlept;
+        portEXIT_CRITICAL();
+     #endif
   }
 } /*lint !e715 pvParameters is not accessed but all task functions
     require the same prototype. */

+#if configUSE_TICKLESSIDLE == 1
+  portTickType xTaskNextTick(void){
+        /*If any tasks are ready to run, start from the top of idle
    loop again,
+        and turn on interrupts. The fact that uxTopReadyPriority is
    used to check
+        if any tasks are ready to run will ignore tasks sharing the
    priority of the
+        idle task*/
+        if(uxTopReadyPriority>0 || xPendingReadyList.uxNumberOfItems
    >0
+           || pxReadyTasksLists[0].uxNumberOfItems>1){
+           return 0;
+        }
+        /*Find out when the next task is to be woken up, and set
    tickNextEvent to
+        this value if it is smaller.*/
+        if(pxDelayedTaskList->uxNumberOfItems>0){
+           return (pxDelayedTaskList->xListEnd.pxNext->xItemValue);
+        }else{
+           return portMAX_DELAY;
+        }
+  }
+
+  void vTaskCanGoToEM(unsigned portBASE_TYPE em){
+     vPortEnterCritical();
+     pucTaksAbleToGoToEM[pxCurrentTCB->ucEmAbleToGoTo]--;
+     pucTaksAbleToGoToEM[em]++;
+     pxCurrentTCB->ucEmAbleToGoTo = em;
+     vPortExitCritical();
+  }
+
+  unsigned portBASE_TYPE ucTaskEmAllowed(){
+     for(int i = 0; i < configENERGYMODES;i++){
+        if(pucTaksAbleToGoToEM[i] > 0){
+           return i;
+        }
+     }
+     return configENERGYMODES-1;
+  }
+
+  portBASE_TYPE lTaskGetNumberOfOverflows(){
+     return xNumOfOverflows;
+  }
```

```
+#endif


@@ -1904,6 +1960,10 @@
      ( void ) xRegions;
      ( void ) usStackDepth;
   }
+  #endif
+  #if (configUSE_TICKLESSIDLE == 1)
+     pxTCB->ucEmAbleToGoTo = confgiDEFAULT_EM_FOR_NEW_TASK;
+     pucTaksAbleToGoToEM[confgiDEFAULT_EM_FOR_NEW_TASK]++;
   #endif
 }
 /*-----------------------------------------------------------*/
```

B.5 C++ program simulating GPS

This file was developed as a part of this thesis.

```
#include <iostream>
#include <windows.h>
#include <process.h>
#include <string.h>
#include <ctime>
using namespace std;
void receive(void *arg);
HANDLE hSerial;
int main(void){
   hSerial=CreateFile(TEXT("COM1"), GENERIC_READ|GENERIC_WRITE, 0, 0,
       OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,0);
   if(hSerial==INVALID_HANDLE_VALUE){
      if(GetLastError()==ERROR_FILE_NOT_FOUND){
         cout << "Does not exsist" << endl;
      }
      cout << "Some other error" << endl;
   }
   DCB dcbSerial;
   DCB dcbSerialParams={0};
   dcbSerial.DCBlength=sizeof(dcbSerialParams);
   if(!GetCommState(hSerial,&dcbSerialParams)){
      cout << "Error getting state" << endl;
   }
   dcbSerialParams.BaudRate=CBR_9600;
   dcbSerialParams.ByteSize=8;
   dcbSerialParams.StopBits=ONESTOPBIT;
   dcbSerialParams.Parity=NOPARITY;
   if(!SetCommState(hSerial,&dcbSerialParams)){
      cout <<"error setting serial port state" << endl;
   }
   HANDLE whSerial=CreateFile(TEXT("Z:\\fag\\master\\report\\martintv-
       master\\plot\\powerMeasurment"), GENERIC_WRITE, 0, 0,
       CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,0);
   if(whSerial==INVALID_HANDLE_VALUE){
      if(GetLastError()==ERROR_FILE_NOT_FOUND){
         cout << "Does not exsist" << endl;
```

```
        }
            cout << "Some other error 2" << endl;
    }

    int run =1;
    int write = 1;

    bool startNewThread = true;

    while(run){
        char szBuff[2]={0};
        DWORD dwBytesRead=0;
        if(!ReadFile(hSerial,szBuff,1,&dwBytesRead,NULL)){
            cout << "Error receiving!"<<endl;
            //erroroccurred.Reporttouser.
        }
        if(dwBytesRead == 1){
            cout << szBuff[0];
            if(write==1)
            {
                string h(1,szBuff[0]);
                DWORD dwBytesRead=0;
                if(!WriteFile(whSerial,h.c_str(),h.length(),&dwBytesRead,
                    NULL))
                {
                    cout << "Error writing file" << endl;
                    write = 0;
                }
                FlushFileBuffers(whSerial);
            }
            if(startNewThread )
            {
                _beginthread( receive, 0, (void*)12 );
                startNewThread = false;
                cout << "Started new thread" << endl;
            }

        }else{
            cout <<"Received more or less then 1" << endl;
        }
    }


    cout << "closing down! " <<endl;
    CloseHandle(hSerial);
    while(1);

}

void send(string s){
    DWORD dwBytesRead=0;
    if(!WriteFile(hSerial,s.c_str(),s.length(),&dwBytesRead,NULL))
    {

    }

}
```

```
void receive(void * arg)
{
   Sleep(200);
   while(1){
      clock_t start_tick(clock());
      send("$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M
          ,46.9,M,,*47\r");
      Sleep(500);
      while(clock()-start_tick < CLOCKS_PER_SEC);
   }
}
```

B.6 Drivers

B.7 Serial

This file was developed as a part of this thesis. The prvLEUART1_Init function is based on the serial driver from Energy Micro.

