

Lab #2-Introduction to queues and mutexes

Leonardo Fusser (1946995)

Objectives:

- Use queues in FreeRTOS.
- Use mutex and global shared data in FreeRTOS.
- Use functions to access mutex-protected shared data.
- Determine the tasks' best priority levels.
- Measure spare processing capacity using the Idle Task hook function.

Hardware: MPLAB X with simulator, Explorer 16/32.

To hand in:

These sheets including answers to all questions on **Teams**.

C code on **GitHub**:

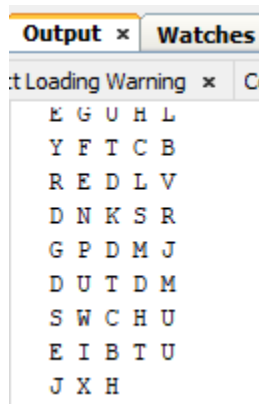
- In all source files:
 - All non-automatic variables must be declared static to make them local to the file, i.e., private.
 - All private variables must be accessible through public functions only. This way, global broadcast is replaced by public setter-getter functions.
 - All local functions must be declared static to make them local to the file, i.e., private.
 - A multiple file system.
- All initializing code must use functions. All functions must have their own headers.
- All files must have their headers: file name, description, date, author, and version history.
- All header (.h) files must be located inside a folder named "include".
- You must use names that are representative for all variables, macros, tasks, and functions.
- The software must respect the following qualities: modularity, maintainability (readability, understandability) and portability.
- All macros in capital letters.
- Use preprocessor instruction to enable/disable debugging.
- All diagrams.
- All tasks and functions must have a header (see figure1).

```
/****** Task 1******/
*
* Description:
*   This task ...
*
* Returns: NONE
*
***** /
```

Figure 1 task header example

Requirements:

1. A taskA must unblock taskB every four seconds using a queue. taskA fills a transmitting queue with a burst of 5 random capital letters every four seconds.
2. TaskB, when unblocked, must trickle all 5 letters one by one every 200mS or every 400mS.
3. TaskC changes the period to 200mS whenever PB1 is pressed-released and change it to 400mS whenever PB2 is pressed-released.
4. All shared data must be mutex-protected and must be accessed using re-entrant functions.
5. An IdleHook function must visually monitor the idle task spare time available: inside the IdleHook a message "Idle Hook" must be printed each time the function is called. To reduce the printing frequency, implement a skip counter that will print only when the task has run 100 000 times or so.
6. TaskA and taskB must be deterministic.



```
Output x Watches
Loading Warning x C
E G U H L
Y F T C B
R E D L V
D N K S R
G P D M J
D U T D M
S W C H U
E I B T U
J X H
```

Encapsulation:

You must encapsulate your project in a multi-source-file system.

Table 1: examples of file contents

Files	Content
initBoard.c	Includes its own header file: initBoard.h Contain all functions related to initializing the board: i.e. #pragma, initTimer(), initOscillator(), initIO(),...
taskA.c taskB.c taskC.c	Each task includes its own header file. A private taskX() A public startTasX() to create the task. External global variables are prohibited.
public.h	Includes device header file: xc.h Includes all dependencies and function prototypes. Includes all common macros.
initBoard.h	Includes device header file: xc.h Includes all dependencies and function prototypes.

Also:

- The queue's name must be suffixed with the file holding the queue (e.g., queueB).
- The file holding a queue, a mutex or a semaphore must be the receiving task file (e.g., taskB.c holds queueB queue).

Task diagram

Draw a task diagram by downloading *task_diagram_template.vsd* file.

Simulation mode

As a rule of thumb, in simulation mode, all delays must be divided by 4 to 10.

For example, you could setup a macro to enable/disable DEBUG mode:

```
#define      DEBUG
#ifdef      DEBUG
    #define DELAY_4SEC          1000
    #define DELAY_200MSEC      50
    #define DELAY_400MSEC      100
    #define IDLE_LOOP          10000
#else
    #define DELAY_4SEC          4000
    #define DELAY_200MSEC      200
    #define DELAY_400MSEC      400
    #define IDLE_LOOP          10000
#endif
```

Note: frequencies, periods and skip values might change depending on whether using the simulator or the target.

Lab work:

Use the same repo as the previous lab.

You must Exclude or Remove vTask1.c and vTask2.c files from the project.

