Assignment (2) Simple Kernel Report

Program: Senior 2 CSE

Name: Abdelrahman Sherif Fayez Aly Saleh

ID: 1900773

Email: 1900773@eng.asu.edu.eg

Ain Shams University
Faculty of Engineering
2024

Abstract

In this assignment, I have implemented a simple kernel that uses Round Robin Scheduling Algorithm to multi-task two periodic tasks which are Green_Led_TaskA and Blue_Led_TaskB as is they were running simultaneously by giving each task a systick time slice after which the Os scheduler is called to schedule the next Task.

All Green_Led_TaskA does is that it turns on the green led on tiva-c board for some delay then it turns it off for some delay and loop again, similarly Blue_Led_TaskB but with the blue led.

Also, we called the tasks periodic as I implemented an Os_delay api for the delay of the task, in which it changes the state of the task as blocked (not ready) and calls the scheduler to schedule the next task automatically, and when all tasks are blocked it schedules an idle task which turns on and off the red led then saves the CPU clock cycles and power by turning it to the power saving mode until an interrupt comes using WFI instruction.

Explanation with Scenario

1) The application tasks:

- ➔ Each Task is allocated a private stack for it and an OSTask C-struct which have the resources a task needs as a pointer to its stack and a timeout integer that contains the number of ticks the task should wait for blocked until it will be in ready state again.

- ➔ And the functionality of each task is as explained above.

```c
uint32_t stack_TaskA[40];
OSTask TaskA;
void Green_Led_TaskA() {
    while (1) {
        BSP_ledGreenOn();
        OS_delay(BSP_TICKS_PER_SEC / 4U);
        BSP_ledGreenOff();
        OS_delay(BSP_TICKS_PER_SEC * 3U / 4U);
    }
}

uint32_t stack_TaskB[40];
OSTask TaskB;
void Blue_Led_TaskB() {
    while (1) {
        BSP_ledBlueOn();
        OS_delay(BSP_TICKS_PER_SEC / 2U);
        BSP_ledBlueOff();
        OS_delay(BSP_TICKS_PER_SEC / 3U);
    }
}
```

2) main application code:

- ➔ At first it calls some initializations to enable GPIO port F and its configurations through BSP_init().
- ➔ Then initializes the kernel by defining the idle task to be the background task if no task is ready and setting the priority of the PendSV interrupt to be the lowest in the system through OS_init().
- ➔ Then Starts initializing each of the 2 tasks in the kernel.
- ➔ And finally, transfer control to the RTOS to run the tasks.

```c
int main() {
    BSP_init();
    OS_init(stack_idletask, sizeof(stack_idletask));

    /* start TaskA */
    OSTask_start(&TaskA,
                 &Green_Led_TaskA,
                 stack_TaskA, sizeof(stack_TaskA));

    /* start TaskB */
    OSTask_start(&TaskB,
                 &Blue_Led_TaskB,
                 stack_TaskB, sizeof(stack_TaskB));


    /* transfer control to the RTOS to run the tasks */
    OS_run();

    //Code shouldn't reach here
    while(1){};
}
```

3) Helper structures in the Kernel

- ➔ OS_curr and OS_next: used to know which task is currently running and which is scheduled to be run.
- ➔ OS_tasks: an array of OSTask that keeps inside all started tasks so far.
- ➔ OS_tasksNum: to track the number of tasks started so far.
- ➔ OS_currIdx: to track which task the order is on for the scheduler to see if it can be scheduled or not.
- ➔ OS_readySet: 32 bit-mask to tell if each started task is ready (bit = 1) or is blocked (bit = 0) for the scheduler to know if he should schedule it or skip it.

board_support_package.c | board_support_package.h | kernel.c

```c
#include <stdint.h>
#include "kernel.h"
#include "qassert.h"

#define PENDSVREG *(uint32_t volatile *)0xE000ED04

Q_DEFINE_THIS_FILE

OSTask * volatile OS_curr; /* pointer to the current task */
OSTask * volatile OS_next; /* pointer to the next task to run */

OSTask *OS_tasks[32 + 1]; /* array of tasks started so far */
uint8_t OS_tasksNum; /* number of tasks started so far */
uint8_t OS_currIdx; /* current task index for round robin scheduling */
uint32_t OS_readySet; /* bitmask of tasks that are ready to run */

OSTask idletask;
void main_idletask() {
    while (1) {
        OS_onIdle();
    }
}

void OS_init(void *stkSto, uint32_t stkSize) {
    /* set the PendSV interrupt priority to the lowest level 0xFF */
    *(uint32_t volatile *)0xE000ED20 |= (0xFFU << 16);
```

4) Scenario of running the Application

➔ At the first breakpoint OS_run() is called from main

```c
}

uint32_t stack_idletask[40];

int main() {
    BSP_init();
    OS_init(stack_idletask, sizeof(stack_idletask));

    /* start TaskA */
    OSTask_start(&TaskA,
                 &Green_Led_TaskA,
                 stack_TaskA, sizeof(stack_TaskA))

    /* start TaskB */
    OSTask_start(&TaskB,
                 &Blue_Led_TaskB,
                 stack_TaskB, sizeof(stack_TaskB))


    /* transfer control to the RTOS to run the tasks
    OS_run();

    //Code shouldn't reach here
    while(1){};
}
```

➔ In OS_run(), after we configure the Systick Interrupt in the System with the highest priority, we call the scheduler to run the started tasks.

```c
void OS_run(void) {
    /* callback to configure and start interrupts */
    OS_onStartup();

    __asm volatile ("cpsid i");
    OS_sched();
    __asm volatile ("cpsie i");

    /* the following code should never execute */
    Q_ERROR();
}
```

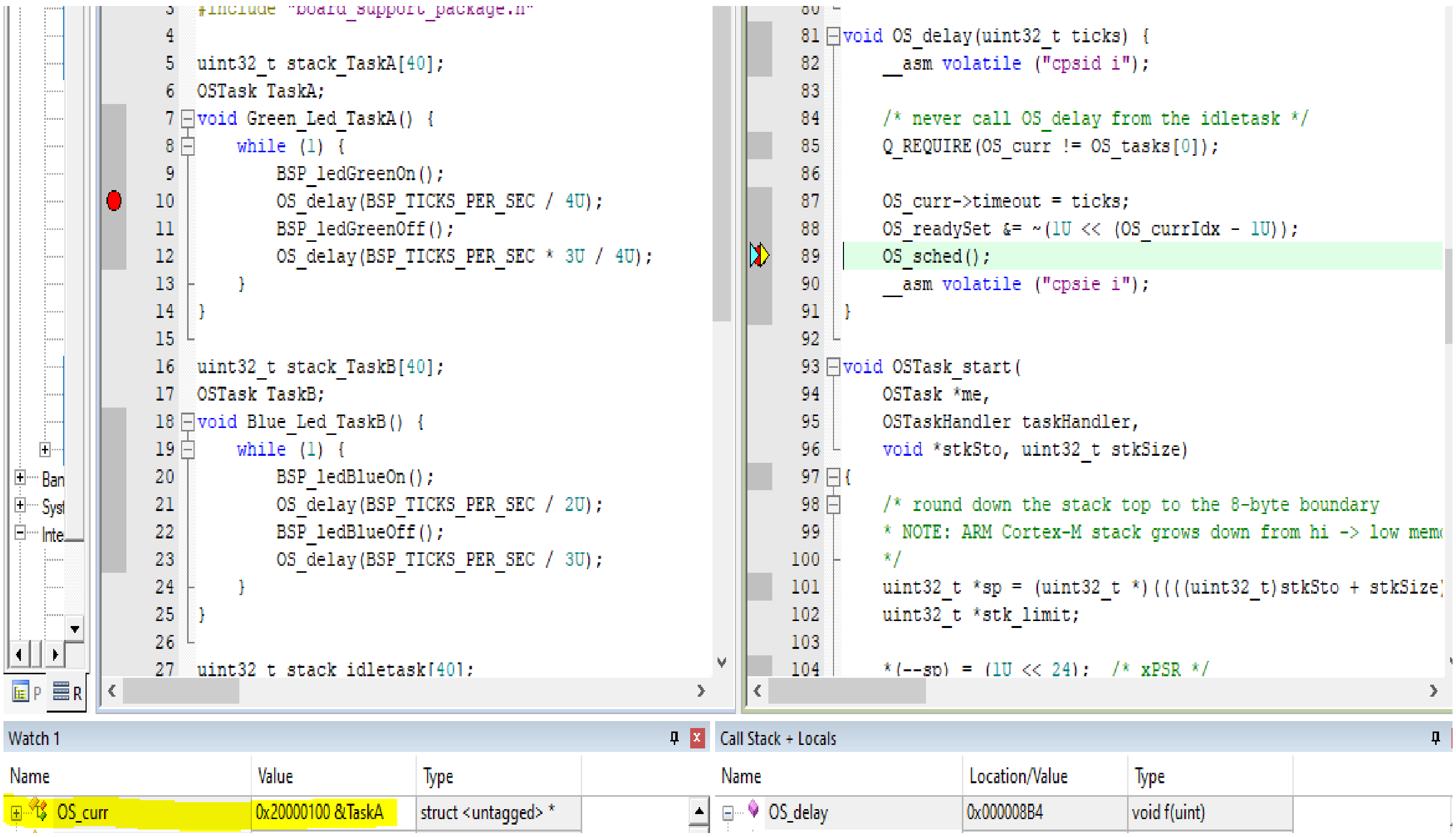
➔ The Scheduler will trigger PendSV as there will be a context switch from main to first task started which is Green_Led_TaskA.

