UNIVERSITY OF VICTORIA

Department of Electrical and Computer Engineering

ECE 455 – Real Time Computer Systems

Project 2: Deadline-Driven Scheduler

Report Submitted on: April 27, 2021

Name: Kutay Cinar - V00******

1.0 Introduction

The goal of this project is to create a custom Deadline-Driven Scheduler (DDS) to handle tasks with hard deadlines in a dynamic way using FreeRTOS tasks, queues, and timers.

This system follows the system overview diagram provided in the lab slides which shows the interactions between F-Tasks, queues, and DDS functions as recommended to be implemented by the lab manual [1]:

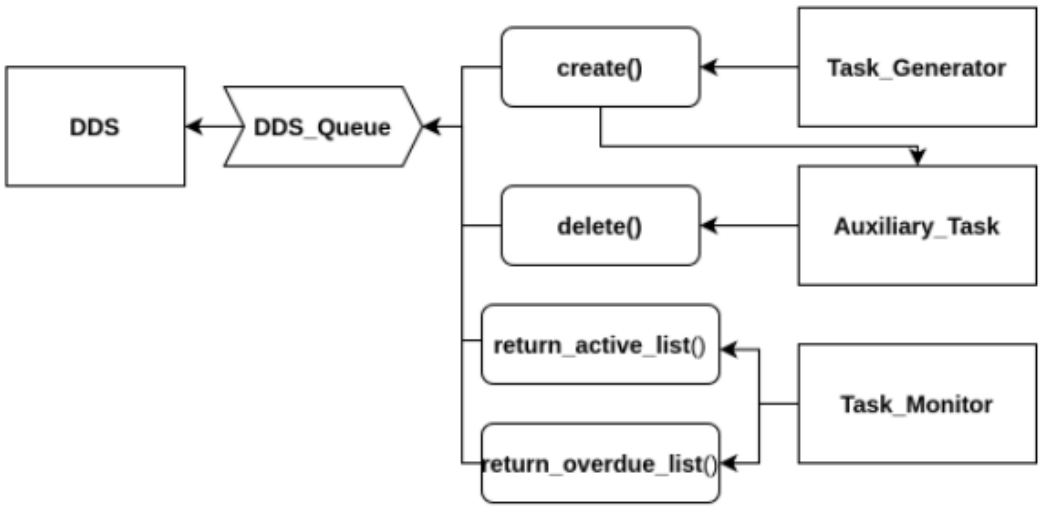


Fig 1: Whole System from the lab slides of project [2]

There are four major parts to implement with two being types of tasks (user and generator) for the system to run and behave as intended.

The four major components are:

Deadline-Driven Scheduler: Deadline-Driven Scheduler task is the task scheduler in this system and implements the EDF algorithm and controls the priorities of user-defined F-tasks from an actively managed list of DD-Tasks.

User-Defined Tasks: User Tasks contain the actual deadline-sensitive application code written by the user but in this implementation are mainly busy loops for system demonstration. They are scheduled by Task Generators and run by the DD-Scheduler.

Task Generators: Task Generators periodically create (request) DD-Tasks that need to be scheduled by the DD Scheduler. Task Generators use software timers to run at a period to schedule periodic tasks, or run one-shot timers to schedule aperiodic tasks.

Monitor Task: Monitor Task is also needed for system evaluation to see the currently running, completed, and overdue tasks. Monitor Task is a FreeRTOS task in this system that runs and extracts information from the DD-Scheduler and reports the list of active, completed and overdue tasks.

2.0 Design Solution

This project involved 4 major parts. These were Deadline-Driven Scheduler, User-Defined Tasks, Deadline-Driven Task Generator, and Monitor Task.

2.1 Deadline-Driven Scheduler

Deadline-Driven Scheduler uses EDF (Earliest Deadline First) algorithm to schedule tasks and to run them. Below is a diagram of the system demonstrating how each of the tasks interact with each other through the DD-Scheduler.

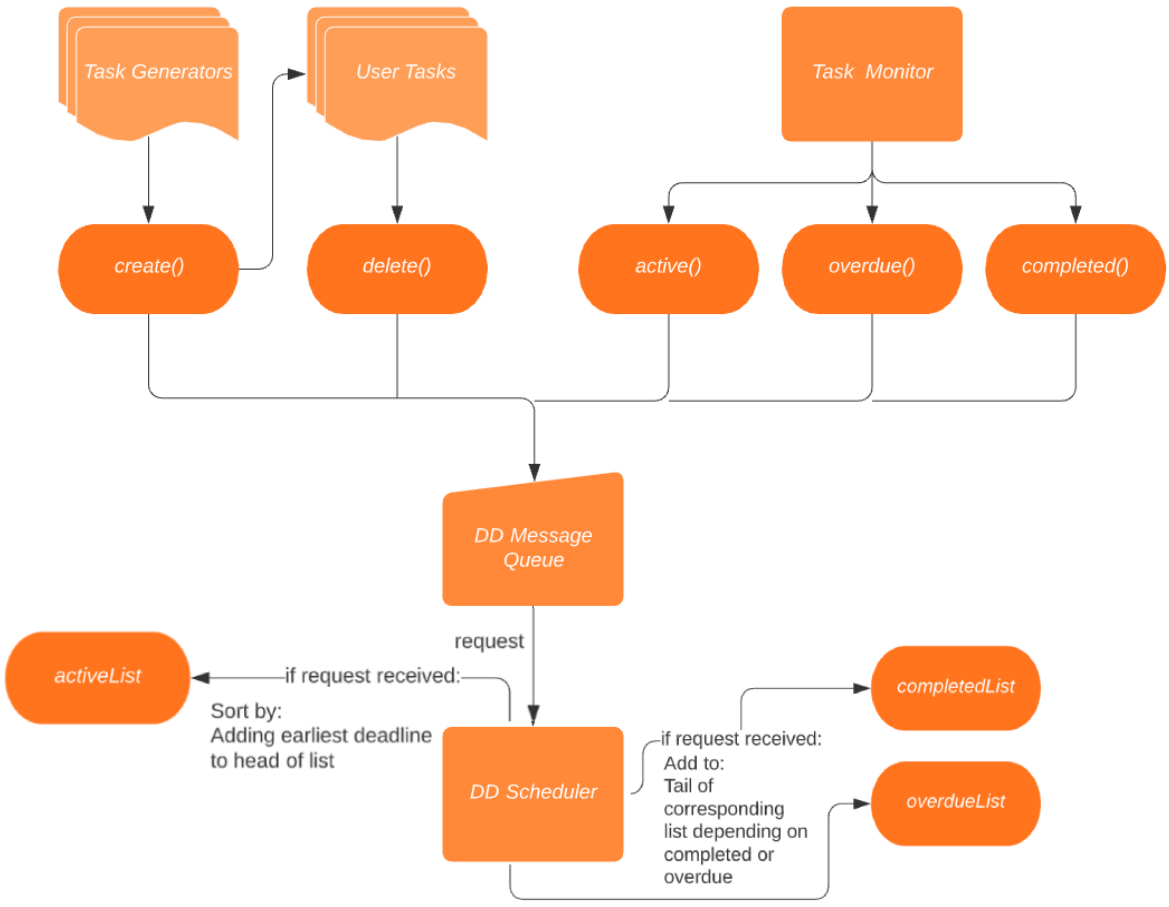


Fig 2: Flow chart of how the DD Scheduler works

Following this point, the tasks run by the DD-Scheduler are referred to as DD-Task and tasks run by the FreeRTOS are referred to as F-Task to differentiate between them in this report. Below is a list of all F-Tasks and their priority levels.

F-Task	Priority
DD Scheduler	PRIORITY_MAX
Monitor Task	PRIORITY_MAX - 1
Task Generator(s)	PRIORITY_HIGH
User Task(s)	PRIORITY_MIN

Table 1: List of all F-Tasks and their priority levels.

In FreeRTOS and my implementation MAX priority is equal to 4, HIGH priority is equal to 2, and MIN priority is equal to 1. They are defined in my code as follows:

```
#define PRIORITY_MAX          4
#define PRIORITY_HIGH         2
#define PRIORITY_MIN          1
```

Note: Monitor Task is given a priority level one less than DD-Scheduler in order for it not to be interrupted by User Tasks or Generator Tasks. If using MIN priority for Monitor Task, in the middle of the print statements generated by the monitor task, print statements from other tasks could be executed, making the output convoluted. Monitor Task is assigned MAX - 1 priority in order to prevent this.

2.1.1 DD-Task

DD-Task is a data structure that follows the template given in the report as follows:

```
typedef enum task_type task_type;

enum task_type {
	PERIODIC,
	APERIODIC
};

typedef struct dd_task dd_task;

struct dd_task {
	TaskHandle_t t_handle;
	task_type type;
	uint32_t task_id;
	uint32_t release_time;
	uint32_t absolute_deadline;
	uint32_t completion_time;
	uint32_t period;
};
```

Note: There is an additional variable in dd_task for period which is used for checking in the DD-Scheduler whether a dd_task that completed missed its deadline or not.