Write taskA and taskB in simulation mode.

Idle Hook

The `vApplicationIdleHook()` function will execute whenever the kernel is not busy.

To enable this feature, `configUSE_IDLE_HOOK` must be set to 1 within `FreeRTOSConfig.h` for the idle hook function to get called.

The idle hook is implemented in the main source file.

Your system must have some spare processing capacity.

You must measure the spare processing capacity using the idle hook.

- Implement the IdleHook function to monitor spare processing capacity of the system. Every 100 000 calls to the hook function, a message is printed. Reminder: delays are divided by 4 to 10 in simulator mode.

The screenshot of figure 3 is an example of output on the simulator.

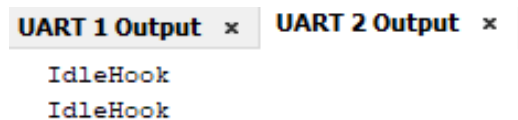


Figure 3 Simulating the idle hook being executed. Every time the Idle hook is called, a message is printed on Uart2.

Mutex-protected shared data

When finished debugging taskA and taskB, add taskC:

- a) With blocking delay:
 - Add taskC to change the period- you must use a mutex-protected shared variable accessible by setter-getter functions only. You must include a small blocking delay.

To simulate the BPs, implement the following stimuli:

	Asynchronous	Pin/Register Actions	Advanced Pin/Register	Clock Stimulus	Register Injection
Fire	Pin	Action	Value	Units	Comments
	RD6	Pulse Low		200 ms	long delay
	RD7	Pulse Low		200 ms	short delay

Since the simulator sets inputs to zero by default and the since all PBs are active low, you must simulate the pull-up resistors by setting all inputs to high at boot time:

Time	RD6	RD7	Click here to
0	1	1	

Does it work properly? Is it responsive? Explain.

- With taskC created and added to the project, the pushbuttons (in stimulus) work well and the system remains responsive when the printing delay in taskB is modified by the pushbuttons. The system also has some spare processing time since a message continuously prints from the idle hook.

This is because no task is hogging the system for the CPU. Up to this point, all three tasks have a blocking delay inside their task routines. This ensures that every task has a chance to use the CPU to perform some sort of action and it also ensures that the system has some spare processing capacity left.

b) Without blocking delay:

- Remove the blocking delay and test your program again.

Does it work properly? Is it responsive? Did you need to change the task priority? Explain.

- With the blocking delay removed in taskC, the pushbuttons are not as responsive when changing the printing delay in taskB, unlike before when the blocking delay was in place. Furthermore, the system no longer has any spare processing time because there is no more message being printed continuously from the idle hook.

This is because taskC is hogging the system for the CPU. Now, instead of entering a blocking state after checking the pushbutton statuses, taskC continuously runs and never really stops. This is possible because the task routine in taskC does not need any required resources, so it will always be in the ready state or running state. In turn, it will hog the system CPU.

From the previous tests, which scheme best works if one wants to have some spare processing capacity available? Explain.

- The scheme that works best if spare processing time is desired, is the one with blocking delays in place. This is simply because blocking delays ensure that no task hogs the system CPU.

This is because once a task has no longer the need to use the system CPU, the blocking delay will make the task “sleep” for a certain amount of time before the task is allowed to run again. In turn, this will free up the system CPU for another task to use, or it will allow for the CPU to be idle when no other task has to use the system it.

In short, there will be spare processing time left if blocking delays are used because of their behavioural nature.

Ask the teacher to verify the monitor spare processing capacity. If there is not enough spare processing capacity, you might have to modify your program.

Task priorities

Determine the tasks priority levels. Justify your answer.

TaskA:

- taskA should have the highest priority over the other two tasks. This is because taskA has to absolutely send a burst of data to a queue (5 randomly generated capital letters), and it should not be disturbed while doing so.

Any disruptions to this task while it is running might result in corrupted or incomplete outputs being produced on the console when taskB is running. This is simply because taskB would not receive the complete burst of data.

TaskB:

- taskB should have the 2nd highest priority over the last task. This is because it is responsible for printing the burst of data received from taskA (through a queue), and it also should not be disturbed while doing so.

Any disruptions to this task might also produce corrupted or incomplete outputs being produced on the console when it is running. This is simply because taskB might not have enough time to complete the printing operation for the received burst of data.