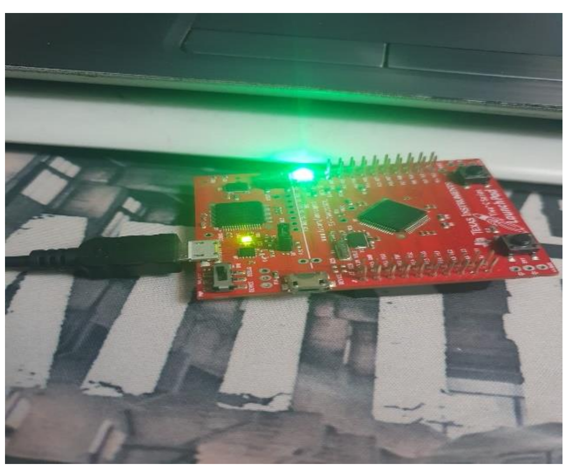
➔ This is for the first time only and after that the context switch will happen from inside the Systick or when a task calls the OS_delay(ticks) API.

```
__attribute__ ((naked))
void PendSV_Handler(void) {
  __asm volatile (
    /* __disable_irq(); */
    "  CPSID          I                  \n"

    /* if (OS_curr != (OSTask *)0) { */
    "  LDR            r1,=OS_curr        \n"
    "  LDR            r1,[r1,#0x00]      \n"
    "  CBZ            r1,PendSV_restore  \n"

    /*      push registers r4-r11 on the stack */
    "  PUSH           {r4-r11}           \n"

    /*      OS_curr->sp = sp; */
    "  LDR            r1,=OS_curr        \n"
    "  LDR            r1,[r1,#0x00]      \n"
    "  STR            sp,[r1,#0x00]      \n"
    /* } */

    "PendSV_restore:                     \n"
    /* sp = OS_next->sp; */
    "  LDR            r1,=OS_next        \n"
    "  LDR            r1,[r1,#0x00]      \n"
    "  LDR            sp,[r1,#0x00]      \n"

```

→ Green led turned on and then calls the delay API in which the kernel turns its state to be blocked and calls the scheduler.

