



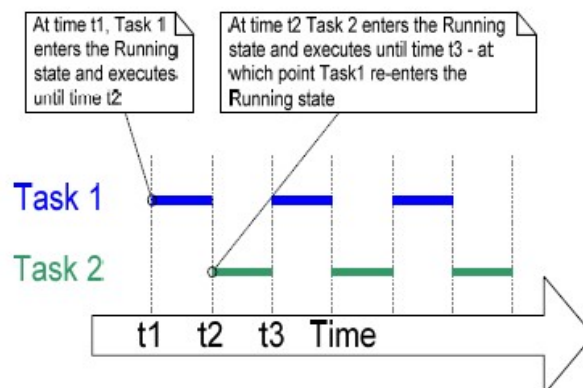
UNIVERSITY OF SAINT THOMAS – SAINT PAUL – MN
Electrical and Computer Engineering
ENGR 432: REAL TIME SYSTEMS
LAB6: STEP-BY-STEP WITH FREERTOS.
Spring 2023
INDIVIDUAL

PART 1: Creating tasks

This example demonstrates the steps of creating two simple tasks and then starting tasks executing. The tasks simply print out a string periodically, using a null loop to create a period delay. Both tasks are created at the same priority and are identical except for the string that they print.

1. First, go through three functions *main()*, *vTask1()* and *vTask2()* in the file *main.c*, and pay attention to following points.
 - In the task function *vTask1* (or *vTask2*), there is an infinite *for(;;)* loop. All computations performed by a task should be implemented within this loop.
 - The *main()* function creates two tasks before starting the scheduler.
 - Study the usage of *xTaskCreate()* function which creates new task instances to be scheduled. This library function requires six input parameters.
Explain the meaning of each of six arguments passed to the this function when creating two task instances *vTask1* and *vTask2* in the *main()* function.
2. Select F7 to build the project; then, select Ctrl+F5 (or click icon  on the tool bar) to start debugging session.
3. Open the *Debug (printf) Viewer* window if it has not been opened by selecting *View > Serial Windows > Debug (printf) Viewer*.
4. Run the project by selecting F5 (or clicking icon ).

Both *vTask1* and *vTask2* are running at the same priority, and so share CPU time on the single processor. Their actual execution pattern should be as the following.



The arrow along the bottom of the above figure shows the passing of time from time t1 onwards. The colored lines show which task is running at each point in time.

It is also possible to create a task within another task. You could have created Task 1 from `main()`, and then created Task 2 from within Task 1.

- Move the statement of creating Task 2 instance in `main()` into Task 1 immediately before the infinite *for(;;)* loop.
- Rebuild and execute the project.

Thus, Task 2 would not get created until the scheduler had been started. Please compare the output produced by the modified example with the one from the original example and check if they would be same.

PART 2: Using the task parameter

The two tasks *vTask1* and *vTask2* created in PART 1 are almost identical, except the text string they print out. We will create a single task implementation and use two instances of this same function to remove the duplication. The parameter *pvParameters* of a task function is used to pass into each task instance the string that it should print out.

1. Define two global (i.e. outside any function) string variables that will be passed in as the task parameters. They are created as *const char ** to ensure they are not allocated in the stack and will remain valid when the tasks are executing.
2. Create a new task function named as *vTaskFunction*. The easiest way is to modify the original definition of *vTask1* instead of coding from scratch.
3. Inside *vTaskFunction*, create a variable *pcTaskName* of type *char ** without the value initialization to hold the task name.
4. Add a statement to assign the function input parameter *pvParameters* to *pcTaskName*. Remember cast the type from *void ** to *char **.
5. In the *main* function, modify the two *xTaskCreate()* function calls.
 - Replace the first parameter *vTask1* (or *vTask2*) with the newly created function name *vTaskFunction* from step 2;
 - Replace the 4th parameter *NULL* with the corresponding string variables defined earlier.
6. Same as the steps 2 – 4 in PART 1.