TaskC:

- taskC should have the lowest priority out of the other two tasks. This is because it has no crucial responsibilities like the other two tasks do; it only has to read the state of some pushbuttons and change a printing delay in taskB according to the pushbutton pressed.

Therefore, if this task would be disturbed, it will not result in any drastic effects done on the system.

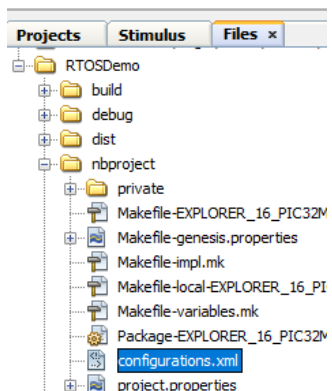
Test on the target

Comment out the code related to Uart1 and Uart2 and download the code to the board. The messages are printed to the LCD.

Give a demo to the teacher.

Comment and commit all files that were populated by you.

Also, commit the configurations.xml file:



Push the project to GitHub.

After lab questions (20% of the lab)

Q1- When taskC had a blocking delay. What was the impact on the system? Explain.

- In short, when taskC had a blocking delay implemented, the system was responsive and there was plenty of spare processing capacity left.

This was because blocking delays were used in taskC.

Refer to the lab sheets above for a more detailed explanation.

Q2- When taskC had no blocking delay. Was there enough spare processing capacity? Since it was non-blocking, what was the impact on the system? Explain.

- In short, when taskC had no blocking delay implemented, the system was no longer as responsive as before and there was no more spare processing capacity left; the system CPU was constantly hogged because of taskC.

This was because there were no more blocking delays used in taskC.

Refer to the lab sheets above for a more detailed explanation.

Q2b- For the following scenario:

- taskC has no blocking delay implemented and is at priority 1.
- taskA priority is at 3.
- taskB priority is at 2.

What are the possible states for the tasks (B, RU, RY)?

taskC

- Even with the lowest possible priority, the possible state for this task is RU (running) and RY (ready). This is because the absence of the blocking delay will make taskC hog the system CPU. This is due to the behavioural nature of the absence of blocking delays in task routines (explained previously in these lab sheets).

taskB

- The possible states for this task are all three: B (blocked), RU (running) and RY (ready). Due to the presence of this task, the OS scheduler will swap out taskC for this one when this task transitions from blocked to ready state (assuming it deems it appropriate to do so).

taskA

- The possible states for this task are also all three: B (blocked), RU (running) and RY (ready). Once again, due to the presence of this task, the OS scheduler will swap out taskC for this one when this task transitions from blocked to ready state (assuming it deems it appropriate to do so).

Q3- If taskC priority level was higher than all other tasks, would it still work?

- If taskC had a higher priority level than all the other tasks, not only will it still work, but it won't even let the other tasks ever entering the running state. This is because of the priority that it has set (> 3) and that it has no blocking delay implemented in its task routine.

Before, taskA and taskB had a chance of entering the running state because they had a higher priority than taskC, so taskC could never really prevent taskA and taskB from entering the running state. But now, since taskC has the higher priority than the rest, the OS scheduler will constantly favor taskC over the rest. taskA and taskB will no longer be ever able to run. Furthermore, there will be no spare processing time left since taskC will be hogging the system CPU.

Q4- What is the best scenario for taskC in terms of blocking delay, priority and processing spare capacity? Explain.

- The best scenario for taskC in terms of blocking delay, task priority and processing spare capacity is as follows:

taskC should have a blocking delay implemented to allow for the system to have spare processing time and to not hog the system CPU. Also, taskC should have the lowest priority since it has the least critical role out of all three tasks (as explained previously in these lab sheets).

Refer to the lab sheets above for a more detailed explanation.

Q5- From what you learned from this lab, what is the best scenario to use when polling inputs?

- In short, when polling inputs, the task that has the polling function should never hog the system CPU, so it should use blocking delays, and it should have the least priority among all other tasks present on the system. This is because tasks that are responsible for polling inputs generally do not have a critical roll in the way the system functions.

Q6- What is the relation between the **idle hook function** and the **idle task**.

- The idle hook function is the function that is called by the OS scheduler when there are no immediate tasks ready to take CPU running time. The idle task is the task that is implemented within the idle hook function, which is responsible for executing any idle related tasks. For example, like what was used in this lab, blinking an LED, or printing an idle message to a console.

Task diagram:

