University of Victoria

Department of Electrical and Computer Engineering

ECE 455 - Real Time Computer Systems

Project 2 – Deadline-Driven Scheduler

Brett Dionello	V01026046
Mudit Jaswal	V00982906

TABLE OF CONTENTS

INTRODUCTION

This project implements a Deadline-Driven Scheduler (DDS) based on the Earlies First (EDF) algorithm, built on top of FreeRTOS. The DDS dynamically manages task priorities to ensure tasks with the closest deadlines are scheduled to run first. By integrating FreeRTOS task priorities at runtime, the scheduler maximizes processor utilization while ensuring hard real time deadlines are met.

FUNCTIONAL REQUIREMENTS

- The Deadline-Driven Scheduler must correctly execute all three Test Benches using the EDF scheduling.
- Auxiliary tasks should only interact with the DDS through its four primary functions.
- The Monitor Task should continuously track and report the number of active, completed, and overdue DD-tasks.
- Queues will be utilized for communication between tasks, with global variables strictly prohibited for this purpose.
- Software Timers will be employed to handle time-dependent tasks.
- The system must support both periodic and aperiodic DD-Tasks.
- Tasks must be used to control code execution.
- Inter-task communication must use FreeRTOS queues to manage and transfer DD-Task information between system components.

DESIGN SOLUTION

System Design

The Initial design process begins with a high-level layout of the code components, function parameters and return types that are required to fulfill the functional requirements. In Figure 1: Design table the system design is laid out before any code is written.

Tasks	Type	Function args	Function Return	Function Calls	Queue	Queue Action	Timers	Interrupts
DD_Scheduler	F	void	void		message_q	RECEIVE		
					active_list_q	SEND		
					complete_list_q	SEND		
					overdue_list_q	SEND		
create_dd_task	DD	task_handle, task_type, task_id, Abs_deadline	void		message_q	SEND		
complete_dd_task		task_id	void			SEND		
get_active_dd_task_list		void			message_q/active_list_q	SEND/RECEIVE		
get_complete_dd_task_li		void	*dd_task_list		message_q/complete_list_q	SEND/RECEIVE		
get_overdue_dd_task_list		void			message_q/overdue_list_q	SEND/RECEIVE		
Generator	F	void	void	create_dd_task(task_info)			task1_timer, task2,_timer, task3_timer	task1_generator, task2_generator, task3, generator
Monitor	F	void	void	get_active_dd_task_list(), get_complete_dd_task_list(), get_overdue_dd_task_list()				
user_task1	F	execute_time	task_handle	complete_dd_task(task_handle)				
user_task2			task_handle					
user_task3			task_handle					

Figure 1: Design table

Timers

Three timers are to be configured to generate user tasks at set intervals for each test bench case.

Queues

Three monitor interface queues and one message queue are to be created.

Interface queues:

- sent to by the scheduler and receive by the monitor
- holds a pointer to the head of a linked list

Message queue:

- sent to by create, delete and monitor interface functions
- received by the scheduler
- holds a message and task information required by the scheduler to perform the action requested by the message type

Scheduling

The DD scheduler performs all actions required to implement the functional requirements of the DDS.

Messages:

- must check the queue and handle the message in a timely manner

Lists:

- allocate memory for new task nodes
- sort the active list for priority based on absolute deadlines
- manage task lists such as moving from active to overdue, and from active or overdue to complete.

Tasks:

- run the highest priority task in the active list
- pre-empt running task when a higher priority task is created
- check for overdue tasks and update lists
- allow overdue tasks to continue running
- handle aperiodic tasks by running in slack time between periodic tasks

Inter-task Communication

Sequence Diagram

In Figure 2: DDS Sequence Diagram, the system interactions can be seen, and the components are briefly described here.

Timers:

- Three timers that run for their respective user task periods and call their own generator callback function.

Generators:

- The callback function will create the user tasks as a F_task and pass the task info to the Create_task() function.

DD_Create_Delete:

- The create task and delete task functions are combined into a single entity in this diagram
- Create_task()
    - Receives task info from generator call backs as function parameters
    - Sends the information to the message queue with a CREATE message.
- Complete_task()
    - Receives task handle from user task as function parameter
    - Sends task handle and completion time to message queue with COMPLETE message

User_TASKn:

- Does user task work
- LED on, wait, LED off
- Created by the generator, run by the scheduler
- Calls complete_task() when execution time expires

MSG_Queue

- Sent to by Monitor Task, and Create and Delete functions
- Received by DD_Scheduler
- `Message types:`
    - `CREATE_TASK, COMPLETE_TASK, GET_ACTIVE, GET_COMPLETE, GET_OVERDUE`

DD_Scheduler:

- Periodically checks for over-due tasks and messages from the message queue
    - Acts based on message in message queue

DD_Interface:

- Combines all “get_list” functions into one enitity
    - Get_list functions:
        - Called by monitor
        - Send message to message queue
        - Receives pointer to head of list from list queue

List_Queues:

- Combines all list queues into one entity
- DD_Scheduler sends pointer to head of respective list
- Monitor receives pointer.

