Operating Systems And Applications For Embedded Systems

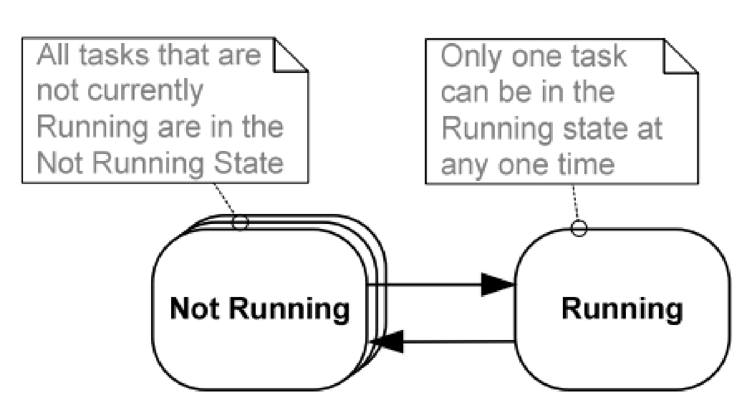
FreeRTOS

KATEDRA INŻYNIERII KOMPUTEROWEJ

Plan

- FreeRTOS
  - TOP LEVEL TASK STATES
  - Creating Tasks
  - The actual execution pattern of the two tasks
  - Tick interrupt executing
  - The execution pattern when one task has a higher priority than the other
  - Full task state machine
  - The execution sequence when the tasks use vTaskDelay() in place of the NULL loop
  - The execution pattern with periodic task
  - The sequence of task execution without idle state
  - The execution sequence with task deleting
  - Execution pattern with pre-emption points highlighted
- Interrupt Management
  - Interrupt example
- Memory Management
  - RAM allocation

TOP LEVEL TASK STATES


All tasks that are not currently Running are in the Not Running State
Only one task can be in the Running state at any one time
Not Running
Running

Creating Tasks I

Listing 1: Listing

```c
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile unsigned long ul;
    /* As per most tasks, this task is implemented in an infinite loop.
    for( ;; )
    {
    /* Print out the name of this task. */
        vPrintString( pcTaskName );
    /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
        /* This loop is just a very crude delay implementation. There i
```

Creating Tasks II

```c
                nothing to do in here. Later examples will replace this crude
                loop with a proper delay/sleep function. */
            }
        }
}
void vTask2( void *pvParameters )
{
        const char *pcTaskName = "Task 2 is running\r\n";
        volatile unsigned long ul;
        /* As per most tasks, this task is implemented in an infinite loop.
        for( ;; )
        {
                /* Print out the name of this task. */
                vPrintString( pcTaskName );
                /* Delay for a period. */
```

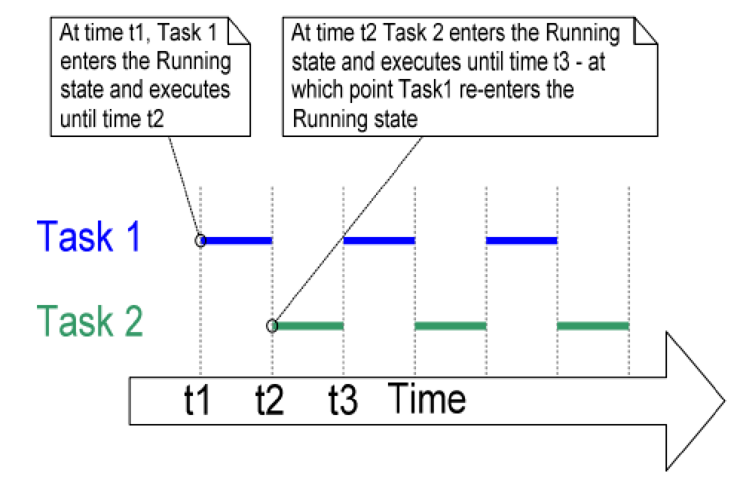
Creating Tasks III

```
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
        /* This loop is just a very crude delay implementation. There i
        nothing to do in here. Later examples will replace this crude
        loop with a proper delay/sleep function. */
        }
    }
}
int main( void )
{
    /* Create one of the two tasks. Note that a real application should
    the return value of the xTaskCreate() call to ensure the task was c
    successfully. */
    xTaskCreate( vTask1, /* Pointer to the function that implements the
                    "Task 1",/* Text name for the task. This is to faci
```