```
 #include "FreeRTOS.h"
 #include "queue.h"
 #include "task.h"
 #include "semphr.h"

 #include "efm32_cmu.h"
 #include "efm32_gpio.h"
 #include "efm32_leuart.h"

 #include "serial.h"
 #include "efm32.h"
 #include "dvk.h"
#define serialLEUART_SIGFRAME '\r'

 static xQueueHandle xRxedChars;
 static xQueueHandle xCharsForTx;
 static xSemaphoreHandle bsemI2C;


 /*
     ************************************************************************
     *//**
  * @brief Intializes LEUART1 for use as an output interface, 9600-8-N
       -1
  ****************************************************************************
       */
 static void prvLEUART1_Init(void)
 {
   LEUART_TypeDef *leuart = LEUART1;
   LEUART_Init_TypeDef init;

   /* Enable CORE LE clock in order to access LE modules */
   CMU_ClockEnable(cmuClock_CORELE, true);

   /* Do not prescale clock */
   CMU_ClockDivSet(cmuClock_LEUART1, cmuClkDiv_1);
```

```
  /* Enable clock */
  CMU_ClockEnable(cmuClock_LEUART1, true);

  /* Use default location 0: TX - Pin C6, RX - Pin C7 */
  /* To avoid false start, configure output as high */
  GPIO_PinModeSet(gpioPortC, 6, gpioModePushPull, 1);
  /* Define input, no filtering */
  GPIO_PinModeSet(gpioPortC, 7, gpioModeInput, 0);

  /* Enable pins at default location */
  leuart->ROUTE = LEUART_ROUTE_RXPEN | LEUART_ROUTE_TXPEN;

  /* Configure LEUART1 */
  init.enable = leuartDisable;
  init.refFreq = 0;
  init.baudrate = 9600;
  init.databits = leuartDatabits8;
  init.parity = leuartNoParity;
  init.stopbits = leuartStopbits1;
  LEUART_Init(leuart, &init);

  /* Clear previous RX interrupts */
  LEUART_IntClear(LEUART1, LEUART_IF_RXDATAV);
  NVIC_ClearPendingIRQ(LEUART1_IRQn);

  /* Enable RX interrupts */
  LEUART_IntEnable(LEUART1, LEUART_IF_RXDATAV);
  NVIC_EnableIRQ(LEUART1_IRQn);

 NVIC_SetPriority(LEUART1_IRQn,7);
  LEUART_IntEnable(LEUART1, LEUART_IF_TXC);
  /* Finally enable it */
  LEUART_Enable(leuart, leuartEnable);
}

/*
    ************************************************************************
    *//**
* @brief LEUART1 interrupt handler
************************************************************************
    */
void LEUART1_IRQHandler(void)
{
  LEUART_TypeDef *leuart = LEUART1;
  portCHAR cRx;
  portCHAR cTx;
  portBASE_TYPE xHigherPriorityTaskWoken=pdFALSE;
  if (leuart->IF & LEUART_IF_RXDATAV)
  {
     cRx = leuart->RXDATA;
     xQueueSendFromISR(xRxedChars,&cRx,&xHigherPriorityTaskWoken);
  }
  if(leuart->IF & LEUART_IF_TXC ){
     if(xQueueReceiveFromISR(xCharsForTx,&cTx,&
         xHigherPriorityTaskWoken) == pdTRUE){
        leuart->TXDATA = 0xFF & cTx;
        while(leuart->SYNCBUSY);
```

```
      }
      leuart->IFC = LEUART_IFC_TXC;
   }
   portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
}

xComPortHandle xSerialPortInitMinimal( unsigned long ulWantedBaud,
    unsigned portBASE_TYPE uxQueueLength )
{
   xRxedChars = xQueueCreate( uxQueueLength, ( unsigned portBASE_TYPE
       ) sizeof( signed portCHAR ) );
   xCharsForTx = xQueueCreate( uxQueueLength, ( unsigned portBASE_TYPE
        ) sizeof( signed portCHAR ) );
   vSemaphoreCreateBinary(bsemI2C);
   CMU->HFPERCLKEN0 |= CMU_HFPERCLKEN0_USART2;
   DVK_enablePeripheral(DVK_RS232B);
   if(xRxedChars != 0 && xCharsForTx !=0){
      prvLEUART1_Init();
      return NULL;
   }else{
      return 0;
   }

