Exercises and report template

Investigating the embedded operating system FreeRTOS (1)

|  |
| --- |
| **Information:**   * During the laboratory you should record your work. The aim of this is to help the lecturers to rate your work (so it is not necessary to record all the details). However the following two are required: * The relevant parts (modifications) of the source files made during solving the exercises (it is not necessary to copy the source code here – unless the lecturers ask you otherwise). * Answers to the questions in this document (the areas to be filled are marked by light yellow background). * We marked advanced exercises with a \*. If you want your work to be rated as “outstanding” you should solve some of them. * If you have any feedback regarding the laboratory you can send an e-mail to NASZÁLY Gábor (naszaly AT mit.bme.hu). |
| **Name and Neptun-code of the student(s):** |

|  |
| --- |
| **Preparing for the laboratory:**   1. Read the FreeRTOS guide made for this laboratory! 2. Have a look at the exercises! If something is unclear please consult with the guide!   At the beginning of the laboratories a little test should be accomplished. To pass this test you should be clear with the exercises and the theoretical background of them. |

# Introduction to the development environment (done together)

|  |
| --- |
| The aim of this exercise is to get familiar with the development environment used during the laboratories. We are about to investigate the followings:   * Connect the development board to the virtual machine used during the laboratories and to check if everything works as expected. * Download a demo application to the board, launch it and observe the power consumption while the application is running (tool to be used: Energy Profiler) * Get familiar with the development process using a simple example project (tool to be used: Simplicity IDE):   + Investigate the structure of a project. See what files are needed during the development.   + Build the example project.   + Download the project to the board.   + Get familiar with the debug functionalities by debugging the application we have just downloaded. * Have a look at some API functions needed to handle the followings:   + LEDs and push buttons.   + Usage of the C standard I/O library. |

# Start one thread

|  |
| --- |
| The aim of this exercise is to see if FreeRTOS is capable to create a task[[1]](#footnote-1) from a function and to run it after the scheduler has been started by making a simple “Hello, World!” application.  Notes:   * Create a task with single-shot structure! * Fill in the file **01\_hello\_world.c** at places marked as “TODO”! |
| Place a screenshot of the terminal showing the output generated by the application: |
|  |
| \* What happens if we omit deleting the task (which is normally needed at the and for a single-shot task)? At first it seems OK. However something is wrong! Pause the application and figure out what is the problem! |
|  |

# Scheduling two (independent) threads

## Base exercise

|  |
| --- |
| The aim of this exercise is to test (by the creation of two task with different priorities) if FreeRTOS is able to schedule these tasks (taking their priorities into account).  Notes:   * The tasks should have endless-loop structure! * Both tasks should print a message to the standard output. The messages should contain something which identifies the task and a counter which says what cycle the task is actually running. * The lower priority task should delay its execution by half a second in each cycle! * The higher priority task should delay its execution by a second in each cycle! * To delay the execution of a task use API function vTaskDelay()[[2]](#footnote-2) which puts the task into the BLOCKED state for the given amount of time. * Fill in the file **02\_two\_independent\_threads.c** at places marked as “TODO”! |
| Place a screenshot of the terminal showing the output generated by the application: |
|  |
| Explain the order of execution by the first five messages! |
|  |

## Lower the power consumption during idle state

|  |
| --- |
| The aim of this exercise is to show that it is possible to lower the power consumption if we put the microcontroller into some sleep mode during idle state.  Notes:   * Check the power consumption of the application at its current state! * Then modify the application as follows:   + Enable calling void vApplicationIdleHook(void) from the idle task (to do this configUSE\_IDLE\_HOOK should be set to the right value in FreeRTOSConfig.h)!   + Implement this function in such a way that by invoking EMU\_EnterEM1()[[3]](#footnote-3) it puts the processor into a sleep state (the declaration of this function can be found in em\_emu.h)! * Let’s see the power consumption again! |
| By the help of Energy Profiler measure the average power consumption of the application (the result should be confirmed by putting a screen shot of Energy Profiler)! |
|  |
| Modify the code to put the microcontroller into a sleep mode during idle state. How much is the power consumption now (the answer should be confirmed again by a screen shot)? |
|  |

# Stack usage

|  |
| --- |
| The stack is vital for the correct execution of the application. First of all, it stores the return addresses after a function call. And second, local variables of a C function are also stored on the stack. If we use an operating system we have threads. Each thread shall have its very own instance of stack!  Sometimes stacks can be smaller than required. In simple processor architectures there isn’t any protection against stack overflow. And if a stack grows beyond its boundaries really bad and nasty things can happen as the stack can overwrite some part of the data memory (potentially holding the values of other variables)! This results in very unpredictable errors!  On some architectures a limit can be configured for the size of the stack. If the stack tries to grow beyond this limit an exception is generated. The exception is handled by software, usually by stopping execution (like entering into an endless loop) or by resetting the CPU. Although this prevents potential damage but it comes at the cost of halting or resetting the application.  In this exercise we are about to first make an application breaching stack. And second, turn on the stack overflow monitoring capabilities of FreeRTOS to save the application. |

## \* Leaking the stack

|  |
| --- |
| Make an application that periodically consumes more and more stack. |
| Present the application to the lecturer. |
|  |

## \* Stack overflow monitoring under FreeRTOS

|  |
| --- |
| Under certain circumstances FreeRTOS can detect if a task’s stack is about to overflow. Modify the application made before to utilize the stack overflow monitoring capabilities of FreeRTOS to prevent the task consuming its entire stack! |
| Present the modified application to the lecturer. |
|  |

1. The word “task” is a little bit ambiguous: it is sometime referring to a “thread” and sometimes to a “process”. In correlation with FreeRTOS we mean a “thread”. During this document words “task” and “thread” are used to interchangeably. [↑](#footnote-ref-1)
2. Hint: you can use configTICK\_RATE\_HZ (defined in FreeRTOS.h) when defining the argument. [↑](#footnote-ref-2)
3. The microcontroller has four energy friendly modes from EM1 to EM4 (the mode having a greater number represent a deeper sleep). In mode EM1 only the instruction executing engine goes to sleep (by disabling its clock signal), other peripheral components are still running. [↑](#footnote-ref-3)