University of Victoria

ECE 455 B02

Some Hints to Complete Project 2

Amir Sepahi

Khalid Al-Hammuri

March 2022

```
 */Standard includes/* .
#include <stdint.h<
#include <stdio.h<
#include <string.h<
#include "stm32f4_discovery.h"
 */Kernel includes/* .
#include "stm32f4xx.h"
#include "../FreeRTOS_Source/include/FreeRTOS.h"
#include "../FreeRTOS_Source/include/queue.h"
#include "../FreeRTOS_Source/include/semphr.h"
#include "../FreeRTOS_Source/include/task.h"
#include "../FreeRTOS_Source/include/timers.h"



/*-----------------------------------------------------------*/
#define DD_SCHEDULER_PRIO  configMAX_PRIORITIES-1
#define GENERATOR_PRIO           configMAX_PRIORITIES-2
#define HIGH_PRIO          configMAX_PRIORITIES-3
#define LOW_PRIO           2
#define Aux_STACK_SIZE     ((unsigned short ) 30)
#define Mul_Divider        11000
#define schQUEUE_LENGTH          3
#define genQUEUE_LENGTH          3
#define tcQUEUE_LENGTH           3
```

Then you need to define structures here:

```
enum task_Types{
        //You need to complete this part
}task_Types;

enum Request_Types{
        //You need to complete this part
}Request_Types;

typedef struct node{
        //You need to complete this part
};

typedef struct node{
        //You need to complete this part
}node;

struct overdue_tasks {
        //You need to complete this part
};

typedef struct {
        //You need to complete this part
}tcMSG, * tcMSG_ptr;

typedef struct {
        //You need to complete this part
} auxTPARAM;
```

```
 ********/Tasks
/***************************************************************

***************************************************************************/
/***

static void DD_Scheduler( void *pvParameters );

static void Task_Generator( void *pvParameters );

static void Auxiliary_Task( void *pvParameters );

static void Task_Monitor (void *pvParameters );


 ********/local Functions/*****************************************************

***************************************************************************/
/***

static TaskHandle_t dd_tcreate(auxTPARAM* auxtParameter, const char * const
task_name);

static BaseType_t dd_delete(TaskHandle_t TaskToDelet );

static BaseType_t dd_return_active_list(void );

static BaseType_t dd_return_overdue_list(void );

static BaseType_t insert_node(node** head, node* new_node );

static void adjust_prios(node* head );

static node* remove_node(node** head, TaskHandle_t target );

static void print_list(node *head );

static void Delay_Init(void );

static void prvSetupHardware( void );


 ********/local Functions/*****************************************************

***************************************************************************/
/***

node *                          head = NULL;

node *                          overdue_head  = NULL;

uint32_t                        EXECUTION = 0;

uint32_t                        multiplier =0;

volatile uint32_t               utilization=0;
```

```
TickType_t          START=0;             //The start time of the scheduler.
TickType_t          CURRENT_SLEEPک0=

xQueueHandle        xDDSQueue_handle   =0;
xQueueHandle        xDDSG_Queue_handle=0;
xQueueHandle        deleteQueue_handle = 0;
xQueueHandle        activeQueue_handle = 0;
xQueueHandle        overdueQueue_handle = 0;
xQueueHandle        creatQueue_handle = 0;
/*-----------------------------------------------------------*/
```

You need to complete main part:

```
int main(void)
{
        /*Configure the system ready to run the demo.  The clock configuration
        can be done here if it was not done before main() was called*/
                prvSetupHardware;()
        /*initialize the multiplier that is used to convert delay into cycle.*/
                Delay_Init();

        /* Create the queue used by the queue send and queue receive tasks. */
                Complete this part
        /* Add to the registry, for the benefit of kernel aware debugging. */
                Complete this part
        /* Start the tasks and timer running. */
                vTaskStartScheduler();
        for( ;; );          // we should never get here!
        return 0;}
```