};
signed portBASE_TYPE xSerialPutChar( signed char cOutChar,
    portTickType xBlockTime ){
   if(xQueueSend(xCharsForTx,&cOutChar,xBlockTime)==pdPASS){
      if(LEUART1->STATUS & LEUART_STATUS_TXBL){
        if(!(LEUART1->SYNCBUSY & LEUART_SYNCBUSY_TXDATA)){
         LEUART1->IFS |= LEUART_IFS_TXC;
        }
      }
      return pdPASS;
   }
   return pdFAIL;
}


void vSerialPutString(char * pcString, unsigned short usStringLength
    ){
   signed portCHAR * pcChar = (signed char *)pcString;

   xSemaphoreTake(bsemI2C,2);
   while(*pcChar){
      xSerialPutChar(*pcChar,2);
      pcChar++;
   }
   xSemaphoreGive(bsemI2C);
}

/*
 * Returns, in pcRxString the string read. The pointer needs
 * to have room for RXBUFFERSIZE bytes.
 */
void xSerialGetString( char * pcRxedString )
{
   int i = 0;
```

```
    xQueueReceive( xRxedChars,&pcRxedString[i],portMAX_DELAY );
    while(pcRxedString[i] != serialLEUART_SIGFRAME & i < RXBUFFERSIZE){
            i++;
             xQueueReceive( xRxedChars,&pcRxedString[i],portMAX_DELAY
                 );

    }
    pcRxedString[i] = '\0';//insert a \0 at the end to make it a
        propper string.
}
```

B.7.1 Serial with DMA

This file was developed as a part of this thesis. The prvLEUART1_Init function is based on the serial driver from Energy Micro.

```
/*Read this if LEUART and EM2 does not work!
Errata LEUART1:

LEUART + DMA
EM2 cannot be entered when transmit-
ting the last byte using LEUART and
DMA.

*/
#include "FreeRTOS.h"
#include "queue.h"
#include "task.h"
#include "semphr.h"

#include "efm32_cmu.h"
#include "efm32_gpio.h"
#include "efm32_leuart.h"

#include "serial.h"
#include "efm32.h"
#include "dvk.h"
#include "efm32_dma.h"

#define serialDMATXCHANEL 0
#define serialDMARXCHANEL 1
#define serialLEUART_SIGFRAME '\r'
/*
 * Semaphore used to Signal when the DMA cycle has completed.
 * Is released in the DMACallback() function.
 */
static xSemaphoreHandle bsemDmaTx;

/*
 * Used to ensure that only one task tries to use the DMA for TX at a
    time.
 */
static xSemaphoreHandle bsemDmaActiveTx;

/*
 * Semaphore used to signal when the DMA receive cycle has completed.
```

```
 * Is released in the DMACallback() function
 */
static xSemaphoreHandle bsemDmaRx;

/*
 * Called when a DMA cycle has completed.
 */
void DMACallback(unsigned int channel, bool primary, void *user);

/*
 * Structs used to init the DMA
 */
DMA_Init_TypeDef init;
DMA_CB_TypeDef cbStruct;
DMA_CfgChannel_TypeDef chCfgStuctTx;
DMA_CfgDescr_TypeDef cfgDescrTx;
DMA_CfgChannel_TypeDef chCfgStuctRx;
DMA_CfgDescr_TypeDef cfgDescrRx;

/*
 * Two buffers are used when receiving from the LEUART using the DMA,
 * too ensure that the DMA does not write over something before we are
 * able to copy it.
 */
static char RxBuffer1[RXBUFFERSIZE];
static char RxBuffer2[RXBUFFERSIZE];
static char* RxBufferInUseByDMA = RxBuffer1;
static char* RxBufferIdle = RxBuffer2;

/*
    ************************************************************************
    *//**
 * @brief Intializes LEUART1 for use as an output interface, 9600-8-N
     -1
 ***************************************************************************
     */
static void prvLEUART1_Init(void)
{
  LEUART_TypeDef *leuart = LEUART1;
  LEUART_Init_TypeDef init;

  /* Enable CORE LE clock in order to access LE modules */
  CMU_ClockEnable(cmuClock_CORELE, true);

  /* Do not prescale clock */
  CMU_ClockDivSet(cmuClock_LEUART1, cmuClkDiv_1);

  /* Enable clock */
  CMU_ClockEnable(cmuClock_LEUART1, true);

  /* Use default location 0: TX - Pin C6, RX - Pin C7 */
  /* To avoid false start, configure output as high */
  GPIO_PinModeSet(gpioPortC, 6, gpioModePushPull, 1);
  /* Define input, no filtering */
  GPIO_PinModeSet(gpioPortC, 7, gpioModeInput, 0);

  /* Enable pins at default location */
```

```
  leuart->ROUTE = LEUART_ROUTE_RXPEN | LEUART_ROUTE_TXPEN;

  /* Configure LEUART1 */
  init.enable = leuartDisable;
  init.refFreq = 0;
  init.baudrate = 9600;
  init.databits = leuartDatabits8;
  init.parity = leuartNoParity;
  init.stopbits = leuartStopbits1;
  LEUART_Init(leuart, &init);