Creating Tasks IV

```
                            only. */
                            1000, /* Stack depth - most small microcontrollers
                            less stack than this. */
                            NULL, /* We are not using the task parameter. */
                            1, /* This task will run at priority 1. */
                            NULL ); /* We are not going to use the task handle.
    /* Create the other task in exactly the same way and at the same pr
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );
    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
    /* If all is well then main() will never reach here as the schedule
    now be running the tasks. If main() does reach here then it is like
    there was insufficient heap memory available for the idle task to b
    for( ;; );
}
```

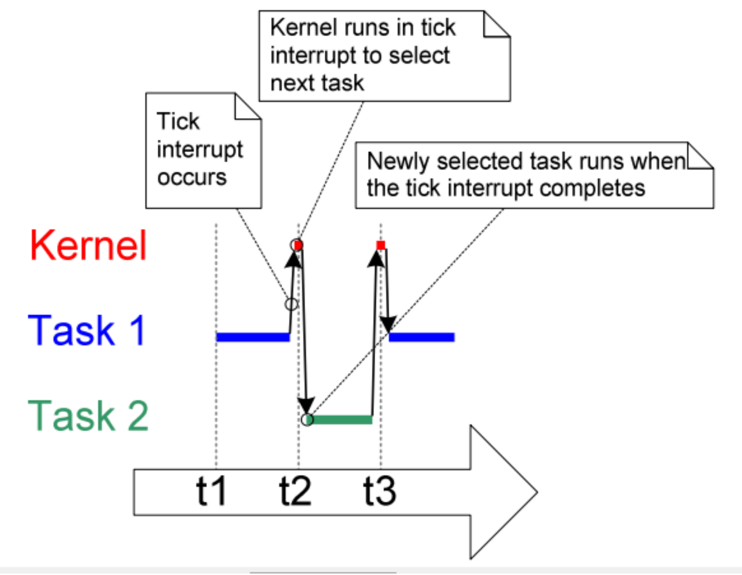
The actual execution pattern of the two tasks



At time t1, Task 1 enters the Running state and executes until time t2

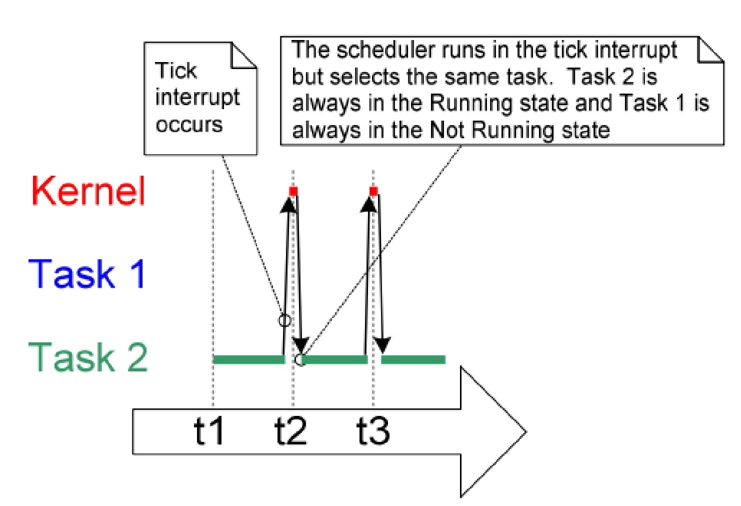
At time t2 Task 2 enters the Running state and executes until time t3 - at which point Task1 re-enters the Running state

Tick interrupt executing



Kernel runs in tick interrupt to select next task
Tick interrupt occurs
Newly selected task runs when the tick interrupt completes
Kernel
Task 1
Task 2
t1
t2
t3

