Lab Exercise: MCU Energy Modes

Author: jeremy.constantin@epfl.ch

This lab exercise illustrates the effects and importance of different energy modes for power efficient application design, using an EFM32 starter kit board (STK-3600), which hosts an ARM Cortex-M3 MCU. The board supports real-time energy profiling, allowing us to analyze the energy consumption of applications, while using different energy modes and MCU clock frequencies.

Preparation

1. Boot the machines in ELG022 and log-in with your EPFL account.
2. Open the zip file containing the project files on the Moodle, following the link “Exercise (EFM32 Project Files)”.
3. Open Simplicity Studio.
4. Connect the starter-kit board via USB over the J-Link interface to the workstation, and wait for the drivers to automatically be installed. Make sure the switch on the board is set to “DBG”. The “EFM 32 Leopard Gecko Starter Kit Board” (EFM32LG990F256 MCU) should be automatically detected now. It should appear under “Debug Adapters”
5. Select the EFM 32 Board on the “Debug Adaptors” panel.
6. The reference manual of the EFM32LG MCU family (in the following, this document will be referred to as RM) can be accessed in Simplicity Studio on the Documentations tab. You also find the MCU data sheet of the EFM32LG990 device there, with electric specifications and a system summary.
7. The EFM32LG emlib API documentation (MCU & peripheral configuration) can be accessed over the “Documentation” tab in Simplicity Studio after selecting the connected boards in the “Debug Adapters” tab. On the website, select your device (Leopard Gecko) and find the EMLIB (peripheral library)
8. Start Eclipse through the “Simplicity IDE” button on the “Compatible Tools” tab. You may receive a “Workspace Unavailable” error; Choose the EAWS_Gecko_Workspace” directory on your desktop or create a Workspace directory in the Documents folder of your users profile.
9. Check for your workspace directory, by choosing *File -> Switch Workspace -> Other…* . Do not change the workspace location, just remember it
10. Copy the “workload” directory from the zip file to the workspace directory from the previous point.
11. Import the project by selecting Project -> Import -> MCU Project…; Browse under “Select a project to import” to find and select the workload directory. For project name, use “workload” even if another name is suggested. Then proceed. The workload project will be automatically selected for import). Now open the workload.c source file in the src/ directory of the imported project on the left-hand side. The file contains all the C code you will work on for this exercise.

Task 1: Energy Modes

The exercise is based on an example application that is provided to you, which executes a workload (matrix multiplications) in periodic time intervals on the MCU. The periodic time interval is realized with the help of a real-time counter (RTC) peripheral, which is configured to generate an interrupt every second. The main() function of the application initially sets up the MCU and peripherals, and then runs an infinite loop. The loop waits for a ready-flag to be set by the interrupt, and after that proceeds to perform the workload operations.

After finishing the workload the application returns into the waiting state. In the following, we will analyze three different possibilities for the implementation of the waiting period:

1. The MCU remains in active mode (EM0), and the core actively checks for the ready flag to be set (busy waiting / spin lock).
2. The MCU enters the sleep mode (EM1), disabling the clock of the core (clock gating), which means that no instructions are executed on the MCU until it is woken up by an interrupt. The wakeup of the MCU happens instantaneous.
3. The MCU enters the deep-sleep mode (EM2), turning off all high frequency clocks in the MCU (including the core clock). The MCU is woken up by an interrupt with a wakeup time of about 3 us.

Please refer to RM page 8 and 109 for a complete overview of the different energy modes (EM0-EM4) and their associated active clocks and modules.

To enter the sleep / energy modes, call the following C functions: (cf. APIDOC: Modules / EM_Library / EMU)

- for EM1: **`EMU_EnterEM1();`**
- for EM2: **`EMU_EnterEM2(true);`**

In addition to investigating the different energy modes, we want to examine the impact of different core clock frequencies on the energy efficiency of our application. We will use three different frequencies, by setting the core clock divider to either 1 [= 48 Mhz] (base frequency), 2 [= 24 MHz], or 4 [= 12 Mhz]. This setting can be adjusted by changing the **`CORE_CLKDIV`** define at the top of the C source file.

The provided version of the source code will perform waiting in active mode (EM0) with a core clock of 48 MHz. Your task is now to modify the application and profile its energy consumption for all 9 combinations of the 3 waiting strategies and the 3 clock frequencies. Fill the following table using the profiling tool introduced below.