  /* Clear previous RX interrupts */
  LEUART_IntClear(LEUART1, LEUART_IF_SIGF);
  NVIC_ClearPendingIRQ(LEUART1_IRQn);
  LEUART1->CTRL |= LEUART_CTRL_TXDMAWU | LEUART_CTRL_RXDMAWU;
  /* Enable RX interrupts */
  LEUART_IntEnable(LEUART1, LEUART_IEN_SIGF);
  NVIC_EnableIRQ(LEUART1_IRQn);
  while(LEUART1->SYNCBUSY);
  NVIC_SetPriority(LEUART1_IRQn,7);
  LEUART1->SIGFRAME = serialLEUART_SIGFRAME;

  /*If revision A of chip, TXC inerrupt and rxdatav must be active*/
  volatile uint32_t *reg = (volatile uint32_t *) 0xE00FFFE8;
  uint32_t minor = reg[0] & 0xF0;
  minor |= (reg[1] >> 4) & 0xF;
  if (minor == 0)
  {

     LEUART1->IEN |=  LEUART_IEN_TXC |LEUART_IEN_RXDATAV;
  }

  /* Finally enable it */
  LEUART_Enable(leuart, leuartEnable);
}


xComPortHandle xSerialPortInitMinimal( unsigned long ulWantedBaud,
    unsigned portBASE_TYPE uxQueueLength )
{
   vSemaphoreCreateBinary(bsemDmaTx);
   bsemDmaActiveTx = xSemaphoreCreateMutex();
   vSemaphoreCreateBinary(bsemDmaRx);
   xSemaphoreTake(bsemDmaTx,portMAX_DELAY);
   xSemaphoreTake(bsemDmaRx,portMAX_DELAY);

   CMU->HFPERCLKEN0 |= CMU_HFPERCLKEN0_USART2;
   DVK_enablePeripheral(DVK_RS232B);

   init.hprot = 0;
   char * tempPtr = pvPortMalloc((16 * DMA_CHAN_COUNT * 2 * 2));

   /*
    *This if is needed to ensure that the pointer is alligned to 256
        bits.
    */
   if(((uint32_t)(tempPtr) & ((16 * DMA_CHAN_COUNT * 2) - 1))){
```

```
      tempPtr = (char *) ( ~((16 * DMA_CHAN_COUNT * 2) - 1)&(uint32_t)
          tempPtr);
      tempPtr += ((16 * DMA_CHAN_COUNT * 2));
   }
   init.controlBlock=(DMA_DESCRIPTOR_TypeDef *) tempPtr;
   DMA_Init(&init);

   chCfgStuctTx.highPri=1;
   chCfgStuctTx.enableInt = 1;
   chCfgStuctTx.select = DMAREQ_LEUART1_TXEMPTY;
   cbStruct = (DMA_CB_TypeDef){DMACallback,0,0};
   chCfgStuctTx.cb = &cbStruct;
   DMA_CfgChannel(serialDMATXCHANEL,&chCfgStuctTx);

   cfgDescrTx.dstInc=dmaDataIncNone;
   cfgDescrTx.srcInc=dmaDataInc1;
   cfgDescrTx.size=dmaDataSize1;
   cfgDescrTx.arbRate=dmaArbitrate1;
   cfgDescrTx.hprot = 0;
   DMA_CfgDescr(serialDMATXCHANEL,1,&cfgDescrTx);



   chCfgStuctRx.highPri=1;
   chCfgStuctRx.enableInt = 1;
   chCfgStuctRx.select = DMAREQ_LEUART1_RXDATAV;
   chCfgStuctRx.cb = &cbStruct;
   DMA_CfgChannel(serialDMARXCHANEL,&chCfgStuctRx);

   cfgDescrRx.dstInc=dmaDataInc1;
   cfgDescrRx.srcInc=dmaDataIncNone;
   cfgDescrRx.size=dmaDataSize1;
   cfgDescrRx.arbRate=dmaArbitrate1;
   cfgDescrRx.hprot = 0;
   DMA_CfgDescr(serialDMARXCHANEL,1,&cfgDescrRx);

   /*Important to set the DMA interrupt to a level below
       configMAX_SYSCALL_INTERRUPT_PRIORITY*/
   NVIC_SetPriority(DMA_IRQn,7);
   prvLEUART1_Init();

   DMA_ActivateBasic(serialDMARXCHANEL,1,0,RxBufferInUseByDMA,(void *)
       &LEUART1->RXDATA,RXBUFFERSIZE-1);

   return NULL;
};

/*
 * pcString is the string to send.
 * The function will not return untill the string has been transfered
     by DMA,
 * hence it is safe to reuse the pointer.
 */
void vSerialPutString(char * pcString, unsigned short usStringLength )
    {
        xSemaphoreTake(bsemDmaActiveTx,portMAX_DELAY); //ensures that
            only task is allowed to start the DMA at a time.
```

```
        DMA_ActivateBasic(serialDMATXCHANEL,1,0,(void *)&LEUART1->
            TXDATA,pcString,usStringLength-1);
   xSemaphoreTake(bsemDmaTx,portMAX_DELAY); //wait untill the DMA
       cycle is complete.
   xSemaphoreGive(bsemDmaActiveTx);
}