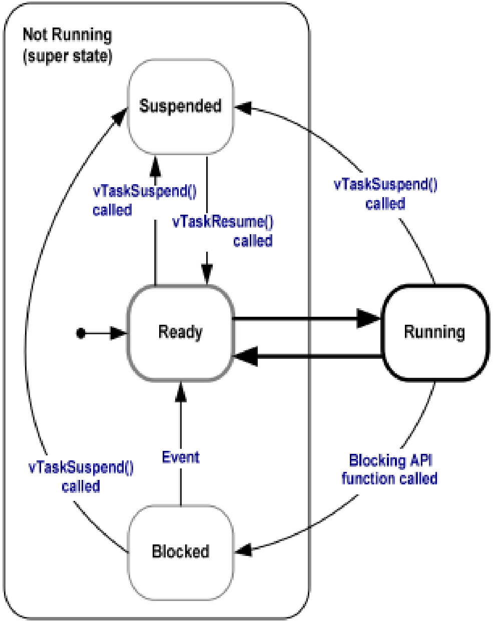
The execution pattern when one task has a higher priority than the other



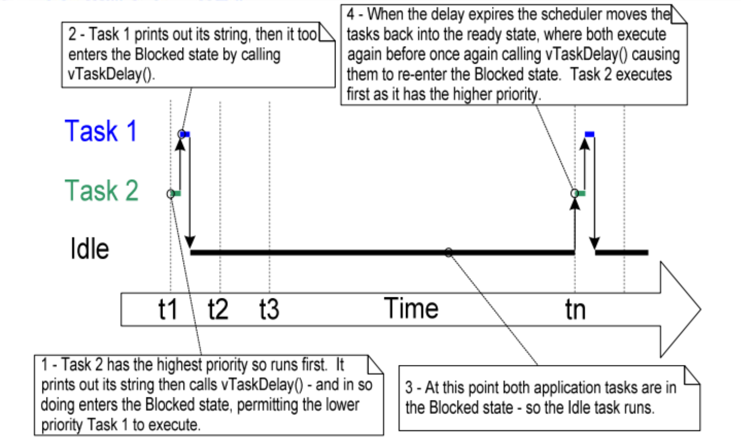
Tick interrupt occurs

The scheduler runs in the tick interrupt but selects the same task. Task 2 is always in the Running state and Task 1 is always in the Not Running state

Full task state machine



The execution sequence when the tasks use vTaskDelay() in place of the NULL loop



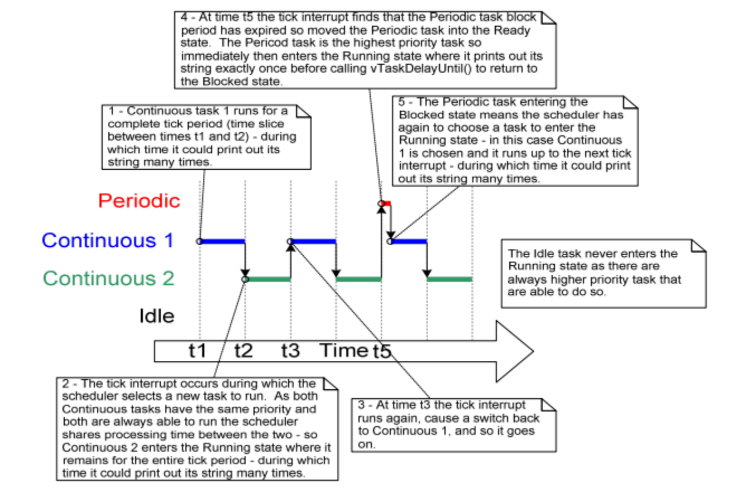
1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.

2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().

3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.

The execution pattern with periodic task



4 - At time t5 the tick interrupt finds that the Periodic task block period has expired so moved the Periodic task into the Ready state. The Pericod task is the highest priority task so immediately then enters the Running state where it prints out its string exactly once before calling vTaskDelayUntil() to return to the Blocked state.

1 - Continuous task 1 runs for a complete tick period (time slice between times t1 and t2) - during which time it could print out its string many times.