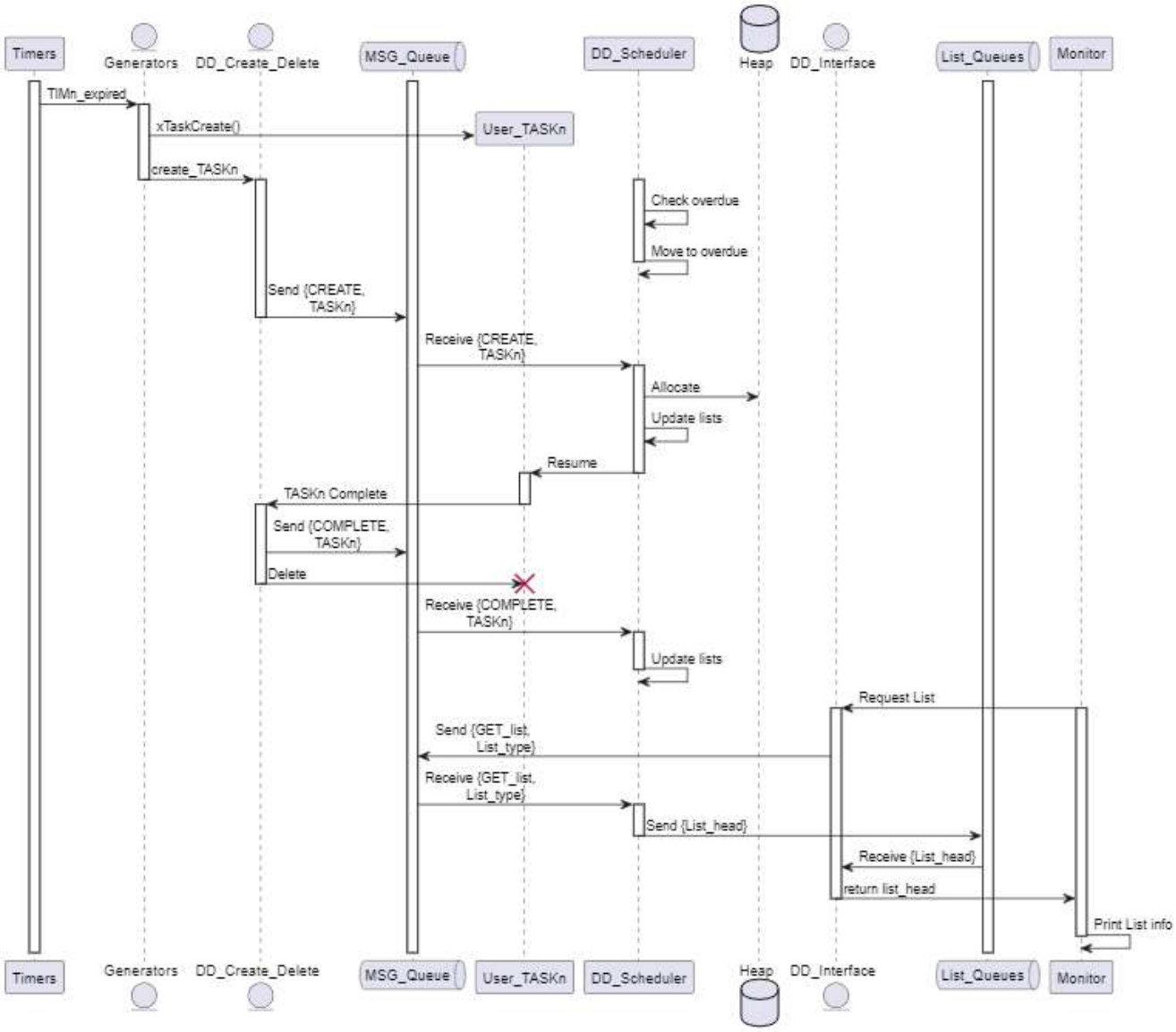

Figure 2: DDS Sequence Diagram

TASK DESCRIPTIONS

DD Scheduler Task

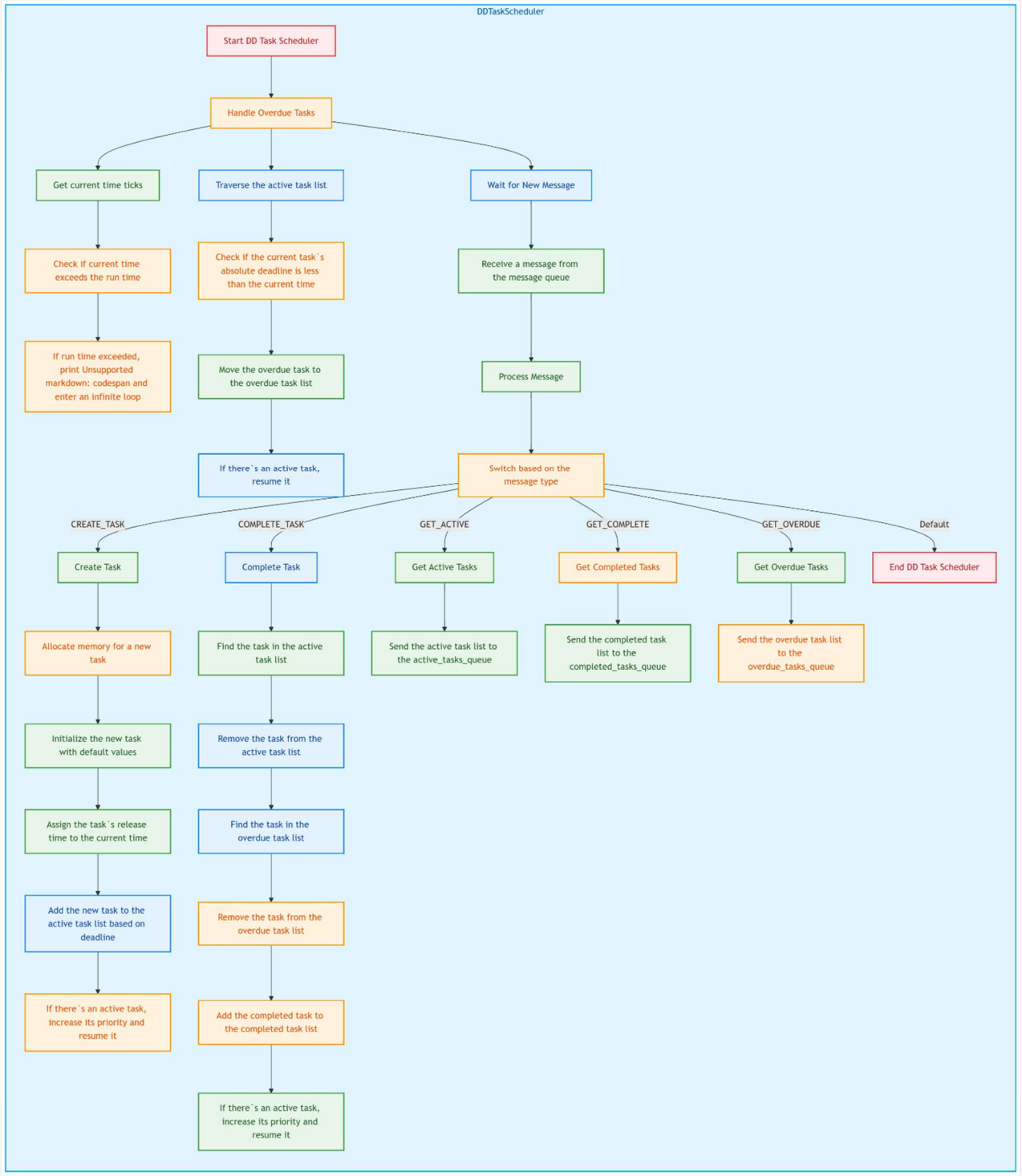

Figure 3: DDS Scheduler flow chart

The DD_Task_Scheduler function (priority 4) is designed to manage tasks dynamically, based on their deadlines and priority. It uses Active task list, completed task list and overdue task list to organize tasks for tasks that are currently active, tasks that have been processed successfully, and tasks that have missed their deadlines and need to be managed accordingly.

- Initialization: The function starts by initializing three pointers to the head of the list named as active_head, completed_head, and overdue_head.
- Overdue Task Handling: The scheduler continuously checks for overdue tasks. Each iteration of the loop checks the current system time to determine if any tasks have missed their deadlines. If the current time exceeds a task's absolute deadline, that task is moved from the active_head list to the overdue_head list. The task is removed from the active list, and if there are any tasks left in the active_head list, the next task is resumed.

- Message Handling: The scheduler listens for messages in the message queue (`message_queue`). When a new message is received, the scheduler processes it based on the message type.

- CREATE_TASK: When the message type is CREATE_TASK, periodic tasks are inserted in a sorted order based on the deadlines, while aperiodic tasks are added at the end. If there are any active tasks, the task with the earliest deadline is resumed, and its priority is raised to ensure its executed.

- COMPLETE_TASK: When the message type is COMPLETE_TASK, the scheduler searches for it in the active list. Once found, the task is removed from the active list and added to the completed_head list. If the completed task is overdue, it is also removed from the overdue_head list. Once a task is completed, the next active task is resumed and given higher priority.

- The messages GET_ACTIVE, GET_COMPLETE, and GET_OVERDUE retrieve the current lists of active, completed, and overdue tasks, respectively, and send them to their respective queues (active_tasks_queue, completed_tasks_queue, and overdue_tasks_queue) for external use or reporting.

DD Monitor Task

The DD_Task_Monitor function (priority 3) is responsible for continuously monitoring and reporting the status of tasks in three categories: active, completed, and overdue. It retrieves the current lists of tasks for each category, processes them, and prints the relevant details such as task ID, release time, deadline, and completion time. The function runs in an infinite loop, updating task counts for each category and providing periodic updates with a short delay between iterations.

- **Monitors Active Tasks**: Retrieves and prints details of active tasks, then increments the active task count.
- **Monitors Completed Tasks**: Retrieves and prints details of completed tasks, then increments the completed task count.

- **Monitors Overdue Tasks**: Retrieves and prints details of overdue tasks, then increments the overdue task count.

DD Task Generator

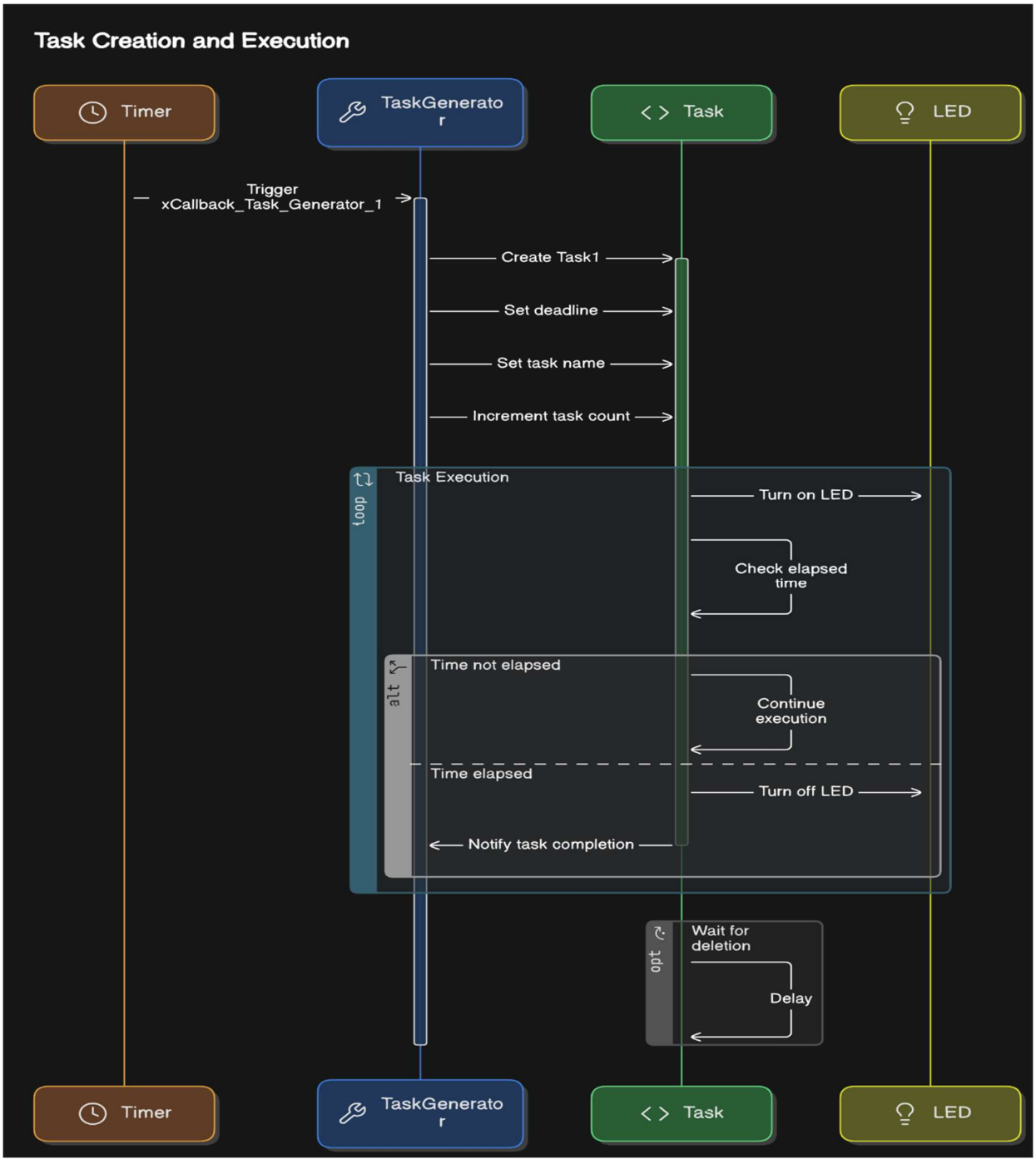

Figure 4: DD Task Generator flow diagram

The code defines three separate tasks, each controlling an LED (Red, Green, and Blue) on a system for a specific period. Each task turns on the corresponding LED, then waits for the specified execution time before turning off the LED. After the LED is turned off, the task calls complete_dd_task to indicate that it has finished its execution and should be marked as complete. The task then enters an idle state, waiting to be deleted.

- **Task1 (Red LED)**: Turns on the red LED, waits for the specified execution time, then turns it off and marks the task as complete.
- **Task2 (Green LED)**: Turns on the green LED, waits for the specified execution time, then turns it off and marks the task as complete.
- **Task3 (Blue LED)**: Turns on the blue LED, waits for the specified execution time, then turns it off and marks the task as complete.

The three functions named xCallback_Task_Generator_1, xCallback_Test_Generator_2, and xCallback_Task_Generator_3 are responsible for generating periodic tasks at specified intervals. Each function calculates the next absolute deadline based on the current tick count and the predefined period for the respective task (Task 1, Task 2, or Task 3). Once the deadline is calculated, a new task is created using xTaskCreate, where each task is associated with a unique name. The task is then registered with the task management system by calling create_dd_task. Afterward, a global counter is incremented to ensure each task has a unique identifier.

- **xCallback_Task_Generator_1**: Calculates the next deadline for Task 1, creates Task 1, registers it with the task management system, and increments the global counter for Task 1.
- **xCallback_Task_Generator_2**: Calculates the next deadline for Task 2, creates Task 2, registers it with the task management system, and increments the global counter for Task 2.
- **xCallback_Task_Generator_3**: Calculates the next deadline for Task 3, creates Task 3, registers it with the task management system, and increments the global counter for Task 3.

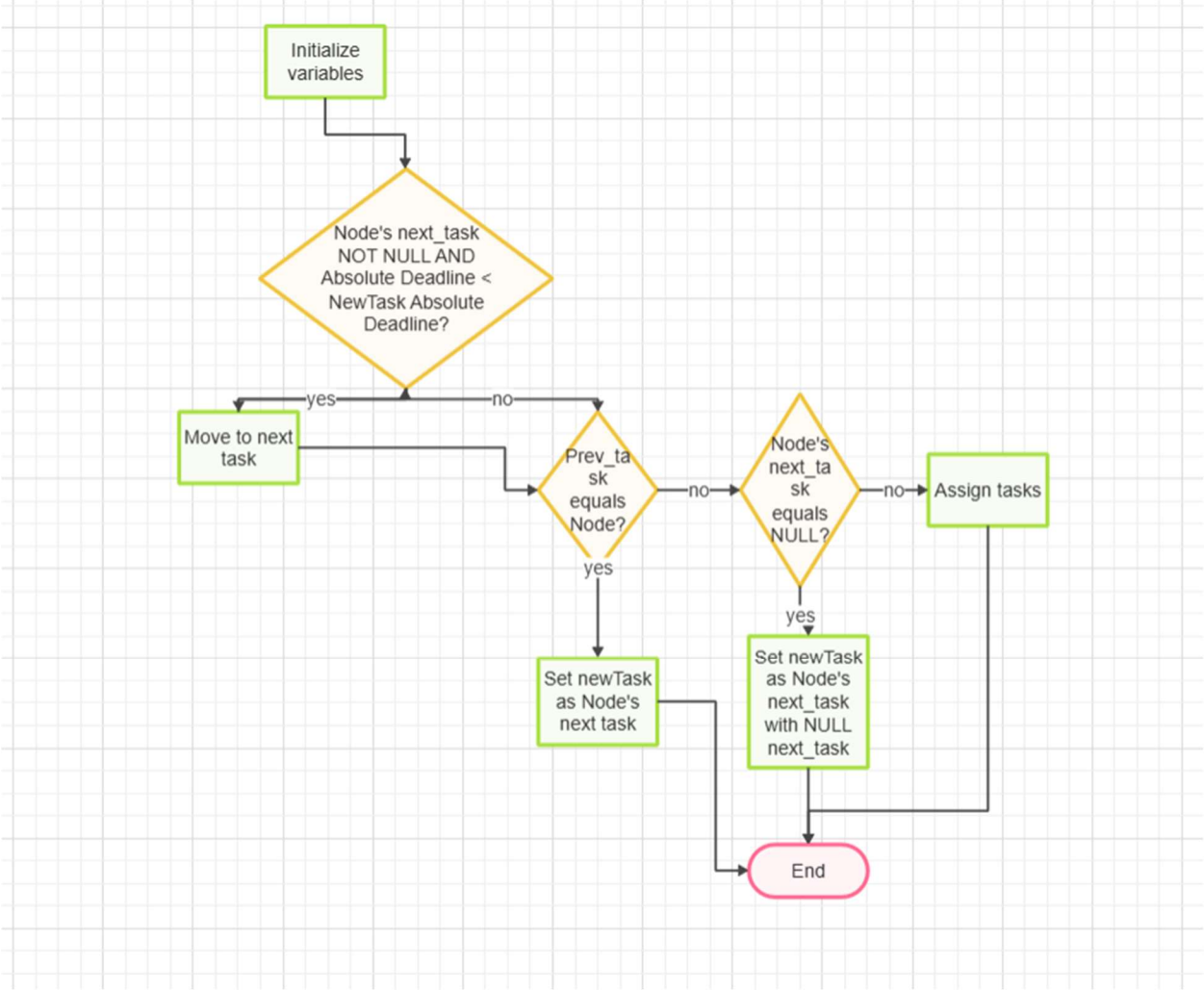

Figure 5: Sorting algorithm flow diagram

HARDWARE IMPLEMENTATION

Define macros are used for the hardware and freeRTOS configuration, and test bench macros are used to configure the test bench case.

```
/*-------------  Defines  --------------*/
#define DEFAULT_PRIORITY 1
#define amber_led   LED3
#define green_led   LED4
#define red_led     LED5
#define blue_led    LED6
#define QUEUE_LENGTH 10
```

Figure 6: Config macros

```
/*-------------  Test 1  --------------*/
#define TASK_1_PERIOD_MS 500
#define TASK_2_PERIOD_MS 500
#define TASK_3_PERIOD_MS 750
#define TASK_1_EXECUTE_TIME_MS 95
#define TASK_2_EXECUTE_TIME_MS 150
#define TASK_3_EXECUTE_TIME_MS 250
#define RUN_TIME 1500
```

Figure 7: Task test bench macros

GPIO Initialization

The built-in STM evaluation HAL is used to initialize the LEDs for use in the user tasks.

```
/* Initialize LEDs */
STM_EVAL_LEDInit(amber_led);
STM_EVAL_LEDInit(green_led);
STM_EVAL_LEDInit(red_led);
STM_EVAL_LEDInit(blue_led);
```

Figure 8: LED initialization

FreeRTOS Configuration

The FreeRTOSConfig.h file is used to set the FreeRTOS system parameters, the heap size is modified to handle more tasks. Note that the FreeRTOS system tick is 1000Hz as seen under “ConfigTICK_RATE_HZ”, meaning that the FreeRTOS scheduler has a time slice of 1ms which is the period of between each context switch in round robin scheduling of equal priority tasks.

```
#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             1
#define configUSE_TICK_HOOK             0
#define configCPU_CLOCK_HZ              ( SystemCoreClock )
#define configTICK_RATE_HZ              ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES            ( 5 )
#define configMINIMAL_STACK_SIZE        ( ( unsigned short ) 130 )
#define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 20 * 1024 ) )
#define configMAX_TASK_NAME_LEN         ( 10 )
#define configUSE_TRACE_FACILITY        0
#define configUSE_16_BIT_TICKS          0
#define configIDLE_SHOULD_YIELD         1
#define configUSE_MUTEXES               1
#define configQUEUE_REGISTRY_SIZE       8
#define configCHECK_FOR_STACK_OVERFLOW  2
#define configUSE_RECURSIVE_MUTEXES     1
#define configUSE_MALLOC_FAILED_HOOK    1
#define configUSE_APPLICATION_TASK_TAG  0
#define configUSE_COUNTING_SEMAPHORES   1
#define configGENERATE_RUN_TIME_STATS   0
```

Figure 9: FreeRTOSConfig.h

Task Initialization

The only F-tasks are the Scheduler, monitor, and user tasks. The scheduler and the monitor are created in main, while the user tasks are created in the timer callback function.

```
// user tasks
static void Task1(void *pvParameters);
static void Task2(void *pvParameters);
static void Task3(void *pvParameters);
// DDR system
static void DD_Task_Scheduler(void *pvParameters);
static void DD_Task_Monitor(void *pvParameters);
```

Figure 10: F-Task prototypes

```
xTaskCreate(DD_Task_Scheduler, "Scheduler", configMINIMAL_STACK_SIZE, NULL, 4, NULL);
xTaskCreate(DD_Task_Monitor, "Monitor", configMINIMAL_STACK_SIZE, NULL, 3, NULL);
```

Figure 11: F-task creation in main

```
xTaskCreate(Task2, task_str, configMINIMAL_STACK_SIZE, NULL, DEFAULT_PRIORITY, &t_handle); // create task
```

Figure 12: F-task creation in timer callback

Queue Initialization

```
xQueueHandle message_queue = 0;
xQueueHandle active_tasks_queue = 0;
xQueueHandle completed_tasks_queue = 0;
xQueueHandle overdue_tasks_queue = 0;
xQueueHandle get_completed_list_queue = 0;
```

Figure 13: Queue handle declaration

```
message_queue = xQueueCreate(QUEUE_LENGTH, sizeof(queue_message));
active_tasks_queue = xQueueCreate(QUEUE_LENGTH, sizeof(dd_task_list *));
completed_tasks_queue = xQueueCreate(QUEUE_LENGTH, sizeof(dd_task_list *));
overdue_tasks_queue = xQueueCreate(QUEUE_LENGTH, sizeof(dd_task_list *));
```

Figure 14: Queue creation

```
vQueueAddToRegistry( message_queue, "Message_Queue" );
vQueueAddToRegistry( active_tasks_queue, "Active_Queue" );
vQueueAddToRegistry(completed_tasks_queue, "Completed_Queue");
vQueueAddToRegistry(overdue_tasks_queue, "Overdue_Queue");
```