PART 3: Experimenting with priorities

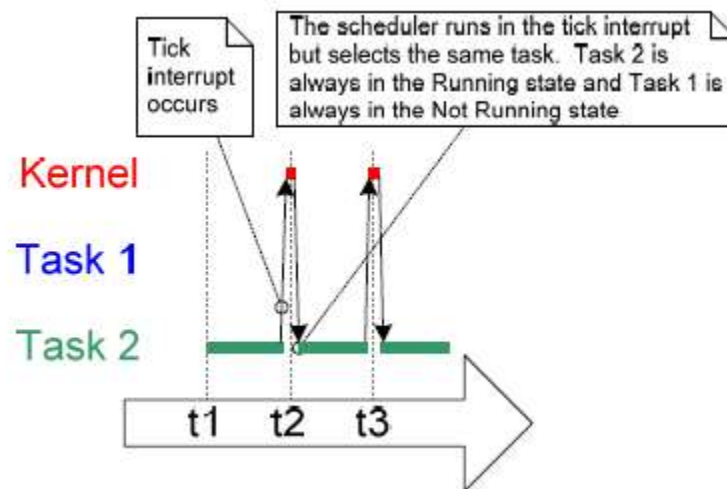
In the PARTs 1 and 2, two tasks have been created at the same priority, so both entered and exited the Running state in turn. This example looks at what happens when we change the priority of one of the two tasks created in PART 2.

- In the main() function, assign the priority of the first task instance at 1, and the priority of the second task instance at 2.

The single function vTaskFunction() that implements both tasks remain the same: it still simply prints out a string periodically using a null *for* loop to create a delay.

- Rebuild and execute the project.

The following figure shows the expected execution sequence.



PART 4: Using the Blocked state to create a delay

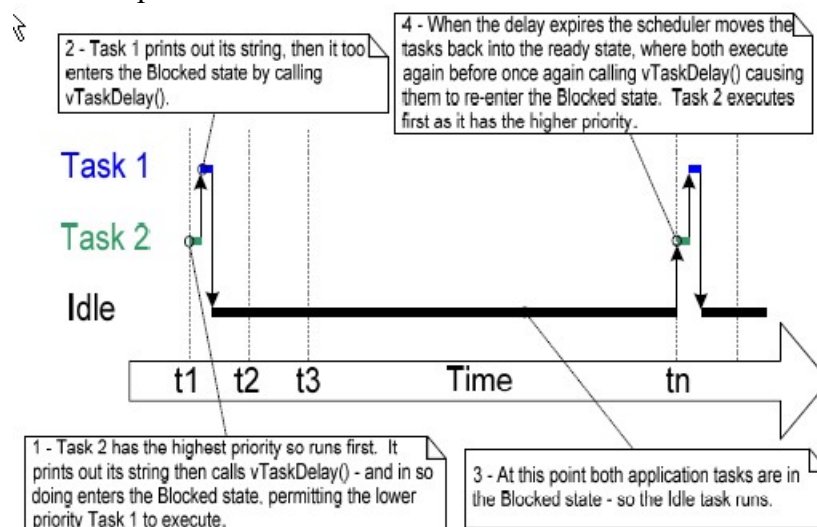
All the tasks created so far have been ‘periodic’ – they have delayed for a period and print out their string, before delaying once more, and so on. So the task effectively polled an incrementing loop counter using a null loop until it reached a fixed value. While executing the null loop, the task remained in the Ready state, starving the other tasks with lower priority.

This example, we will replace the polling null loop with a call to the `vTaskDelay(portTickType xTicksToDelay)` library function. It places the calling task into the Blocked state for a fixed number of tick interrupts. While in the Blocked state, the task does not use any processing time. Thus, the processing time is consumed only when there is work to be done.

The input parameter `xTicksToDelay` is the number of tick interrupts that the calling task should remain in the blocked state before being transitioned back into the Ready state. For example, if a task called `vTaskDelay(100)` while the tick count was **10,000**, then it would immediately enter the blocked state and remain there until the tick count reached **10,100**.