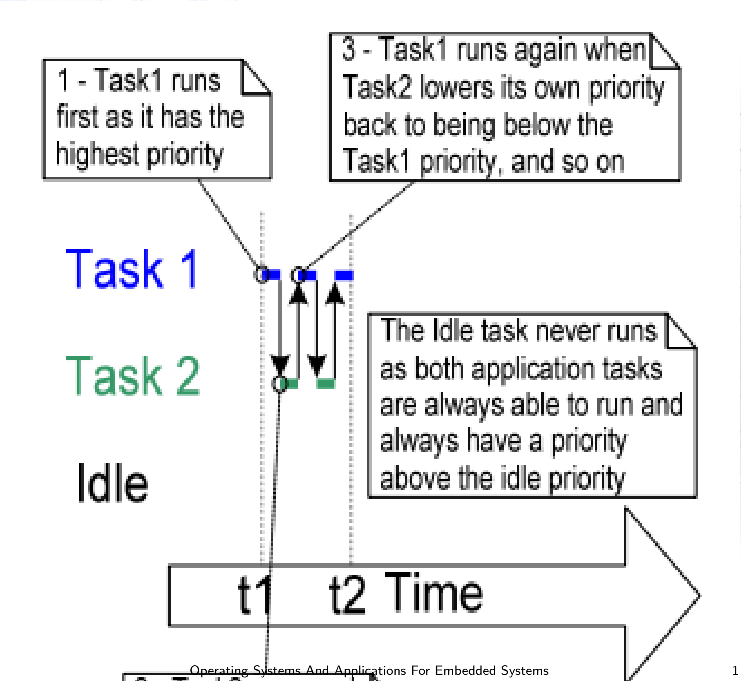
5 - The Periodic task entering the Blocked state means the scheduler has again to choose a task to enter the Running state - in this case Continuous 1 is chosen and it runs up to the next tick interrupt - during which time it could print out its string many times.

The Idle task never enters the Running state as there are always higher priority task that are able to do so.

2 - The tick interrupt occurs during which the scheduler selects a new task to run. As both Continuous tasks have the same priority and both are always able to run the scheduler shares processing time between the two - so Continuous 2 enters the Running state where it remains for the entire tick period - during which time it could print out its string many times.

3 - At time t3 the tick interrupt runs again, cause a switch back to Continuous 1, and so it goes on.

The sequence of task execution without idle state

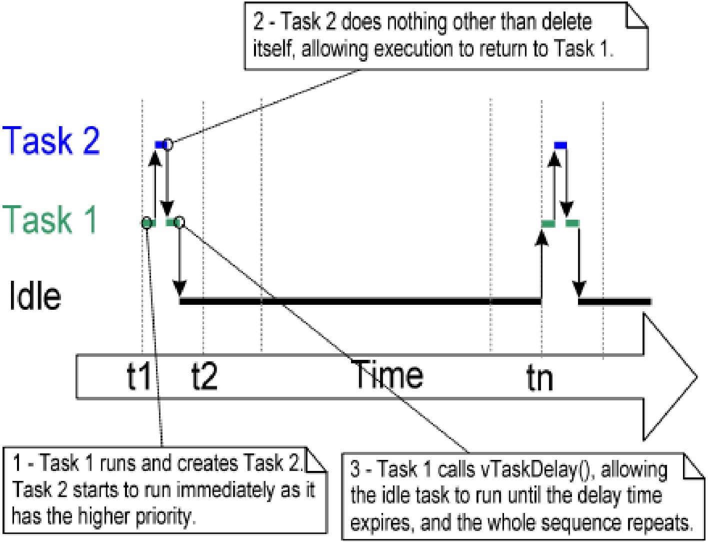


1 - Task1 runs first as it has the highest priority

3 - Task1 runs again when Task2 lowers its own priority back to being below the Task1 priority, and so on

The Idle task never runs as both application tasks are always able to run and always have a priority above the idle priority