Figure 15: Queue Registration

Timers

```
// timer call-backs
static void xCallback_Task_Generator_1();
static void xCallback_Task_Generator_2();
static void xCallback_Task_Generator_3();
```

Figure 16: Timer callback function prototypes

```
TimerHandle_t xTask1_timer;
TimerHandle_t xTask2_timer;
TimerHandle_t xTask3_timer;
```

Figure 17: Timer handle declaration

```
xTask1_timer = xTimerCreate("timer1", pdMS_TO_TICKS(TASK_1_PERIOD_MS), pdTRUE, 0, xCallback_Task_Generator_1);
xTask2_timer = xTimerCreate("timer2", pdMS_TO_TICKS(TASK_2_PERIOD_MS), pdTRUE, 0, xCallback_Task_Generator_2);
xTask3_timer = xTimerCreate("timer3", pdMS_TO_TICKS(TASK_3_PERIOD_MS), pdTRUE, 0, xCallback_Task_Generator_3);
```

Figure 18: Timer Creation

```
xTimerStart(xTask1_timer, 0);
xTimerStart(xTask2_timer, 0);
xTimerStart(xTask3_timer, 0);
```

Figure 19: Timer start

```
//These functions are responsible for generating tasks at certain time intervals.
static void xCallback_Task_Generator_1(TimerHandle_t pxTimer) {
    uint32_t absolute_deadline = xTaskGetTickCount() + pdMS_TO_TICKS(TASK_1_PERIOD_MS); //this should be task 1 period
    TaskHandle_t t_handle;
    task_type type = PERIODIC;
    // Generate task name
    char task_str[7];

    snprintf(task_str, sizeof(task_str), "T1_%u", global_task1_count);
    xTaskCreate(Task1, task_str, configMINIMAL_STACK_SIZE, NULL, DEFAULT_PRIORITY, &t_handle); // create task
    create_dd_task(t_handle, type, task_str, absolute_deadline);
    global_task1_count++;
}
```

Figure 20: Timer callback function definition

Results

The three testbenches are run and the monitor output is recorded into excel. The results are shown in Figure 22: Test bench results. The results are as predicted and confirm the functionality of the system.

The output of the monitor is organized by list type (Active, Overdue and Complete), Task ID, release time, absolute deadline, and completion time. Followed by the total number of tasks in the list.

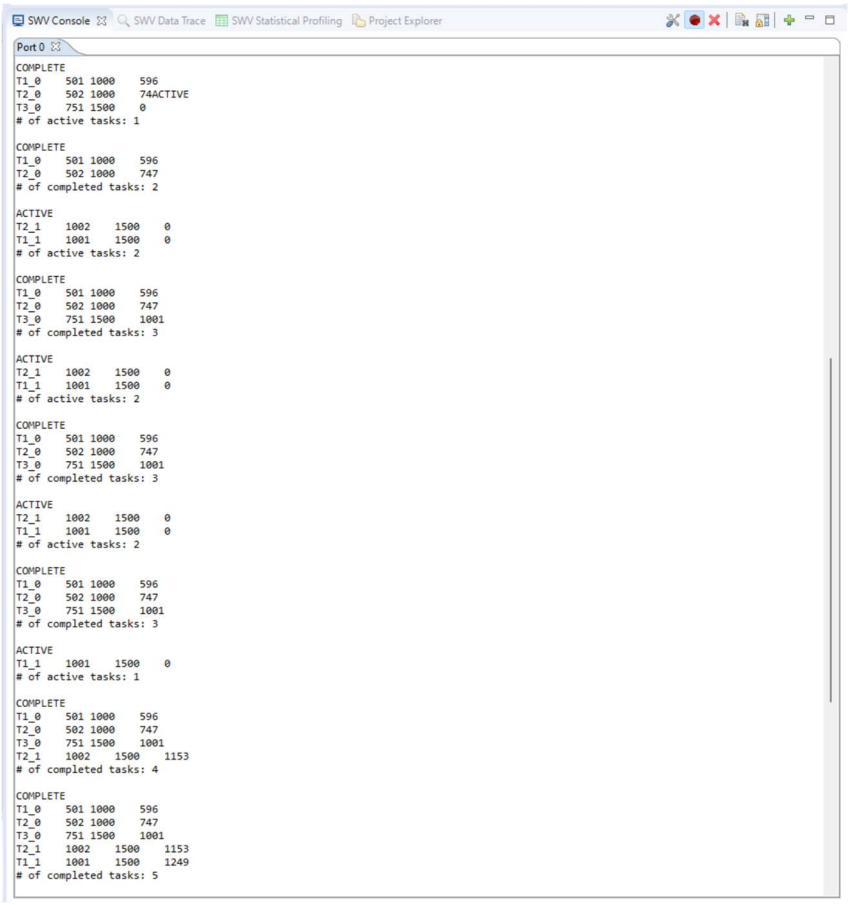

Figure 21: Monitor output

	Test Bench #1	
Task	Execution	Period (ms)
T_1	95	500
T_2	150	500
T_3	250	750

Test Bench #1						
task id	Release	Deadline	Complete	Run time	Slack	Overdue?
T1_0	501	1000	596	95	404	n
T2_0	502	1000	747	245	253	n
T3_0	751	1500	1001	250	499	n
T2_1	1002	1500	1153	151	347	n
T1_1	1001	1500	1249	248	251	n

	Test bench #2	
Task	Execution	Period (ms)
T_1	95	250
T_2	150	500
T_3	250	750

Test bench #2						
task id	Release	Deadline	Complete	Run time	Slack	Overdue?
T1_0	251	500	346	95	154	n
T1_1	502	750	597	95	153	n
T2_0	501	1000	651	150	349	n
T1_2	752	1000	847	95	153	n
T3_0	751	1500	1001	250	499	n
T1_3	1002	1250	1098	96	152	n
T2_1	1001	1500	1249	248	251	n
T1_4	1251	1500	1346	95	154	n

	Test Bench #3	
Task	Execution	Period (ms)
T_1	100	500
T_2	200	500
T_3	200	500

Test Bench #3						
task id	Release	Deadline	Complete	Run time	Slack	Overdue?
T1_0	501	1000	601	100	399	n
T3_0	503	1000	802	299	198	n
T2_0	502	1000	1003	501	-3	y
T1_1	1001	1500	1104	103	396	n
T3_1	1003	1500	1305	302	195	n
T2_1	1002	1500	1506	504	-6	y

Figure 22: Test bench results

- **Test Bench #1** includes three tasks with varying execution times and periods. All tasks in this setup completed before their deadlines, with positive slack values, indicating that the scheduler effectively prioritized tasks by their deadlines and maintained sufficient time margins. No task missed its deadline.
- **Test Bench #2** introduced a higher task load by increasing the number of task releases over the observed time frame. Despite the increased number of tasks, the scheduler maintained its performance, successfully completing all tasks within their deadlines. Positive slack values for all entries confirm that the EDF policy handled the workload efficiently, maintaining real-time constraints.
- **Test Bench 3** shows that overdue tasks can occur when the system is heavily loaded. In this case, some tasks missed their deadlines, which is indicated by the “y”. This confirms that the scheduler correctly identifies overdue tasks. Despite the missed deadlines, the code handles these overdue tasks properly.