1. Open the `FreeRTOSConfig.h` header file in the PART 4 directory, set the macro definition of `INCLUDE_vTaskDelay` to **1**. This enables compiling the `vTaskDelay()` library function.
2. Open the `main.c` file, edit the infinite `for (;)` loop body within the task function `vTaskFunction()`.
 - Remove the null for loop for generating delay and all related variables:
`for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++)`
 - Call the `vTaskDelay()` library function following the statement `vPrintString();`. The input parameter is given as `“250/portTICK_RATE_MS”`. The constant `portTICK_RATE_MS` can be used to convert milliseconds into ticks.
3. Now, build and run the project. You could observe that both tasks run even though they are created at different priorities.

The expected execution pattern is shown below.



Part 5. Converting the example tasks to use *vTaskDelayUntil()*

The two tasks created in PART 4 are periodic tasks, but using *vTaskDelay()* does not guarantee that the frequency at which they run is fixed, as the time at which the tasks leave the Blocked state is relative to when they call *vTaskDelay()*. We solve this potential problem by using the *vTaskDelayUntil()* library function.

```
void vTaskDelayUntil( portTickType * pxPreviousWakeTime, portTickType xTimeIncrement );
```

Listing 13. *vTaskDelayUntil()* API function prototype

Table 4. *vTaskDelayUntil()* parameters

Parameter Name	Description
<code>pxPreviousWakeTime</code>	<p>This parameter is named on the assumption that <i>vTaskDelayUntil()</i> is being used to implement a task that executes periodically and with a fixed frequency. In this case <code>pxPreviousWakeTime</code> holds the time at which the task last left the Blocked state (was ‘woken’ up). This time is used as a reference point to calculate the time at which the task should next leave the Blocked state.</p> <p>The variable pointed to by <code>pxPreviousWakeTime</code> is updated automatically within the <i>vTaskDelayUntil()</i> function; it would not normally be modified by the application code, other than when the variable is first initialized. Listing 14 demonstrates how the initialization is performed.</p>
<code>xTimeIncrement</code>	<p>This parameter is also named on the assumption that <i>vTaskDelayUntil()</i> is being used to implement a task that executes periodically and with a fixed frequency—the frequency being set by the <code>xTimeIncrement</code> value.</p> <p><code>xTimeIncrement</code> is specified in ‘ticks’. The constant <code>portTICK_RATE_MS</code> can be used to convert milliseconds to ticks.</p>

1. Open the *FreeRTOSConfig.h* header file in the Example 5 directory, and set the macro definition of *INCLUDE_vTaskDelayUntil* to **1**. This enables compiling the *vTaskDelayUntil()* library function.
2. Open the *main.c* file and edit the task function *vTaskFunction()*.
 - Declare a new local variable *xLastWakeTime* with the type *portTickType*;

- Initialize the variable *xLastWakeTime* with the current tick count. Note that this is the only time we access this variable. This tick count could be obtained by calling the library function *xTaskGetTickCount()*.

From this point on, *xLastWakeTime* is managed automatically by the *vTaskDelayUntil()* function.

- Inside the infinite *for(;;)* loop, call *vTaskDelayUntil()* library function instead of *vTaskDelay()* to generate delay.

vTaskDelayUntil() requires two parameters. You need to pass the pointer of the variable *xLastWakeTime* (i.e., *&xLastWakeTime*) as the first parameter, then the second parameter *250/portTICK_RATE_MS*.

3. Build and debug example 5.

The output produced is same as the output from PART 4.