/*
 * Returns, in pcRxString the string read. The pointer needs
 * to have room for RXBUFFERSIZE bytes.
 */
void xSerialGetString( char * pcRxedString )
{
   xSemaphoreTake(bsemDmaRx,portMAX_DELAY);
   int i = 0;
   while(RxBufferIdle[i] != serialLEUART_SIGFRAME & i < RXBUFFERSIZE){
      pcRxedString[i] = RxBufferIdle[i];
      i++;
   }
   pcRxedString[i] = '\0';//insert a \0 at the end to make it a
       propper string.
}

void DMACallback(unsigned int channel, bool primary, void *user){
    static signed portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
   if(channel==serialDMATXCHANEL){
          xSemaphoreGiveFromISR(bsemDmaTx, &xHigherPriorityTaskWoken )
              ;
   }
   if(channel==serialDMARXCHANEL){
      static char * tmp;
      tmp = RxBufferInUseByDMA;
      RxBufferInUseByDMA= RxBufferIdle;
      RxBufferIdle = tmp;
      DMA_ActivateBasic(serialDMARXCHANEL,1,0,RxBufferInUseByDMA,(void
           *)&LEUART1->RXDATA,RXBUFFERSIZE-1);
      xSemaphoreGiveFromISR(bsemDmaRx, &xHigherPriorityTaskWoken );
   }
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}

/*
 * Only the SIGFRAME interrupt is enabled.
 * Meaning that this interrut occurs when the LEUART recives a
    character
 * defined in the serialLEUART_SIGFRAME macro.
 *
 * When a sigframe arrives, stop the DMA channel. and run the
    DMACallback function.
 */
void LEUART1_IRQHandler(){
        int tmp;
   if(LEUART1->IF & LEUART_IF_SIGF){
      DMA->CHENC = 0x1UL << serialDMARXCHANEL;
      LEUART1->IFC=LEUART_IF_SIGF;
      DMACallback(serialDMARXCHANEL,1,NULL);
   }
```

```
    if(LEUART1->IF & LEUART_IF_TXC ){
        LEUART1->IFC = LEUART_IF_TXC;
    }else if(LEUART1->IF &LEUART_IF_RXDATAV){
        tmp = LEUART1->RXDATA;
    }
}
```

B.8 I2C

B.8.1 Modified driver from EFMLIB

```
#include "i2cdrv.h"
#include "FreeRTOS.h"
#include "queue.h"

#include "i2cdrv.h"
#include "efm32.h"
#include "dvk.h"
#include "semphr.h"

#include "efm32_gpio.h"

/*
 * Queue used to send item from interrupt routine to the
     I2CDRV_Transfer function
 *
 */
static xQueueHandle xRxedChars;

/*
 * Semaphore used to make sure only one task uses the I2C buss
 * at once.
 */
static xSemaphoreHandle bsemI2C;

#include <stddef.h>
#include "dvk_boardcontrol.h"
#include "i2cdrv.h"
#include "efm32_cmu.h"
#include "efm32_gpio.h"

/*
    ***********************************************************************
    *//**
 * @brief
 *   Initalize basic I2C master mode driver for use on the DVK.
 *
 * @details
 *   This driver only supports master mode, single bus-master. In
     addition
 *   to configuring the EFM32 I2C peripheral module, it also
     configures DVK
 *   specific setup in order to use the I2C bus.
 *
 * @param[in] init
```

```
 *   Pointer to I2C initialization structure.
 ******************************************************************************
     */
void I2CDRV_Init(const I2C_Init_TypeDef *init)
{
   int i;

   vSemaphoreCreateBinary(bsemI2C);
   xRxedChars = xQueueCreate( 1, ( unsigned portBASE_TYPE ) sizeof(
       signed portCHAR ) );

   DVK_enablePeripheral(DVK_I2C);

   CMU_ClockEnable(cmuClock_HFPER, true);
   CMU_ClockEnable(cmuClock_I2C0, true);

   /* Use location 3: SDA - Pin D14, SCL - Pin D15 */
   /* Output value must be set to 1 to not drive lines low... We set
       */
   /* SCL first, to ensure it is high before changing SDA. */
   GPIO_PinModeSet(gpioPortD, 15, gpioModeWiredAnd, 1);
   GPIO_PinModeSet(gpioPortD, 14, gpioModeWiredAnd, 1);

   /* In some situations (after a reset during an I2C transfer), the
       slave */
   /* device may be left in an unknown state. Send 9 clock pulses just
       in case. */
   for (i = 0; i < 9; i++)
   {
      /*
       * TBD: Seems to be clocking at appr 80kHz-120kHz depending on
           compiler
       * optimization when running at 14MHz. A bit high for standard
           mode devices,
       * but DVK only has fast mode devices. Need however to add some
           time
       * measurement in order to not be dependable on frequency and
           code executed.
       */
      GPIO_PinModeSet(gpioPortD, 15, gpioModeWiredAnd, 0);
      GPIO_PinModeSet(gpioPortD, 15, gpioModeWiredAnd, 1);
   }

   /* Enable pins at location 3 (which is used on the DVK) */
   I2C0->ROUTE = I2C_ROUTE_SDAPEN |
                 I2C_ROUTE_SCLPEN |
                 (3 << _I2C_ROUTE_LOCATION_SHIFT);

   NVIC_SetPriority(I2C0_IRQn,7);
   /* Enable interrupt in Cortex Core */
   NVIC_ClearPendingIRQ(I2C0_IRQn);
   NVIC_EnableIRQ(I2C0_IRQn);

   I2C_Init(I2C0, init);
   CMU_ClockEnable(cmuClock_I2C0, false);
}
```

```
/*
    ***************************************************************************
    *//**
 * @brief
 *   Perform I2C transfer.
 *
 * @details
 *   This driver only supports master mode, single bus-master. It does
      not
 *   return until the transfer is complete. Uses interrupts to poll
     for completion
 *
 * @param[in] seq
 *   Pointer to sequence structure defining the I2C transfer to take
     place. The
 *   referenced structure must exist until the transfer has fully
     completed.
 ****************************************************************************
     */
I2C_TransferReturn_TypeDef I2CDRV_Transfer(I2C_TransferSeq_TypeDef *
    seq)
{
   I2C_TransferReturn_TypeDef ret;
   ret = I2C_TransferInit(I2C0, seq);
   if(ret == i2cTransferInProgress)
   {
      xQueueReceive( xRxedChars, &ret, portMAX_DELAY );
   }
   return(ret);
}
/*
    **************************************************************************
    *//**
 * @brief I2C0 interrupt handler
 ***************************************************************************
     */
void I2C0_IRQHandler(void)
{
   I2C_TransferReturn_TypeDef ret;
   I2C_TypeDef *I2C = I2C0;
   portBASE_TYPE xHigherPriorityTaskWoken=pdFALSE;
   ret = I2C_Transfer(I2C0);
   if(ret != i2cTransferInProgress){
      xQueueSendFromISR(xRxedChars,&ret,&xHigherPriorityTaskWoken);
   }
   I2C->IFC = _I2C_IFC_MASK;
   portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
}
```

B.8.2 My own I2C Driver

This is a driver I developed. When the official I2C driver from Energy Micro was released, I modified that driver to use my version of FreeRTOS instead of using this driver.

```
#include "FreeRTOS.h"
```

```
#include "queue.h"

#include "i2c.h"
#include "efm32.h"
#include "dvk.h"
#include "semphr.h"
static xQueueHandle xRxedChars;

static xSemaphoreHandle bsemI2C;


/*
    ************************************************************************
    *//**
 * @brief Intializes I2C0 interrupt on RX (receive)
 ***************************************************************************
     */
static void I2C0_IRQ_init(void)
{
  I2C_TypeDef *I2C = I2C0;

  /* Clear all pending interrupts */
  I2C->IFC = _I2C_IFC_MASK;
  /* Enable desired I2C interrupts */
  I2C->IEN = I2C_IEN_ARBLOST|I2C_IEN_RXDATAV|I2C_IEN_ACK|I2C_IEN_NACK;
      //I2C_IEN_TXBL
  NVIC_SetPriority(I2C0_IRQn,7);
  /* Enable interrupt in Cortex Core */
  NVIC_ClearPendingIRQ(I2C0_IRQn);
  NVIC_EnableIRQ(I2C0_IRQn);

}


/*
    ************************************************************************
    *//**
 * @brief Intializes LFXO as LFBCLK for use with I2C0
 ***************************************************************************
     */
static void I2C0_CMU_init(void)
{
  CMU_TypeDef *cmu = CMU;

  /* Setup and enable I2C clock */
  cmu->HFPERCLKEN0 |= CMU_HFPERCLKEN0_I2C0;
  cmu->HFPERCLKDIV |= CMU_HFPERCLKDIV_HFPERCLKEN;

}

/*
    ************************************************************************
    *//**
 * @brief Intializes I2C0 for use as an output interface
 * @param baudrate 300 - 9600 baud
 * @param databits 7 or 8 data bits
```

```
 * @param parity 0 = No parity, 01 = Resrvd, 02 = Even parity, 03 =
     Odd parity
 * @param stopbits 0 or 1 stop bits
 ***************************************************************************
     */
void I2C0_init()
{
 GPIO_TypeDef   *gpio   = GPIO;
  I2C_TypeDef *I2C = I2C0;
  uint32_t       clkdiv;

  //This connects the temperature sensor on the kit to the i2c buss.
  DVK_enablePeripheral(BC_PERCTRL_I2C);
  DVK_disablePeripheral(BC_PERCTRL_I2C);
  DVK_enablePeripheral(BC_PERCTRL_I2C);