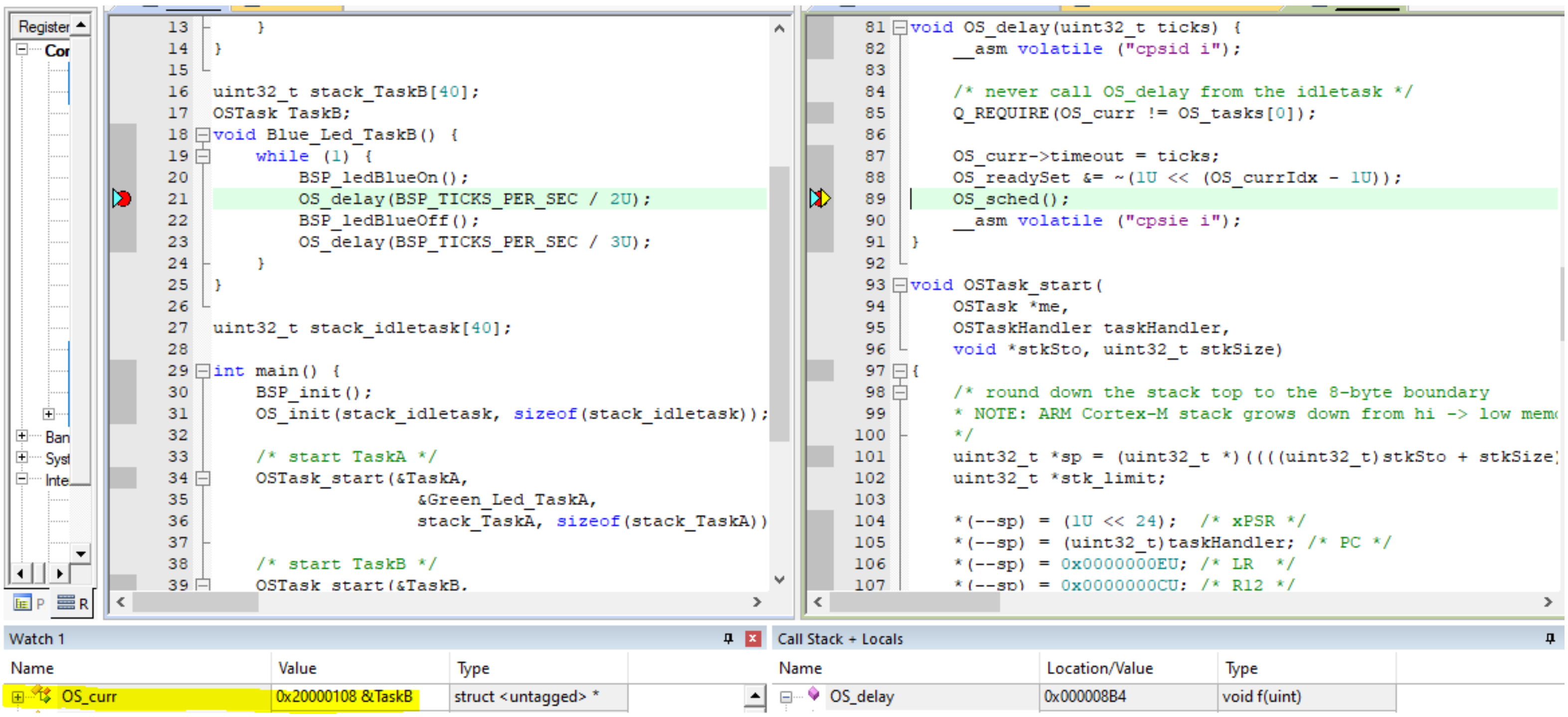
→ Note that OS_curr is coming from TaskA which turns the green led on.

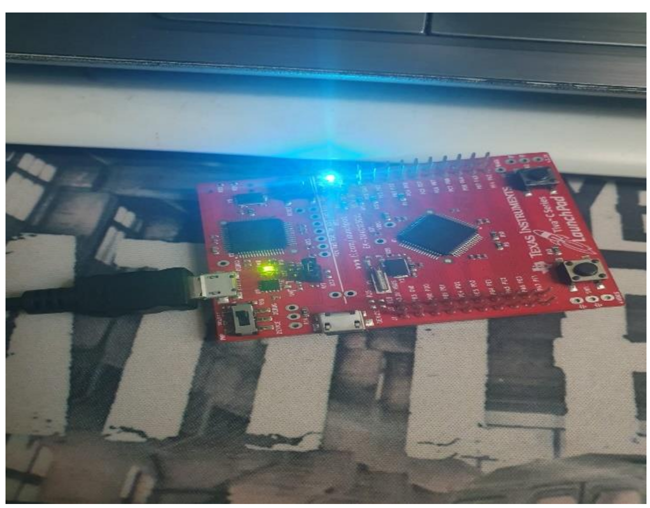




➔ Now, the scheduler turns to the next ready task which is Blue_Led_TaskB, the Blue led turned on and then calls the delay API in which the kernel turns its state to be blocked and calls the scheduler.