The execution sequence with task deleting

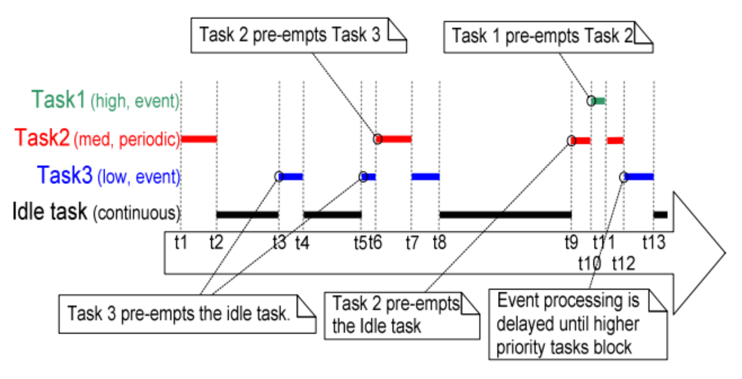


2 - Task 2 does nothing other than delete itself, allowing execution to return to Task 1.

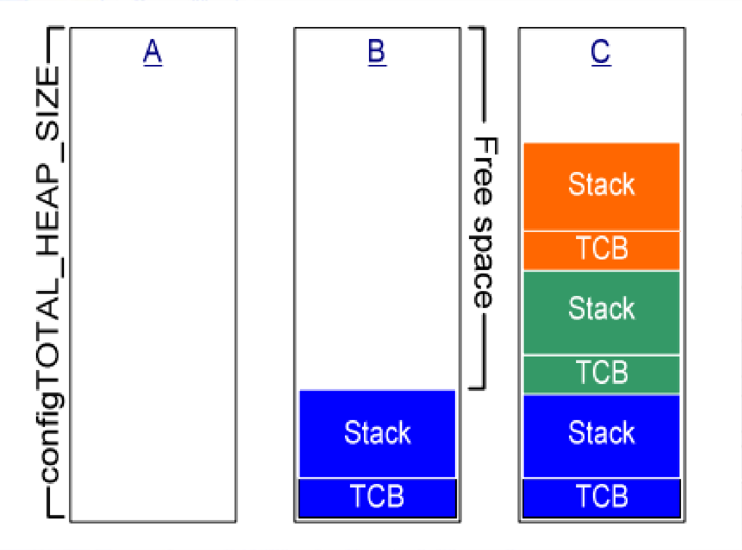
1 - Task 1 runs and creates Task 2. Task 2 starts to run immediately as it has the higher priority.

3 - Task 1 calls vTaskDelay(), allowing the idle task to run until the delay time expires, and the whole sequence repeats.

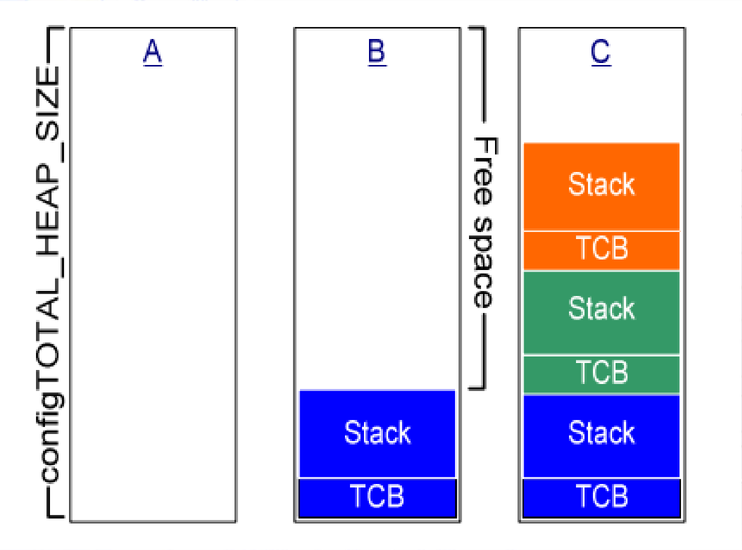
Execution pattern with pre-emption points highlighted



Interrupt example



RAM allocation



References

R. Barry.
Using the FreeRTOS Real Time Kernel: A Practical Guide.
Real Time Engineers Limited, 2010.

The End