DD_Scheduler:

```
static void DD_Scheduler( void *pvParameters )
{
    tcMSG        sch_msg;
    BaseType_t   response = pdFAIL;
    CURRENT_SLEEP = 100; // This is for initialization and to give the generator task to have a
                                        chance to run at the start of the program.

    node* deleted = NULL;
    node* head_add = head;

    tcMSG gen_msg;
    node gen_data;
    gen_msg.DATA = &gen_data;

     START = xTaskGetTickCount();

    while(1)
    {

    /* waits to receive a scheduling request. If there is a task running, the CURRENT_SLEEP
    time is the deadline of the running task. If the scheduler does not receive anything and times
    out, it means that the task has missed the deadline. Because tasks send a delete request if they
    meet their deadline. */
        if(xQueueReceive(xDDSQueue_handle, &sch_msg, CURRENT_SLEEP))
        {
            switch (sch_msg.req)
            {
            case create:
                response = insert_node(&head, sch_msg.DATA);
                Complete Here
            case delete:
                deleted = remove_node(&head, sch_msg.DATA->tid);
                Complete Here

                if (deleted != NULL)
                {
                    Complete Here
                }



            case return_active_list:

                Complete Here



            case return_overdue_list:
                Complete Here

            default:
                printf("DDScheduler received an invalid msg!\n");
            }
        }
        else
        {
        /* Check the task_list. If it is not empty, being here means a deadline is missed!
         * the scheduler should do these:
        * lower the priority of the overdue task
        * move it to the overdue list
        * raise the priority of the next task in the ready queue
        * notifies the task generator to create the next instance of the (now overdue) task. */
            if (head != NULL)
            {
                /* deadline is reached */
```

```
                    // set task's priority to lowest
                        Complete Here

                    // place task in overdue list; remove from active list
                        Complete Here
                    // send message to generator to create periodic task again
                        Complete Here

                else
                {
                    CURRENT_SLEEP = 100;
                    printf("Noting to do yet!\n");
                }
            }
        }
    }
```