  /* Configure I2C0 LFBCL */
  I2C0_CMU_init();
  /* Clear RX/TX buffers */
  I2C0->CMD =  I2C_CMD_CLEARTX |I2C_CMD_ABORT | I2C_CMD_CLEARPC|
      I2C_CMD_CLEARTX;
  I2C0->CLKDIV=0xFF;
  I2C0->ROUTE=I2C_ROUTE_SCLPEN|I2C_ROUTE_SDAPEN|
      I2C_ROUTE_LOCATION_LOC3;

    gpio->P[3].DOUT   = (1 << 14) | (1 << 15);
  gpio->P[3].MODEH &= ~(
    _GPIO_P_MODEH_MODE14_MASK|
    _GPIO_P_MODEH_MODE15_MASK);
  gpio->P[3].MODEH |= GPIO_P_MODEH_MODE14_WIREDAND;
  gpio->P[3].MODEH |= GPIO_P_MODEH_MODE15_WIREDAND;


  /* Configure interrupt handler */
  I2C0_IRQ_init();

  I2C0->CTRL |=I2C_CTRL_EN;
}

/*
    ************************************************************************
    *//**
 * @brief I2C0 interrupt handler
 ***************************************************************************
     */
void I2C0_IRQHandler(void)
{
  I2C_TypeDef *I2C = I2C0;
  signed portCHAR cRx;
  portBASE_TYPE xHigherPriorityTaskWoken=pdFALSE;
  if (I2C->IF & I2C_IF_RXDATAV)
  {
   cRx = I2C->RXDATA;
   xQueueSendFromISR(xRxedChars,&cRx,&xHigherPriorityTaskWoken);
   I2C->CMD=I2C_CMD_NACK;
   I2C->CMD=I2C_CMD_STOP;
   I2C->IFC = I2C_IFC_RXDATAV;
```

```
  }

  if (I2C->IF & I2C_IF_ACK)
  {
   I2C->IFC = I2C_IFC_ACK;
  }

  if (I2C->IF & I2C_IF_NACK)
  {
   I2C->CMD=I2C_CMD_STOP;
   I2C->IFC = I2C_IFC_NACK;
  }
  if (I2C->IF & I2C_IF_ARBLOST)
  {
   I2C->IFC = I2C_IFC_ARBLOST;
  }
  portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);


}

xComPortHandle xI2CPortInitMinimal( )
{

  vSemaphoreCreateBinary(bsemI2C);
  xRxedChars = xQueueCreate( 1, ( unsigned portBASE_TYPE ) sizeof(
      signed portCHAR ) );
  I2C0_init();
}

signed portBASE_TYPE xI2CGetChar( unsigned portCHAR pcAdress, signed
    portCHAR *pcRxedChar, portTickType xBlockTime )
{
  I2C_TypeDef *i2c = I2C0;
  portBASE_TYPE read = pdFALSE;
  if(xSemaphoreTake(bsemI2C,xBlockTime)){
   //Transmit the adress, 7 bits, bluss the last bit set to 0, to
       signal a read
// I2C0->CMD =  I2C_CMD_CLEARTX |I2C_CMD_ABORT | I2C_CMD_CLEARPC|
    I2C_CMD_CLEARTX;
/* i2c->CMD=I2C_CMD_START;
   i2c->CMD=I2C_CMD_STOP;
*/
   i2c->CMD=I2C_CMD_START;
   portBASE_TYPE tx=pcAdress|0x01;//(pcAdress<<1) | 1;

   i2c->TXDATA =tx;

   read = xQueueReceive( xRxedChars, pcRxedChar, xBlockTime );
   xSemaphoreGive(bsemI2C);
  }
  if (read==pdTRUE){
   return pdTRUE;
  }else{
    return pdFALSE;
  }
}
```

B.8.3 Script to process power csv files

This file was developed as a part of this thesis.

```
import os
dirList=os.listdir(".")
listOfFiles = []

for fname in dirList:
    if fname.find(".csv") >=0:
        if fname.find("~")<0:
            listOfFiles.append(fname)
timeStart = -1
minTimeDiff = 4000000000
i = 0
filenameOfSmalest = ""
for fileName in listOfFiles:
    timeStart = -1
    file = open(fileName,"r")
    fileWriteTo = open(fileName.replace(".csv","Temp.csv"),"w")
    fileWriteAlso = open(fileName.replace(".csv","TempAlso.csv"),"w")
    print "Working on file: " + str(file)
    lastTime = 0
    for line in file:
        fileWriteAlso.write(line.replace("\r\n","") + "; 4856 ; 14 ; 0
            x449ef90 ; 0x449edf0 ; 1775\r\n")
        #fileWriteAlso.write(line + "MARTINERKUL")
        if line[0] != "#":
            values = line.split(";")
            if(timeStart<0):
                timeStart=int(values[0])
            lastTime = int(values[0])-timeStart
            fileWriteTo.write(str(int(lastTime)))
            fileWriteTo.write(" ")
            fileWriteTo.write(str(float(values[1])*float(values[2])))
            fileWriteTo.write("\n")
    if lastTime<minTimeDiff:
        minTimeDiff = lastTime
        filenameOfSmalest = file
print "minumu timeDiff" + str(minTimeDiff) + "in File" + str(
    filenameOfSmalest)

dirList=os.listdir(".")
listOfFiles=[]
for fname in dirList:
    if fname.find("Temp.csv") >=0:
        if fname.find("~")<0:
            listOfFiles.append(fname)

for fileName in listOfFiles:
    file = open(fileName,"r")
    fileWriteTo = open(fileName.replace("Temp.csv", "New.csv"),"w")
    print "\tWorking on file second Time: " + str(file)
    for line in file:
        values = line.split(" ")
        timeOfLine=int(values[0])
        if timeOfLine<=minTimeDiff:
            if timeOfLine>=9000 and timeOfLine<=11900:
```