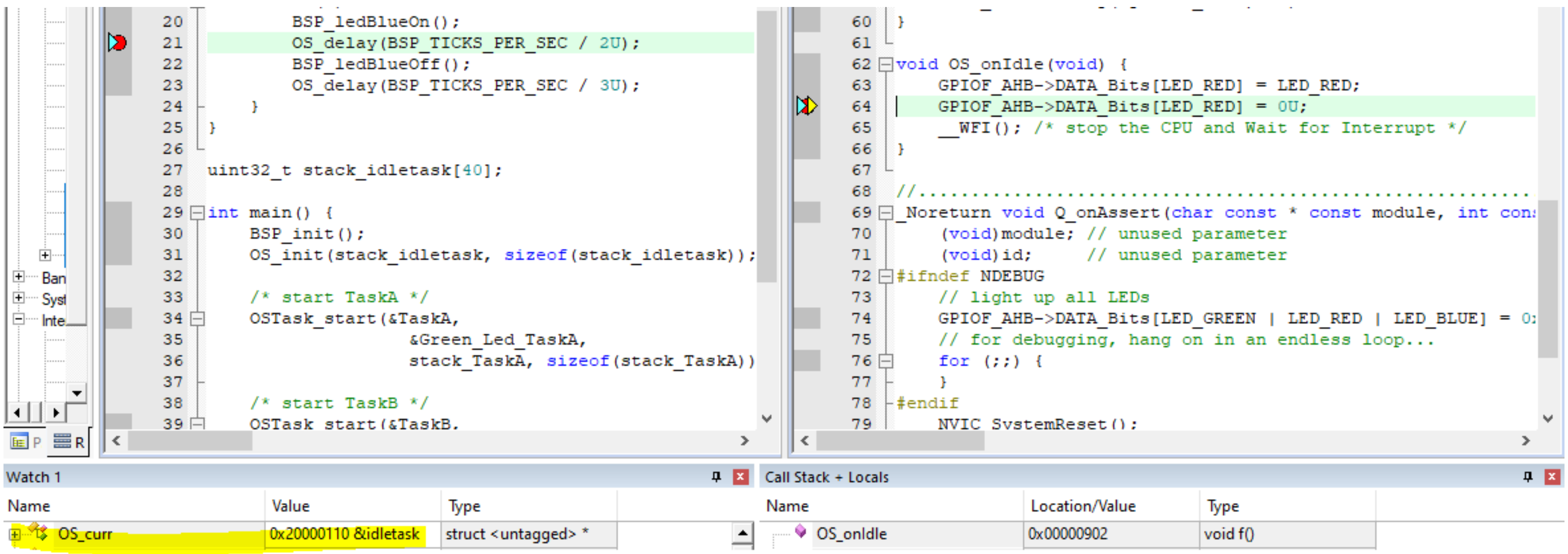
➔ Note that OS_curr is coming from TaskB which turns the blue led on.

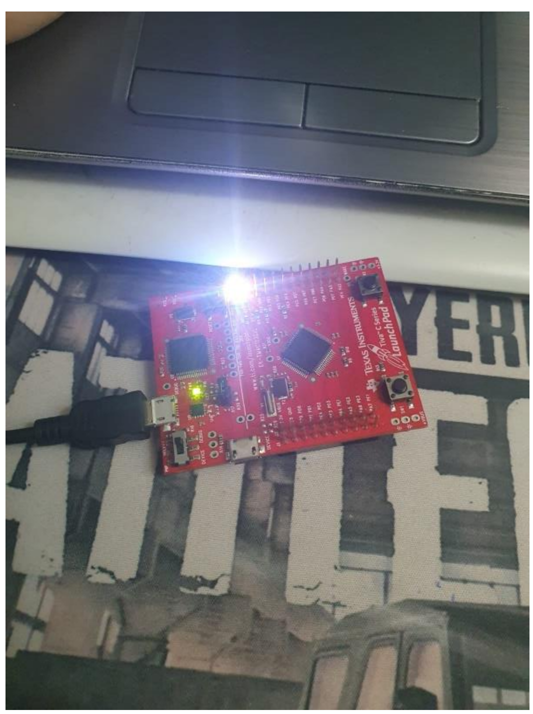




➔ Now, the scheduler doesn't find any ready task, so it schedules the idle task.

➔ Note that OS_curr is the idle task which turns the red led on before line 64 and then saves CPU cycles and waits for interrupt using __WFI().