CONCLUSION

In this project, we successfully designed and implemented a Deadline-Driven Scheduler (DDS) on top of FreeRTOS to handle tasks with hard deadlines using the Earliest Deadline First (EDF) scheduling algorithm. The DDS dynamically managed task priorities based on their deadlines, enabling FreeRTOS to schedule and execute the most urgent tasks efficiently. Through the creation of structured components, including user-defined tasks, a task generator, and a monitor task, we were able to simulate and test real-time scheduling scenarios. The system adhered strictly to functional and technical requirements, utilizing software timers, queues, and proper task isolation to ensure modularity and real-time responsiveness. Testing across various task sets demonstrated the scheduler's ability to manage periodic workloads, detect overdue tasks, and report system status accurately. Overall, this project provided valuable insight into the implementation of real-time scheduling policies in embedded systems and highlighted key design tradeoffs between performance, flexibility, and resource usage.

APPENDIX: PROJECT CODE

```
/* Standard includes. */
#include <stdint.h>
#include <stdio.h>
#include <stdlib.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"

/*-------------  Defines  --------------*/
#define DEFAULT_PRIORITY 1
#define amber_led   LED3
#define green_led   LED4
#define red_led     LED5
#define blue_led    LED6
#define QUEUE_LENGTH 10

/*-------------  Test 1  --------------*/
//#define TASK_1_PERIOD_MS 500
//#define TASK_2_PERIOD_MS 500
//#define TASK_3_PERIOD_MS 750
//#define TASK_1_EXECUTE_TIME_MS 95
//#define TASK_2_EXECUTE_TIME_MS 150
//#define TASK_3_EXECUTE_TIME_MS 250
//#define RUN_TIME 1500

///*-------------  Test 2  --------------*/
//#define TASK_1_PERIOD_MS 250
//#define TASK_2_PERIOD_MS 500
//#define TASK_3_PERIOD_MS 750
//#define TASK_1_EXECUTE_TIME_MS 95
//#define TASK_2_EXECUTE_TIME_MS 150
//#define TASK_3_EXECUTE_TIME_MS 250
//#define RUN_TIME 1500

///*-------------  Test 3  --------------*/
#define TASK_1_PERIOD_MS 500
#define TASK_2_PERIOD_MS 500
#define TASK_3_PERIOD_MS 500
```

```c
#define TASK_1_EXECUTE_TIME_MS 100
#define TASK_2_EXECUTE_TIME_MS 200
#define TASK_3_EXECUTE_TIME_MS 200
#define RUN_TIME 1600


uint32_t global_task1_count = 0;
uint32_t global_task2_count = 0;
uint32_t global_task3_count = 0;

/*------------- Type defines --------------*/
typedef enum message_type {
    CREATE_TASK,
    COMPLETE_TASK,
    GET_ACTIVE,
    GET_COMPLETE,
    GET_OVERDUE
} message_type;

typedef enum {
    PERIODIC,
    APERIODIC
} task_type;

typedef struct dd_task {
    TaskHandle_t t_handle;
    task_type type;
    char task_id[7];
    uint32_t release_time;
    uint32_t absolute_deadline;
    uint32_t completion_time;
} dd_task;

typedef struct dd_task_list {
    dd_task task;
    struct dd_task_list *next_task;
} dd_task_list;

typedef struct queue_message {
    uint32_t message_type;
    dd_task task;
} queue_message;

const dd_task default_task_const = {0, 0, "xx_xxx", 0, 0, 0};
```

```c
/*-------------  Function Prototypes  --------------*/
// timer call-backs
static void xCallback_Task_Generator_1();
static void xCallback_Task_Generator_2();
static void xCallback_Task_Generator_3();
// user tasks
static void Task1(void *pvParameters);
static void Task2(void *pvParameters);
static void Task3(void *pvParameters);
// DDR system
void create_dd_task(TaskHandle_t t_handle, task_type type, char* task_id, uint32_t absolute_deadline);
static void DD_Task_Scheduler(void *pvParameters);
static void DD_Task_Monitor(void *pvParameters);
dd_task_list *get_active_dd_task_list(void);
dd_task_list *get_overdue_dd_task_list(void);
dd_task_list *get_complete_dd_task_list(void);
void addToSortedList(dd_task_list *list, dd_task_list *task);
void complete_dd_task(TaskHandle_t t_handle);
static void prvSetupHardware( void );

/*------------- Timer and Queue handle declaration -------------*/
TimerHandle_t xTask1_timer;
TimerHandle_t xTask2_timer;
TimerHandle_t xTask3_timer;
xQueueHandle message_queue = 0;
xQueueHandle active_tasks_queue = 0;
xQueueHandle completed_tasks_queue = 0;
xQueueHandle overdue_tasks_queue = 0;
xQueueHandle get_completed_list_queue = 0;

/*-----------------------------------------------------------*/
int main(void){
    /* Initialize LEDs */
    STM_EVAL_LEDInit(amber_led);
    STM_EVAL_LEDInit(green_led);
    STM_EVAL_LEDInit(red_led);
    STM_EVAL_LEDInit(blue_led);

    prvSetupHardware();

    message_queue = xQueueCreate(QUEUE_LENGTH, sizeof(queue_message));
    active_tasks_queue = xQueueCreate(QUEUE_LENGTH, sizeof(dd_task_list *));
    completed_tasks_queue = xQueueCreate(QUEUE_LENGTH, sizeof(dd_task_list *));
    overdue_tasks_queue = xQueueCreate(QUEUE_LENGTH, sizeof(dd_task_list *));
```

```
    xTask1_timer = xTimerCreate("timer1", pdMS_TO_TICKS(TASK_1_PERIOD_MS), pdTRUE, 0, xCallback_Task_Generator_1);
    xTask2_timer = xTimerCreate("timer2", pdMS_TO_TICKS(TASK_2_PERIOD_MS), pdTRUE, 0, xCallback_Task_Generator_2);
    xTask3_timer = xTimerCreate("timer3", pdMS_TO_TICKS(TASK_3_PERIOD_MS), pdTRUE, 0, xCallback_Task_Generator_3);

    xTaskCreate(DD_Task_Scheduler, "Scheduler", configMINIMAL_STACK_SIZE, NULL, 4, NULL);
    xTaskCreate(DD_Task_Monitor, "Monitor", configMINIMAL_STACK_SIZE, NULL, 3, NULL);

    vQueueAddToRegistry( message_queue, "Message_Queue" );
    vQueueAddToRegistry( active_tasks_queue, "Active_Queue" );
    vQueueAddToRegistry(completed_tasks_queue, "Completed_Queue");
    vQueueAddToRegistry(overdue_tasks_queue, "Overdue_Queue");

    xTimerStart(xTask1_timer, 0);
    xTimerStart(xTask2_timer, 0);
    xTimerStart(xTask3_timer, 0);

    /* Start the tasks */
    vTaskStartScheduler();

    return 0;
}

/*------------- Task Definitions -------------*/
static void DD_Task_Scheduler(void *pvParameters) {
    queue_message message;
    dd_task_list *cur;
    dd_task_list *prev;
    dd_task_list *active_head = NULL; //for active task
    dd_task_list *completed_head = NULL; //for completed task
    dd_task_list *overdue_head = NULL;  //for overdue task

    while(1) {
        // Handle overdue tasks
        TickType_t time_cur = xTaskGetTickCount();

        if ( time_cur > pdMS_TO_TICKS(RUN_TIME) ) {
            printf("END TEST");
            while(1) {
                // Do nothing
            }
        }
        cur = active_head;
        prev = active_head;
        if (cur != NULL) {
            if (time_cur > cur->task.absolute_deadline) {
```

```
            // TASK IS OVERDUE
            dd_task_list *new_overdue_task;
            new_overdue_task = cur;
            if(active_head->next_task == NULL) {
                active_head = NULL;
            }
            else {
                active_head = active_head->next_task;
            }
            // if there is a task to work on, start it
            if (active_head != NULL) {
                vTaskResume(active_head->task.t_handle);
            }

            // adding task to overdue task list
            new_overdue_task->next_task = NULL;
            cur = overdue_head;