Part 6. Combining block and non-blocking tasks

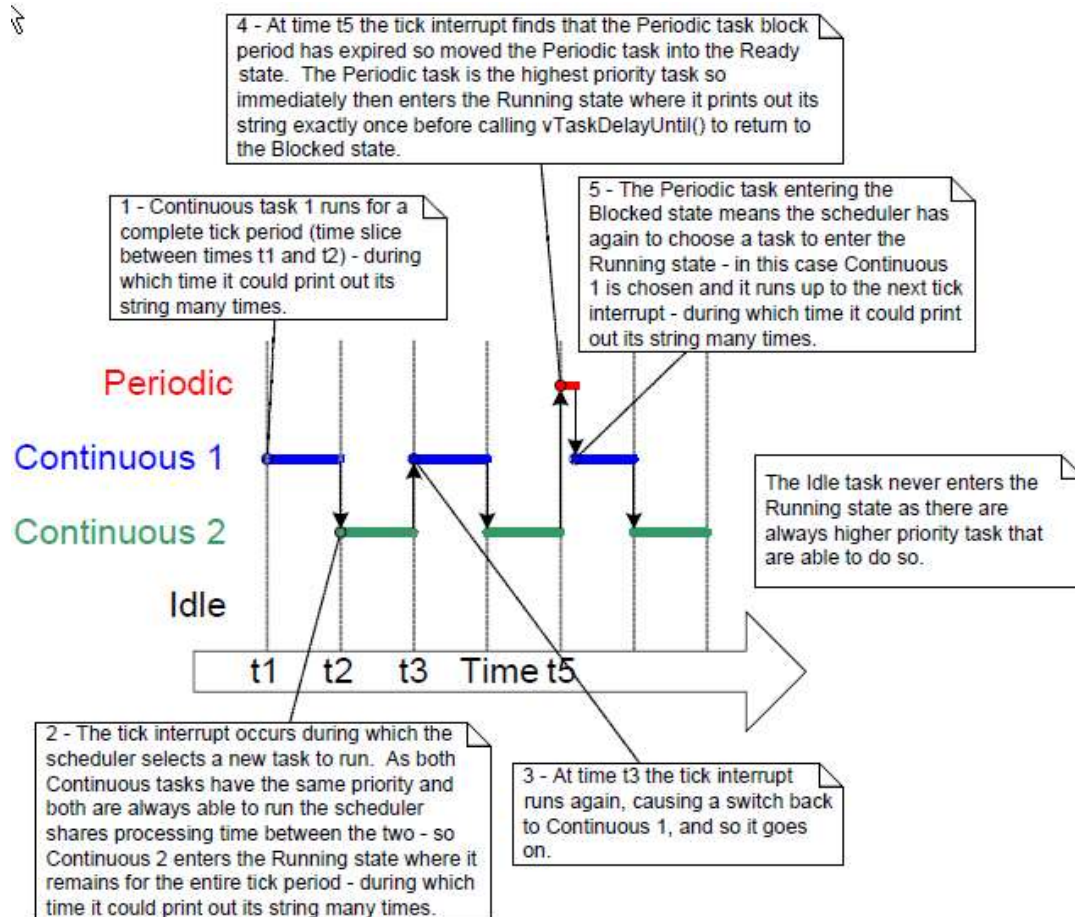
Previous examples have examined the behavior of both polling and blocking tasks in isolation. This example aims at demonstrating an execution sequence when two schemes are combined as follows.

1. Two tasks are created in priority 1. These do nothing other than continuously print out a string.
These tasks never make any library function calls that could cause them to enter the Blocked state, so are always in either the Ready or the Running state. Tasks of this nature are called *continuous processing* tasks as they always have work to do.
2. A third task is created at priority 2. It just prints out a string periodically, so uses the *vTaskDelayUntil()* library function to place itself into the Blocked state between each print iteration.

You need to modify the main.c() file as the following steps.

1. Create one task function for two continuous processing task instances named as *vContinuousProcessingTask()*. It includes a null loop delay its infinite *for(; ;)* loop as the vTask1 (or vTask2) in Example 1.
2. Create another task function for the third periodic task named as *vPeriodicTask()*. Within its infinite *for(; ;)* loop, call the *vTaskDelayUntil()* library function to create the fixed-time delay as in the *vTaskFunction()* function in Example 5.
3. In the main() function, create two tasks instances of *vContinuousProcessingTask()* at priority 1 and one task instance of *vPeriodicTask()* at priority 2.
4. Build and debug the project.

The execution pattern is expected to be as shown below.



Upload to Canvas your project code. Also upload a detailed report including all the necessary steps and pictures to redo your work and snapshots of your experiment results along with your thoughts and analysis including any problems you faced and how you managed to solve them.

Upload your files by Friday April 14.