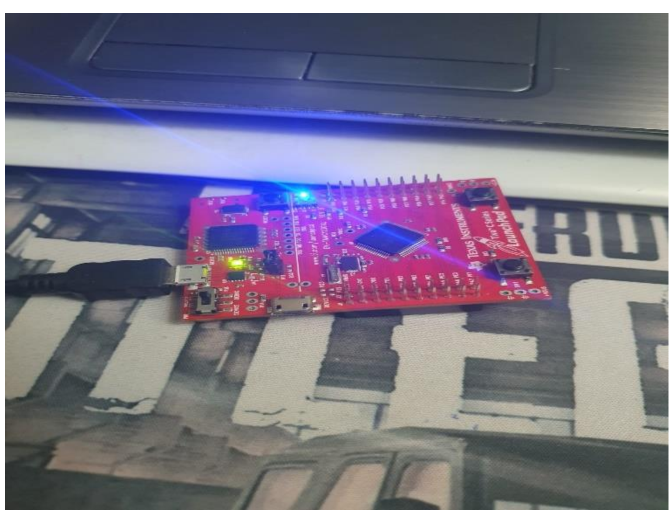
➔ After the time slice elapsed, systick interrupts and decrements the ticks for all blocked tasks and then calls the scheduler to see if any of them become ready.

➔ After several clock ticks, one of the tasks which is TaskA will be ready first and scheduled to be run.

```c
void SysTick_Handler(void) {

    OS_tick();

    __disable_irq();
    OS_sched();
    __enable_irq();
}
```

➔ TaskA returns where it was pre-empted and turns off the green led then delays again on the API so, blocked and idle task scheduled.

```c
uint32_t stack_TaskA[40];
OSTask TaskA;
void Green_Led_TaskA() {
    while (1) {
        BSP_ledGreenOn();
        OS_delay(BSP_TICKS_PER_SEC / 4U);
        BSP_ledGreenOff();
        OS_delay(BSP_TICKS_PER_SEC * 3U / 4U);
    }
}
```

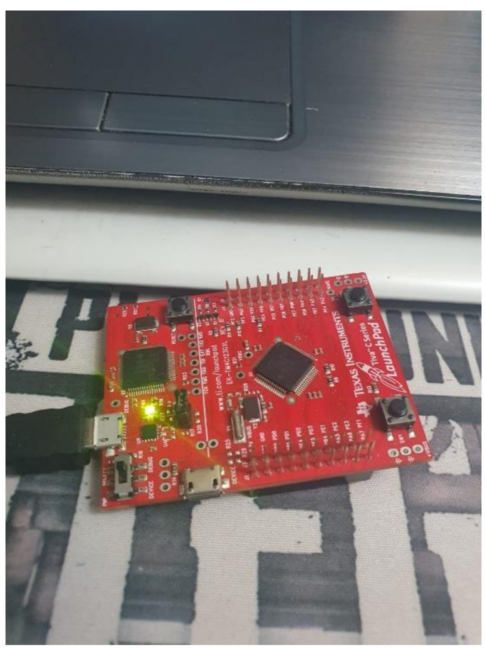


➔ After idle task re-scheduled and after some clock ticks TaskB becomes ready and scheduled from Systick to run.