```
            fileWriteTo.write(line)
    else:
        print "\t\tStoping at time " + str(timeOfLine) + "in file
            " + str(file)
        break
```

C Figures

Bit	Name	Reset	Access	Description
31:16	Reserved	*To ensure compatibility with future devices, always write bits to 0.*		
15	I2C0	0	RW	**I2C 0 Clock Enable**
	Set to enable the clock for I2C0.			
14	ADC0	0	RW	**Analog to Digital Converter 0 Clock Enable**
	Set to enable the clock for ADC0.			
13	VCMP	0	RW	**Voltage Comparator Clock Enable**
	Set to enable the clock for VCMP.			
12	GPIO	0	RW	**General purpose Input/Output Clock Enable**
	Set to enable the clock for GPIO.			
11	DAC0	0	RW	**Digital to Analog Converter 0 Clock Enable**
	Set to enable the clock for DAC0.			
10	PRS	0	RW	**Peripheral Reflex System Clock Enable**
	Set to enable the clock for PRS.			
9	Reserved	*To ensure compatibility with future devices, always write bits to 0.*		
8	ACMP1	0	RW	**Analog Comparator 1 Clock Enable**
	Set to enable the clock for ACMP1.			
7	ACMP0	0	RW	**Analog Comparator 0 Clock Enable**
	Set to enable the clock for ACMP0.			
6	TIMER2	0	RW	**Timer 2 Clock Enable**
	Set to enable the clock for TIMER2.			
5	TIMER1	0	RW	**Timer 1 Clock Enable**
	Set to enable the clock for TIMER1.			
4	TIMER0	0	RW	**Timer 0 Clock Enable**
	Set to enable the clock for TIMER0.			
3	UART0	0	RW	**Universal Asynchronous Receiver/Transmitter 0 Clock Enable**
	Set to enable the clock for UART0.			
2	USART2	0	RW	**Universal Synchronous/Asynchronous Receiver/Transmitter 2 Clock Enable**
	Set to enable the clock for USART2.			
1	USART1	0	RW	**Universal Synchronous/Asynchronous Receiver/Transmitter 1 Clock Enable**
	Set to enable the clock for USART1.			
0	USART0	0	RW	**Universal Synchronous/Asynchronous Receiver/Transmitter 0 Clock Enable**
	Set to enable the clock for USART0.			

Figure 24: Shows bit assignments in HFPERCLKEN0. This is where the clock to high frequency peripherals are turned disabled/enabled. Taken from [EFM32 Manual,].

Bit	Name	Reset	Access	Description
31:16	*Reserved*	*To ensure compatibility with future devices, always write bits to 0.*		
15	I2C0	0	RW	**I2C 0 Clock Enable**
	Set to enable the clock for I2C0.			
14	ADC0	0	RW	**Analog to Digital Converter 0 Clock Enable**
	Set to enable the clock for ADC0.			
13	VCMP	0	RW	**Voltage Comparator Clock Enable**
	Set to enable the clock for VCMP.			
12	GPIO	0	RW	**General purpose Input/Output Clock Enable**
	Set to enable the clock for GPIO.			
11	DAC0	0	RW	**Digital to Analog Converter 0 Clock Enable**
	Set to enable the clock for DAC0.			
10	PRS	0	RW	**Peripheral Reflex System Clock Enable**
	Set to enable the clock for PRS.			
9	*Reserved*	*To ensure compatibility with future devices, always write bits to 0.*		
8	ACMP1	0	RW	**Analog Comparator 1 Clock Enable**
	Set to enable the clock for ACMP1.			
7	ACMP0	0	RW	**Analog Comparator 0 Clock Enable**
	Set to enable the clock for ACMP0.			
6	TIMER2	0	RW	**Timer 2 Clock Enable**
	Set to enable the clock for TIMER2.			
5	TIMER1	0	RW	**Timer 1 Clock Enable**
	Set to enable the clock for TIMER1.			
4	TIMER0	0	RW	**Timer 0 Clock Enable**
	Set to enable the clock for TIMER0.			
3	UART0	0	RW	**Universal Asynchronous Receiver/Transmitter 0 Clock Enable**
	Set to enable the clock for UART0.			
2	USART2	0	RW	**Universal Synchronous/Asynchronous Receiver/Transmitter 2 Clock Enable**
	Set to enable the clock for USART2.			
1	USART1	0	RW	**Universal Synchronous/Asynchronous Receiver/Transmitter 1 Clock Enable**
	Set to enable the clock for USART1.			
0	USART0	0	RW	**Universal Synchronous/Asynchronous Receiver/Transmitter 0 Clock Enable**
	Set to enable the clock for USART0.			

Figure 25: Bit assignments in HFCORECLKEN0. Taken from [EFM32 Manual,].