Task Generator:

```
static void Task_Generator( void *pvParameters )
{
    auxTPARAM       pTaskParameters;
    tcMSG           regen_msg;
    node            regen_DATA;
    regen_msg.DATA = &regen_DATA;

    pTaskParameters.type = periodic;
    pTaskParameters.exetime = 1000;
    pTaskParameters.deadline = 2000;
    pTaskParameters.rel_deadline = 2000;
    pTaskParameters.execution_cycle = Complete Here;
    if(dd_tcreate(&pTaskParameters, "TASK1") == NULL)
        printf("dd_tcreate Failed!\n");

    while(1)
    {
        if(xQueueReceive(xDDSG_Queue_handle, &regen_msg, portMAX_DELAY) == pdPASS)
        {
            // we only re-create periodic tasks
            if (regen_msg.DATA->type == periodic)
            {
                // calculate deadline of next periodic task
                // = previous deadline + relative deadline
                Complete Here

                // calculate time until task should be created (its period)
                Complete Here
        // ensure sleep_time is not negative

        // this happens often when a task is overdue, and its next start time is the current time

        // this sometimes results in a small negative value for sleep_time

        Complete Here


        // creates the next task (new absolute deadline, same execution and relative deadline)

        Complete Here


        }

    }
```

```
static void Task_Monitor (void *pvParameters)
{
    while(1)
    {
        printf("System idle time is %lu\n", utilization);
        printf("ACTIVE TASKS: \n");
        dd_return_active_list();
        printf("\nOVERDUE TASKS: \n");
        dd_return_overdue_list();
        vTaskDelay(10000);
    }
}


static void Auxiliary_Task (void *pvParameters)
{
    auxTPARAM *AuxTaskParam = (auxTPARAM *) pvParameters;
    uint32_t *cycles = &(AuxTaskParam->execution_cycle);
    //printf("AST %d\n",  xTaskGetTickCount());
    while (1)
    {
        while ((*cycles)--);
        // delete the task!
        //printf("AET %d\n",  xTaskGetTickCount());
        dd_delete(xTaskGetCurrentTaskHandle());
    }
}
```

```
static TaskHandle_t dd_tcreate(auxTPARAM * auxtParameter,const char * const
task_name)
{
    BaseType_t                  response = pdFAIL;
    tcMSG                       Aux_tcmsg;
    TaskHandle_t                Aux_thandle = NULL;
    node                        Aux_msg_DATA;

    Aux_tcmsg.DATA = &Aux_msg_DATA;

    creatQueue_handle = xQueueCreate( tcQUEUE_LENGTH, sizeof(response));
    vQueueAddToRegistry(creatQueue_handle, "TCreatorQ");

    //vQueueAddToRegistry( tcQueue_handle, auxTName);
    Complete Here

    if(xTaskCreate( Auxiliary_Task, task_name, Aux_STACK_SIZE, auxtParameter, 1,
    &Aux_thandle) == pdPASS)
    {
        Complete Here

        if(xQueueSend(xDDSQueue_handle, &Aux_tcmsg, 100))
        {
            if(xQueueReceive(creatQueue_handle, &response, 100))
            {
                if (response == pdPASS)
                {
                    Complete Here
                }
                else
                    return NULL;
            }
        }
        else
            return NULL;
    }
    else
    {
        printf ("Cannot Create Auxiliary task at the moment!\n");
        return NULL;
    }
```

```
static BaseType_t dd_delete(TaskHandle_t TaskToDelet)
{
    BaseType_t      response = pdFAIL;
    tcMSG               Aux_tcmsg;
    node            deleteTask;
    Aux_tcmsg.DATA = &deleteTask;

    Complete Here

    //vQueueAddToRegistry( tcQueue_handle, auxTName);

    Complete Here

    if(xQueueSend(xDDSQueue_handle, &Aux_tcmsg, 100))
    {
        if(xQueueReceive(deleteQueue_handle, &response, 100))
        {
            if (response == pdPASS)
            {
                Complete Here
                return pdPASS;
            }
            else
                return pdFAIL;
        }
    }
    else
        return pdFAIL;
}
```

```
static BaseType_t dd_return_active_list(void)
{
    node*               response = NULL;
    tcMSG                   Aux_tcmsg;

    Complete Here

    Aux_tcmsg.req = return_active_list;
    Aux_tcmsg.tcQueue = activeQueue_handle;

    if(xQueueSend(xDDSQueue_handle, &Aux_tcmsg, 100))
    {
        if(xQueueReceive(activeQueue_handle, &response, 100))
        {
            if (response != NULL)
            {
                Complete Here
            }
            else
                return pdFAIL;
        }
    }
    else
    {
        printf ("Cannot send msg to the scheduler!\n");
        return pdFAIL;
    }}
```

```
static BaseType_t dd_return_overdue_list(void)
{
    node*           response = NULL;
    tcMSG               Aux_tcmsg;

    Complete Here

    Aux_tcmsg.req = return_overdue_list;
    Aux_tcmsg.tcQueue = overdueQueue_handle;

    if(xQueueSend(xDDSQueue_handle, &Aux_tcmsg, 100))
    {
        if(xQueueReceive(overdueQueue_handle, &response, 100))
        {
            if (response != NULL)
            {
                Complete Here
            }
            else