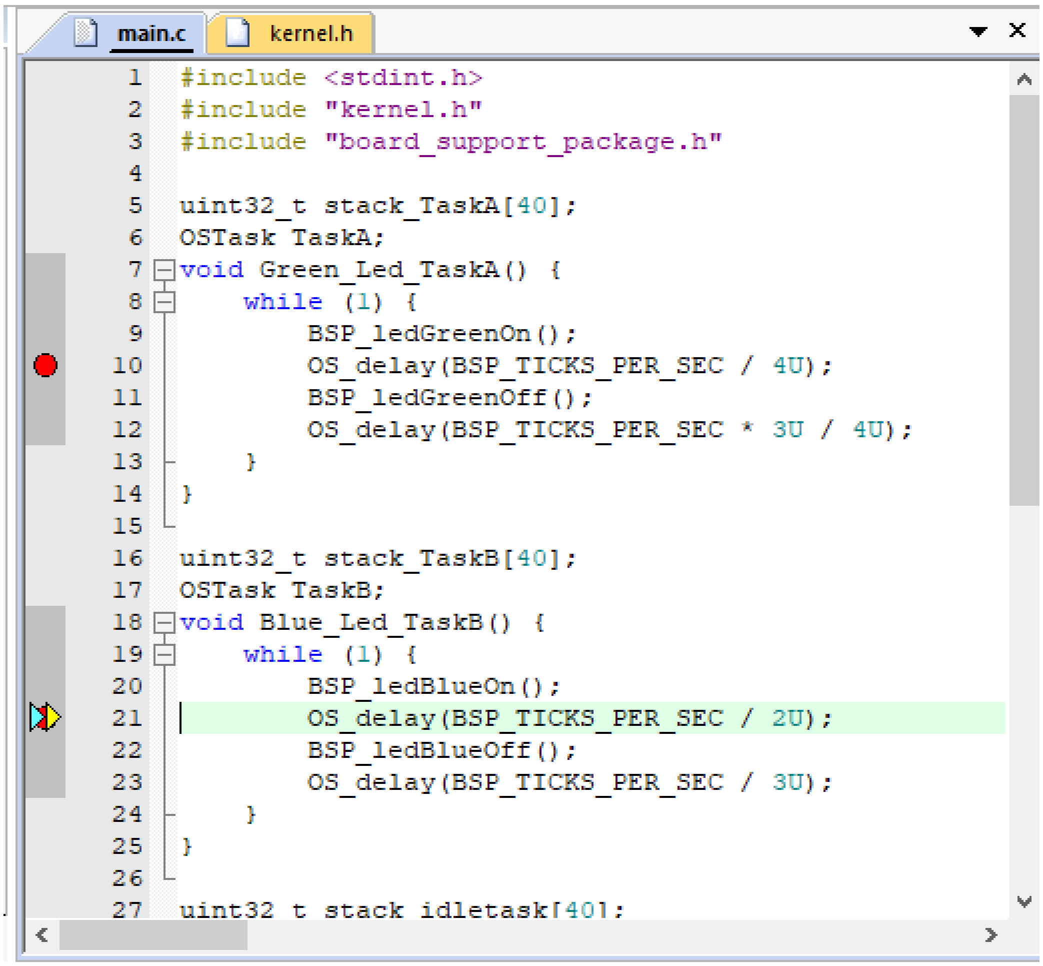
```
void OS_onIdle(void) {
    GPIOF_AHB->DATA_Bits[LED_RED] = LED_RED;
    GPIOF_AHB->DATA_Bits[LED_RED] = 0U;
    __WFI(); /* stop the CPU and Wait for Interrupt */
}
```

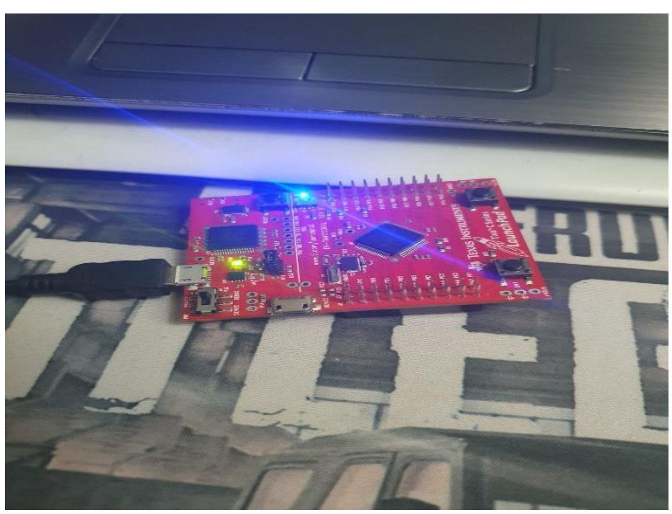
```
uint32_t stack_TaskB[40];
OSTask TaskB;
void Blue_Led_TaskB() {
    while (1) {
        BSP_ledBlueOn();
        OS_delay(BSP_TICKS_PER_SEC / 2U);
        BSP_ledBlueOff();
        OS_delay(BSP_TICKS_PER_SEC / 3U);
    }
}
```

➔ TaskB returns where it was pre-empted and turns off the blue led then delays again on the API so, blocked and idle task scheduled.



➔ After the delay elapsed, this time Task2 is ready faster than Task1 so, scheduled to be run and it returns where it was pre-empted and turns on the blue led again.





Drive link for more details about the code & a small demo (7 seconds) showing the realtime use of the application on tiva-c board:

https://drive.google.com/drive/folders/1AmCjobAvqnSEftj84_5n0aHXcbl2kiag?usp=sharing

End