	MCU frequency:	48 MHz	24 MHz	12 MHz
workload-period: 1s	processing time [ms]			
	processing current [mA]			
EM0 (active)	idle current [mA]			
	total avg. power [mW]			
EM1 (sleep)	idle current [mA]			
	total avg. power [mW]			
EM2 (deepsleep)	idle current [mA]			
	total avg. power [mW]			

Energy Profiling

The application can be compiled, programmed to the board, ran and profiled, simply by choosing **Run -> Profile** in the Eclipse IDE. This will start the Energy Profiler, which displays a real-time trace of the current consumption

of the MCU. Toggle running button ( ) to stop the trace. As can be seen in the following screenshot, the profiler allows for displaying the current consumption of single measurement points by clicking on the waveform (blue line). Ranges can be analyzed by clicking and dragging in the waveform window (green area).

The example screenshot shows the current trace for our application, when using active waiting and a clock speed of 48 Mhz. The selected point indicates that the MCU consumes 15.49 mA during its processing phase (performing the workload). The selected range tells us that the time span between interrupts is indeed 1.00 seconds, and that the total average power is 44.12mW, which corresponds to a consumed energy for one workload period of 12.26µWh. Furthermore, we can observe that even when performing active waiting, the current consumption during the idle phase is lower, since the executed instructions for the busy waiting loop cost less energy than the multiplications and additions and memory accesses of the workload code.

Simplicity Studio Energy Profiler

Questions

1. When comparing the 3 clock frequency configurations, which is the most energy efficient, when performing active waiting (EM0)? By which factor do they differ? Explain why.
2. Which is the most energy efficient clock frequency configuration, when using deep-sleep (EM2)? Explain why the picture changes; what is different to operation in EM0?
3. Now compare your results with the simple sleep mode (EM1); which frequency is most energy efficient here? Contrast the results with EM2, and explain the behavior of the idle current (refer to RM page 109).

Extension to Task 1: Stop Mode (EM3)

Perform the same analysis for a fourth configuration, where during idle the MCU goes into the stop mode (EM3). The main difference between deep-sleep (EM2) and stop mode (EM3) is that in addition to all high frequency clocks, almost all peripheral clocks are also turned off (including the standard RTC peripheral clock). Change the source code to use EM3:

1. Refer to chapter 10.3 and 21.3.5 in the RM for a description of EM3 and how to use the RTC correctly when using EM3.
2. The API call to enter EM3 can be found in the APIDOC: Modules/EMU
3. The clock selection for the RTC peripheral is performed by the **`CMU_ClockSelectSet()`** call; details can be found in the APIDOC: Modules/CMU
4. Adjust the parameters of the RTC clock configuration (divider, compare value, etc.), to match the properties of the new selected clock, ensuring that an interrupt is still generated about once per second.

How does the overall energy consumption further change compared to EM2 deep-sleep?
What is the tradeoff regarding timing accuracy of the workload period interval? Check your measurements and explain the cause. (cf. RM 21.3.5)

Task 2: Energy Profiling of Arithmetic Instructions & Memory Accesses

In this second task we illustrate the impact of addition and multiplication instructions, as well as memory accesses, on the energy consumption of the MCU. To this end, the workload.c source already contains a section that implements two simple workload processing loops, which simply perform repeated additions. The difference between the two versions is that the intermediate result is either being held in a register, or the result is written back to memory (on the stack) after every addition, creating memory accesses.

Run and profile the two addition workloads of task 2 by uncommenting the task2() call in the main() function. This will only execute the part of the application required for this task, since the call will never return to the main() function. Observe the difference in current and energy consumption for the two implementations, showcasing the impact of memory accesses, and note the results in the table below. Make sure the core clock divider is set to 1 [= 48 MHz], to be able to see more pronounced differences.

	Workload:	Addition	Multiplication
temp. result in registers	processing time [ms]		
	processing current [mA]		
	total energy [mJ]		
temp. result in memory	processing time [ms]		
	processing current [mA]		
	total energy [mJ]		

Now extend task 2 by two additional workload loops, which perform multiplications during their operation, either using registers or using memory for storage of the temporary results (analogous to the addition workloads).

How do the current consumptions (surprisingly) compare to the addition workloads? To get the full picture, compare the total energy costs for the different workloads.