Exercises and report template

Investigating the embedded operating system FreeRTOS (2)

|  |
| --- |
| **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 each laboratory a little test should be accomplished. To pass this test you should be clear with the exercises and the theoretical background of them. |

# Using a semaphore to synchronize execution of threads

|  |
| --- |
| In the second exercise we saw that FreeRTOS is capable of scheduling two independent threads. However, in most of the cases threads are not independent. One of them may do some action, generate some data that is required by the other. Eventually we can say that one of them generates events and the other awaits for that event. This way these two threads synchronize their execution to each other.  The event has to be signaled somehow. Operating systems provide various mechanisms to do that. If a thread waits for only one event The aim of this exercise is to present another common use case for semaphores: synchronize the execution of threads.  Notes:   * We need two tasks to solve this exercise:   + One of them is generating the events.   + The other is awaiting for these events. * Use a binary semaphore to signal the events! * Before we could use the semaphore we need to create it first. For this use the API function named **xSemaphoreCreateBinary()** (the other possible API function has been marked as deprecated, see **semphr.h**)! * There are more than one API calls to take a semaphore, and there are also more than one to release a semaphore. Use these:   + **xSemaphoreTake()[[1]](#footnote-1)**   + **xSemaphoreGive()** * Solve this exercise by generating the events once in a second (set USING\_BUTTONS to 0)! * \* Solve this exercise by generating the events only when push button[[2]](#footnote-2) PB0 has been pressed[[3]](#footnote-3) (set USING\_BUTTONS to 1)! (Hint: we do not want the task that checks the button’s state to starve the other task. To avoid this (and also to debounce the buttons) it is required to delay the execution of the task a bit (e.g. for 10 ms).) * Fill in the file **04\_synchronization\_with\_a\_semaphore.c** at places marked as “TODO”! |
| Place a screenshot of the terminal showing the output generated by the application (for the case when events are generated once in every second): |
|  |
| \* Show to the lecturer that the events are generated if (and only if) PB0 is pressed! |

# \* Sending messages between threads (using message queues)

|  |
| --- |
| In the previous exercise we have seen that semaphores can be used for signaling events. If we want to convey more information between two tasks than the mere occurrence of an event then we need a so called message box or mailbox. By using these we can also send an arbitrary data. In many cases more than one mailboxes are connected together in a chain (thus forming a message queue). This can prevent message loss if ­– for short periods of time – the messages are generated faster than they are processed. The FreeRTOS only implements queues. However a queue with a length of 1 is basically equals to a mailbox.  Notes:   * This exercise has a lot in common with the previous one but also with some differences, of course:   + We are about to use a queue (length 1) instead of a semaphore to accomplish communication between the threads.   + While a semaphore can signal only the mere occurrence of the event, now we can also send a message (a value counting the number of events happened so far).   + Fill in the file **05\_synchronization\_with\_a\_queue.c** at places marked as “TODO”! |
| Place a screenshot of the terminal showing the output generated by the application (for the case when messages are sent once in every second): |
|  |

# The problem with shared resources

## Presenting the problem

|  |
| --- |
| The aim of this exercise is to present the problem with the handling of shared resources by using the standard output from more than one tasks. To make this problem frequent enough for observation we have to generate artificial circumstances. It is important to note that these circumstances only make the rate at which problem appears higher. The problem happily can happen also under normal circumstances (although with lower rate of appearance).  Notes:   * Solve this exercise by implementing two tasks! Both of them should have endless-loop structure! * Both tasks print a message to the standard output. * The low priority task should print a single (but very long) line periodically in each second (no matter the content, but at the end place a new line ‘\n’ and a carriage return ‘\r’ characters). * The high priority task should print only a carriage return ‘\r’ character periodically. The period should be very little compared to a second and preferably shouldn’t be an integer divisor of one second (for example 9 ms): * Fill in the file **06\_shared\_resources.c** at places marked as “TODO”! |
| Place a screenshot of the terminal showing the output generated by the application: |
|  |
| Explain the results! |
|  |