                return pdFAIL;
        }
    }
    else
    {
        printf ("Cannot send msg to the scheduler!\n");
        return pdFAIL;
    }
}
```

```
/* Insert a task_list (in sorted order) into a (sorted) task list */
static BaseType_t insert_node(node** head, node* new_node)
{
        new_node->next = NULL;
        node* current = *head;
        node* old_node = (node*) current->next;
        // list is empty or new_task_list is new head
        Complete Here

        // new_task_list is not new head
        Complete Here
}
```

```
/* Function to assign high priority to head of active list,
 * and low priority to all other tasks in list.
 * Additionally, this function modifies the 'CURRENT_SLEEP' value, which is
 * the time until the next deadline.
 */
void adjust_prios(node* head)
{
    UBaseType_t prio;

    if(head == NULL){
        // no task. sleep so task generator can create some tasks ...
        Complete Here
        return;
    }
    // NOTE: at any given time, only one task (the head) should have 'high priority'

    // check if current head of list is 'high priority'
    Complete Here
    if(prio != (UBaseType_t) HIGH_PRIO)
    {
        // set head to highest priority
        Complete Here
        // find task in rest of list with 'high' and set to 'low'
        Complete Here
        while(temp != NULL)
        {
            prio = uxTaskPriorityGet(temp->tid);
            if(prio == HIGH_PRIO)
            {
                Complete Here
```

```
                    break;
                    //break because there should only be one high prio task at once
                }
                temp = (node*) temp->next;
            }
        }

        // recalculate scheduler sleep time as deadline of head task
        Complete Here
    };
```

```
/* Remove a specified task_list from a task list */
static node* remove_node(node** head, TaskHandle_t target)
{
    node*   deleted_node = NULL;
    node*   current = *head;
    // target is head of list
    if ((*head)->tid == target)
    {
        Complete Here
    }
    // target is in middle of list
    else
    {
        Complete Here
        // traverse list, looking for target
        while (current->next != NULL)
        {
            Complete Here
        }
        temp_node = (node*) current->next;
        // target found in list
        if (temp_node->tid == target)
        {
            Complete Here
        }
    }
    return deleted_node;
}
```

```
/* Outputs the task list */
void print_list(node *head)
{
        node* current = head;

        // traverse the list, printing each task_list's id, execution time, and deadline
        while (current != NULL) {
                printf("task Aux, exec_time = %ld, deadline = %ld\n",
                current>execution_time, current->deadline);
                current = (node*) current->next;
        }
}

/*--------------------------------------------------------*/
```

```
static void Delay_Init(void)
{
    RCC_ClocksTypeDef RCC_Clocks;

    /* Get system clocks */
    RCC_GetClocksFreq(&RCC_Clocks);

    /* While loop takes 4 cycles */
    /* For 1 ms delay, we need to divide with 4K */
    printf("Freq: %d \n", RCC_Clocks.HCLK_Frequency);
    multiplier = RCC_Clocks.HCLK_Frequency / Mul_Divider; // to calculate 1 msec

}

/*----------------------------------------------------------*/
```

```
void vApplicationMallocFailedHook( void )
{
	/* The malloc failed hook is enabled by setting
	configUSE_MALLOC_FAILED_HOOK to 1 in FreeRTOSConfig.h.

	Called if a call to pvPortMalloc() fails because there is insufficient
	free memory available in the FreeRTOS heap.  pvPortMalloc() is called
	internally by FreeRTOS API functions that create tasks, queues, software
	timers, and semaphores.  The size of the FreeRTOS heap is set by the
	configTOTAL_HEAP_SIZE configuration constant in FreeRTOSConfig.h. */
	for( ;; );
}
/*-----------------------------------------------------------*/
```

```
void vApplicationStackOverflowHook( xTaskHandle pxTask, signed char
*pcTaskName )
{
        ( void ) pcTaskName;
        ( void ) pxTask;

        /* Run time stack overflow checking is performed if
        configconfigCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2.  This hook
        function is called if a stack overflow is detected.  pxCurrentTCB can be
        inspected in the debugger if the task name passed into this function is
        corrupt. */
        for( ;; );
}
/*--------------------------------------------------------*/
```

```
void vApplicationIdleHook( void )
{
	utilization++;
}
/*-----------------------------------------------------------*/
```

```
static void prvSetupHardware( void )
{
        /* Ensure all priority bits are assigned as preemption priority bits.
        http://www.freertos.org/RTOS-Cortex-M3-M4.html */
        NVIC_SetPriorityGrouping( 0 );

        utilization++;

        /* TODO: Setup the clocks, etc. here, if they were not configured before
        main() was called. */
}
```