            if (overdue_head == NULL) {
                overdue_head = new_overdue_task;
            }
            else {
                while (cur->next_task != NULL) {
                    cur = cur->next_task;
            }
            cur->next_task = new_overdue_task;
        }
    } // end_if "over due tasks"
}// end if cur != Null

    // Wait for new message
    if (xQueueReceive(message_queue, &(message), 0) == pdPASS) {

        switch (message.message_type) {

            case CREATE_TASK:
                cur = active_head;
                dd_task_list *new_task = (dd_task_list *)pvPortMalloc(sizeof(dd_task_list));
                new_task->task = default_task_const;
                new_task->next_task = NULL;
                // assign task release time
                new_task->task = message.task;
                new_task->task.release_time = xTaskGetTickCount(); // current ticks since scheduler start

                // Add new task to list
                // if list is empty, make new element the head
```

```
    if (active_head == NULL) {
        active_head = new_task;
        active_head->next_task = NULL;
    }
    // else, list is not empty
    else {
        // place APERIODIC tasks at the end of the list
        if (new_task->task.type == APERIODIC) {
            while (cur->next_task != NULL) {
                cur = cur->next_task;
            }
            cur->next_task = new_task;
        }
        // if task to be inserted has lower deadline than head of list, insert at the front
        else if (new_task->task.absolute_deadline < cur->task.absolute_deadline) {
            new_task->next_task = cur;
            vTaskSuspend(cur -> task.t_handle);
            vTaskPrioritySet(active_head->task.t_handle, DEFAULT_PRIORITY);
            active_head = new_task;
        }
        // if the deadlines are equal place after head
        else if (new_task->task.absolute_deadline == cur->task.absolute_deadline){
            new_task->next_task = active_head->next_task;
            active_head->next_task = new_task;
        }
        else {
            // if task should not be inserted at start, loop through and find location
            // while next node is not null and next new task deadline is less than current task deadline
            addToSortedList(active_head, new_task);
        }
    }

    // if there is a task to work on, start it
    if (active_head != NULL) {
        //increase task priority
        vTaskPrioritySet(active_head->task.t_handle, 2);
        // resume task
        vTaskResume(active_head->task.t_handle);

    }
    break;

case COMPLETE_TASK:
    // Check and remove task from active task list
    cur = active_head;
```

```
prev = active_head;
dd_task_list *new_completed_task = NULL;
while (cur != NULL) {
    if (message.task.t_handle == cur->task.t_handle) {
        new_completed_task = cur;
        new_completed_task->task.completion_time = message.task.completion_time;
        // remove from active list
        if (prev == cur) {
            // current is the head of the list
            active_head = cur->next_task;
        }
        else if (cur->next_task == NULL) {
            // current is last in list
            cur = NULL;
            prev->next_task = NULL;
        }
        else {
            // current is not last in list
            prev->next_task = cur->next_task;
        }
        break; // break while
    }
    prev = cur;
    cur = cur->next_task;
}

// if there is a task to work on, start it
if (active_head != NULL) {
    vTaskPrioritySet(active_head->task.t_handle, 3);
    vTaskResume(active_head->task.t_handle);
}
// Check and remove task from overdue task list
if ( new_completed_task ==  NULL) {
    cur = overdue_head;
    prev = overdue_head;
    while (cur != NULL) {
        if (message.task.t_handle == cur->task.t_handle) {
            new_completed_task = cur;
            new_completed_task->task.completion_time = message.task.completion_time;
            // remove from active list
            if (prev == cur) {
                // current is the head of the list
                overdue_head = cur->next_task;
            }
            else if (cur->next_task == NULL) {
```

```
                    // current is last in list
                    cur = NULL;
                    prev->next_task = NULL;
                }
                else {
                    // current is not last in list
                    prev->next_task = cur->next_task;
                }
                break; // break while
            }
            prev = cur;
            cur = cur->next_task;
        }
    } // end if check overdue

    // adding task to completed task list
    new_completed_task->next_task = NULL;
    cur = completed_head;
    if (completed_head == NULL) {
                completed_head = new_completed_task;
            }
    else {
        while (cur->next_task != NULL) {
            cur = cur->next_task;
        }
        cur->next_task = new_completed_task;
    }
    break;

case GET_ACTIVE:
    if (xQueueSend(active_tasks_queue, &active_head, 500)) {
    }
    break;

case GET_COMPLETE:
    if (xQueueSend(completed_tasks_queue, &completed_head, 500)) {
    }
    break;

case GET_OVERDUE:
    if (xQueueSend(overdue_tasks_queue, &overdue_head, 500)) {
    }
    break;

default:
```

```
                    break;
            } // end switch
        } // end if msg received
        vTaskDelay(pdMS_TO_TICKS(1));
    } // end while
} // end DD_Task_Scheduler


static void DD_Task_Monitor(void *pvParameters) {
    int num_active;
    int num_complete;
    int num_overdue;
    dd_task_list *cur_active;
    dd_task_list *cur_complete;
    dd_task_list *cur_overdue;

    while(1) {
        num_active = 0;
        num_complete = 0;
        num_overdue = 0;
        cur_active = get_active_dd_task_list();
        cur_complete = get_complete_dd_task_list();
        cur_overdue = get_overdue_dd_task_list();

        //--------active tasks
        if (cur_active!= NULL){
            printf("ACTIVE\n");
            while(cur_active != NULL) {
                printf("%s\t%u\t%u\t%u\n", cur_active->task.task_id, cur_active->task.release_time, cur_active-
>task.absolute_deadline, cur_active->task.completion_time);
                num_active++;
                cur_active = cur_active->next_task;
            }
            printf("# of active tasks: %d\n\n", num_active);
        }

        //--------completed tasks
        if (cur_complete != NULL){
            printf("COMPLETE\n");
            while(cur_complete != NULL) {
                printf("%s\t%u\t%u\t%u\n", cur_complete->task.task_id, cur_complete->task.release_time, cur_complete-
>task.absolute_deadline, cur_complete->task.completion_time);
                num_complete++;
                cur_complete = cur_complete->next_task;
            }
```

```
        printf("# of completed tasks: %d\n\n", num_complete);
        }

        //--------overdue tasks
        if (cur_overdue != NULL) {
            printf("OVERDUE\n");
            while(cur_overdue != NULL) {