## Solving the problem by using a mutex

|  |
| --- |
| The aim of this exercise is to avoid the problem with shared resources presented in the previous exercise by using a so called mutex (MUTually EXclusive Semaphore). Although we could use a traditional binary semaphore for this purpose, the usage of mutexes to protect shared resources is preferred as mutexes have some useful properties (not present in semaphores):   * A mutex can be released (given) by only the task that has previously has acquired (taken) it. While a semaphore can be released by any task. This behavior is not a bug as it is required if a task uses a semaphore to signal an event to another task. * Mutexes are created as free by default while semaphores are initially taken. *This statement is true for FreeRTOS, other OSes may initialize the semaphore in a different way (like taking an argument telling if the semaphore should be initialized as free or acquired).* * Mutexes are utilizing some special algorithms against priority inversion (see exercise 7).   Notes:   * Start from the solution for the previous exercise! * Use a mutex to protect the critical sections (i.e. those parts of the code that accesses the shared resource (i.e. the serial interface used as the standard output)). * Before we could use the mutex we need to create it. For this use the API function named **xSemaphoreCreateMutex()**[[4]](#footnote-4)! * To acquire or free up a mutex the same API calls can be used as with binary semaphores. |
| Place a screenshot of the terminal showing the output generated by the application: |
|  |
| What happens when the lower priority task still owns the mutex and the higher priority task has just awakened? *(It is not enough to answer that “The higher priority task is not able to access the resource”. We are curious what events are following each other which eventually results in that the higher priority task is unable to access the resource.)* |
|  |

# Priority inversion

## Presenting the problem

|  |
| --- |
| We have discussed in the previous exercise that if we use a semaphore to protect shared resources then priority inversion can occur as an unwanted side effect. Let’s see an example for this!  Like presenting the problem of shared resources we have to create artificial circumstances to present priority inversion (however it is also true that without these artificial circumstances the problem can also happily occur).  Notes:   * At this point it is not needed to modify the code. The exercise is about to observe and explain the output generated by the application. * Use (without any modifications) the file **07\_priority\_inversion.c**! |
| Place a screenshot of the terminal showing the output generated by the application: |
|  |
| Explain the results! What happens in these steps? |
|  |
| What can be considered as an unexpected side effect? Why is it named as priority inversion? |
|  |

## Mitigating the problem

|  |
| --- |
| We can mitigate the problem of priority inversion if we increase the priority of the task currently owning a semaphore guarding a shared resource when a higher priority task also wants to get that semaphore.  There are multiple solutions (protocols) to implement this depending on when and to exactly what level are we increasing the priority of the task originally owning the semaphore. If a semaphore implements one of the solutions then it is called as a mutex (MUTually EXclusive semaphore). The name refers to the fact that these protocols are only needed if we are about to use a semaphore to implement mutual exclusion.  Notes:   * In this point we create the same artificial circumstances. But now we are using a mutex instead of an ordinary semaphore. * Start from the **solution of the previous exercise**. Modify it to create a mutex instead of an ordinary semaphore: **xSemaphoreCreateMutex()**. |
| Place a screenshot of the terminal showing the output generated by the application: |
|  |
| Explain the results! What happens in these steps? |
|  |
| What could be mitigated? |
|  |
| What side effect is still remained? |
|  |

# \* Signaling multiple events (event groups / event flags)

|  |
| --- |
| So far we have used semaphores and queues (mailboxes) for communication and synchronization between tasks. Because a queue can be imagined as chaining multiple mailboxes together, and a mailbox can be imagined as a semaphore with the addition of an arbitrary data it can be seen that these solutions are basically close relatives.  There is another, somewhat different, maybe a little bit less frequently used solution to synchronize the execution of tasks: event groups (or event flags).  The main difference compared to semaphores is that with event flags it is possible to wait for multiple events at once. It can also be configured if we would like to wait the OR or the AND combination of these events.  Notes:   * This exercise is consisted of three tasks:   + Two tasks are generating the events (two different one).   + The third is awaiting for these events. * Fill in the file **08\_event\_flags.c** at places marked as „TODO”! |
| Place a screenshot of the terminal showing the output generated by the application: |
|  |

# \* Tickless idle mode

|  |
| --- |
| Reducing power consumption in embedded systems is a crucial task. We have already seen an example for this in exercise 2.2 (first laboratory).  In this exercise we can reduce even further the power consumption by utilizing the tickles idle mode of FreeRTOS.  You are free to use the solution for exercise 2.2 as a template to start with (or create a new application if you prefer). |
| \* Explain the operation of tickles idle mode! How can this mode reduce even further the power usage? |
|  |
| \* Make an application demonstrating the tickles idle mode and show it to the lecturer! |
|  |

1. Hint: when we try to take a semaphore we can define a timeout parameter. If we want to wait as long the semaphore is locked (no matter how much time does it take) then we have to set this timeout value to **portMAX\_DELAY**! [↑](#footnote-ref-1)
2. Hint: push buttons are low active (e.g. a pressed button generates logical 0). [↑](#footnote-ref-2)
3. Under “has been pressed” we mean that in the previous cycle the button was not pressed but in the actual cycle it is pressed. (It is not equal that the button is currently pressed. That condition would generate a lot of consecutive events in every cycle as long as the button is pressed…) [↑](#footnote-ref-3)
4. This API call is implemented only if **configUSE\_MUTEXES** is set to 1 in **FreeRTOSConfig.h**! [↑](#footnote-ref-4)