                printf("%s\t%u\t%u\t%u\n", cur_overdue->task.task_id, cur_overdue->task.release_time, cur_overdue-
>task.absolute_deadline, cur_overdue->task.completion_time);
                num_overdue++;
                cur_overdue = cur_overdue->next_task;
            }
        printf("# of overdue Tasks: %d\n\n", num_overdue);
        }
        vTaskDelay(pdMS_TO_TICKS(1));
    }// end while
}// end monitor


void complete_dd_task(TaskHandle_t task_handle) {
    dd_task task;
    queue_message message;

    task.t_handle = task_handle;
    task.completion_time = xTaskGetTickCount();
    message.message_type = COMPLETE_TASK;
    message.task = task;
    xQueueSend(message_queue, &message, 0);
    vTaskDelete(task_handle);
}

void create_dd_task(TaskHandle_t t_handle, task_type type, char* task_id, uint32_t absolute_deadline) {
    vTaskSuspend(t_handle);
    queue_message message;
    dd_task task;

    task.absolute_deadline = absolute_deadline;
    task.completion_time = 0;
    strcpy(task.task_id, task_id);
    task.type = type;
    task.t_handle = t_handle;
    message.message_type = CREATE_TASK;
    message.task = task;
    xQueueSend(message_queue, &message, 0);
}
```

```
/*--------------------  User code DD_tasks -------------------------------------*/
// RED LED TASK
// RED LED TASK
static void Task1(void *pvParameters) {
    TickType_t elapsed_time, current_time;
    TickType_t exec_time = pdMS_TO_TICKS(TASK_1_EXECUTE_TIME_MS);
    TickType_t start_time = xTaskGetTickCount();
    TaskHandle_t my_task_handle;

    STM_EVAL_LEDOn(red_led);

    while (1) {
        current_time = xTaskGetTickCount();
        elapsed_time = current_time - start_time;
        if (elapsed_time >= exec_time) {
            break;
        }
    }

    STM_EVAL_LEDOff(red_led);

    my_task_handle = xTaskGetCurrentTaskHandle();
    complete_dd_task(my_task_handle);
    // wait to be deleted
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(100000));
    }
}

// GREEN LED TASK
static void Task2(void *pvParameters) {
    TickType_t elapsed_time, current_time;
    TickType_t exec_time = pdMS_TO_TICKS(TASK_2_EXECUTE_TIME_MS);
    TickType_t start_time = xTaskGetTickCount();
    TaskHandle_t my_task_handle;

    STM_EVAL_LEDOn(green_led);

    while (1) {
        current_time = xTaskGetTickCount();
        elapsed_time = current_time - start_time;
        if (elapsed_time >= exec_time) {
            break;
        }
```

```
    }
    STM_EVAL_LEDOff(green_led);
    my_task_handle = xTaskGetCurrentTaskHandle();
    complete_dd_task(my_task_handle);

    while (1) {
        vTaskDelay(pdMS_TO_TICKS(100000));
    }
}

// BLUE LED TASK
static void Task3(void *pvParameters) {
    TickType_t elapsed_time, current_time;
    TickType_t exec_time = pdMS_TO_TICKS(TASK_3_EXECUTE_TIME_MS);
    TickType_t start_time = xTaskGetTickCount();
    TaskHandle_t my_task_handle;

    STM_EVAL_LEDOn(blue_led);

    while (1) {
        current_time = xTaskGetTickCount();
        elapsed_time = current_time - start_time;
        if (elapsed_time >= exec_time) {
            break;
        }
    }

    STM_EVAL_LEDOff(blue_led);
    my_task_handle = xTaskGetCurrentTaskHandle();
    complete_dd_task(my_task_handle);

    while (1) {
        vTaskDelay(pdMS_TO_TICKS(100000));
    }
}

/*---------------------  Get task list interface functions  -----------------------------------*/

dd_task_list* get_active_dd_task_list(void) {
    queue_message message;
    message.message_type = GET_ACTIVE;
    xQueueSend(message_queue, &message, 0);
    dd_task_list *head;

    if(xQueueReceive(active_tasks_queue, &head, pdMS_TO_TICKS(2))) {
```

```
        return head;
    }
    else {
        return NULL;
    }
}

dd_task_list* get_complete_dd_task_list(void) {
    queue_message message;
    message.message_type = GET_COMPLETE;
    xQueueSend(message_queue, &message, 0);
    dd_task_list *head;

    if(xQueueReceive(completed_tasks_queue, &head, pdMS_TO_TICKS(2))) {
        return head;
    }
    else {
        return NULL;
    }
}

dd_task_list* get_overdue_dd_task_list(void) {
    queue_message message;
    message.message_type = GET_OVERDUE;
    xQueueSend(message_queue, &message, 0);
    dd_task_list *head;

    if(xQueueReceive(overdue_tasks_queue, &head, pdMS_TO_TICKS(2))) {
        return head;
    }
    else {
        return NULL;
    }
}


/*---------------------  Scheduler utility function  ----------------------*/
/**
 * Adds to list based on absolute deadline, negates the need to sort after insertion
 * used for active list (ordered by deadline)
 */
void addToSortedList(dd_task_list *list, dd_task_list *task) {
    dd_task_list *node = list;
    dd_task_list *prev_task = list;
    dd_task_list *newTask;
```

```
    newTask = task;

    while(node->next_task != NULL && (node->task.absolute_deadline < newTask->task.absolute_deadline)) {
        prev_task = node;
        node = node->next_task;
    }
    if (prev_task == node) {
        newTask->next_task = node;
    }
    else if (node->next_task == NULL) {
        node->next_task = newTask;
        newTask->next_task = NULL;
    }
    else {
        newTask->next_task = node;
        prev_task->next_task = newTask;
    }
}


/*---------------------  Timer Call back functions  -------------------------------------*/

//These functions are responsible for generating tasks at certain time intervals.
static void xCallback_Task_Generator_1(TimerHandle_t pxTimer) {
    uint32_t absolute_deadline = xTaskGetTickCount() + pdMS_TO_TICKS(TASK_1_PERIOD_MS); //this should be task 1 period
    TaskHandle_t t_handle;
    task_type type = PERIODIC;
    // Generate task name
    char task_str[7];

    snprintf(task_str, sizeof(task_str), "T1_%u", global_task1_count);
    xTaskCreate(Task1, task_str, configMINIMAL_STACK_SIZE, NULL, DEFAULT_PRIORITY, &t_handle); // create task
    create_dd_task(t_handle, type, task_str, absolute_deadline);
    global_task1_count++;
}


static void xCallback_Task_Generator_2(TimerHandle_t pxTimer) {
    uint32_t absolute_deadline = xTaskGetTickCount() + pdMS_TO_TICKS(TASK_2_PERIOD_MS);//this should be task 2
period    TaskHandle_t t_handle;
    TaskHandle_t t_handle;
    task_type type = PERIODIC;
    // Generate task name
    char task_str[7];
```

```
    snprintf(task_str, sizeof(task_str), "T2_%u",global_task2_count);
    xTaskCreate(Task2, task_str, configMINIMAL_STACK_SIZE, NULL, DEFAULT_PRIORITY, &t_handle); // create task
    create_dd_task(t_handle, type, task_str, absolute_deadline);
    global_task2_count++;
}


static void xCallback_Task_Generator_3(TimerHandle_t pxTimer) {
    uint32_t absolute_deadline = xTaskGetTickCount() + pdMS_TO_TICKS(TASK_3_PERIOD_MS); //this should be task 3
period    TaskHandle_t t_handle;
    TaskHandle_t t_handle;
    task_type type = PERIODIC;
    // Generate task name
    char task_str[7];

    snprintf(task_str, sizeof(task_str), "T3_%u",global_task3_count);
    xTaskCreate(Task3, task_str, configMINIMAL_STACK_SIZE, NULL, DEFAULT_PRIORITY, &t_handle); // create task
    create_dd_task(t_handle, type, task_str, absolute_deadline);
    global_task3_count++;
}
```