2.1.2 DD-Task List

DD-Task List is a data type for storing a list of tasks. The following singly-linked list structure recommended from the lab manual is used:

```
typedef struct dd_task_list dd_task_list;

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

There are three lists in the DD-Scheduler that use this structure.

- Active Task List
- Completed Task List
- Overdue Task List

Active Task List: Contains the list of tasks that need to be scheduled by the DD-Scheduler.

Completed Task List: Contains the list of tasks that have completed *before* their deadlines.

Overdue Task List: Contains the list of tasks that have completed *after* their deadlines.

Active Task List is sorted in the deadline driven scheduler at the release and completion of a user task. Completed/Overdue Task Lists do not need to be sorted and are added at the tail.

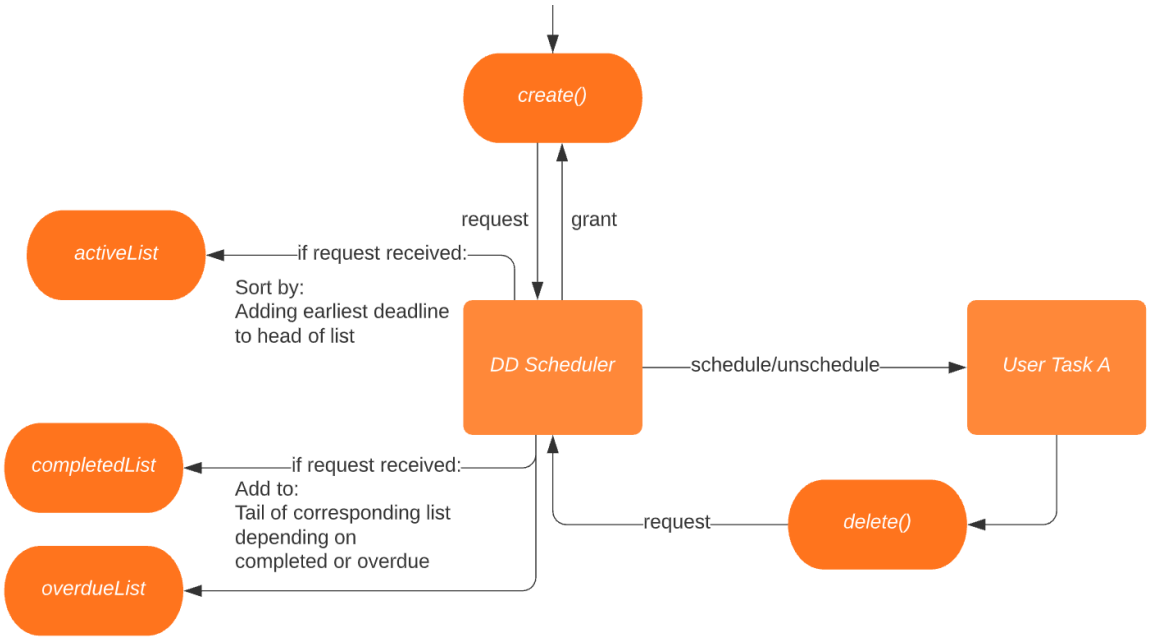


Fig 3: Flow chart of the DD-Task sorting algorithm.

2.1.4 DD-Message

DD-Scheduler is an F-Task with the highest priority and waits in a suspended state until it receives a DD-Message in the DD-Queue (Message Queue).

```
typedef enum message_type message_type;

enum message_type {
	create,
	delete,
	active,
	completed,
	overdue,
};
typedef struct dd_message dd_message;

struct dd_message {
	message_type type;
	dd_task task;
};
```

DD Scheduler works by checking with message is received in a switch/case statements and executes by either creating, deleting, or providing a task list.

2.1.5 Functions

DD-Scheduler is un-suspended when a call is made from one of its interface functions. These functions are:

Create / create_dd_task

In this implementation, all user and generator tasks are created from the start. User Tasks are then suspended right away and are unsuspended with the use of create_dd_task() function by task generators for the DD Scheduler to schedule them.

```
void create_dd_task(
      TaskHandle_t t_handle,
      task_type type,
      uint32_t task_id,
      uint32_t absolute_deadline
);
```

Delete / delete_dd_task

In this implementation delete works by suspending the User Task that currently has finished executing. Delete or delete_dd_task() function is called by User Tasks when they have completed their execution times. This allows the User Task to be unsuspended again in order to re-run its periodic execution time.

```
void delete_dd_task(uint32_t task_id);
```

Active / get_active_dd_task_list

Return a reference of the Active Task List dd_task_list struct from DD Scheduler.

```
dd_task_list** get_active_dd_task_list(void);
```

Completed / get_completed_dd_task_list

Return a reference of the Completed Task List dd_task_list struct from DD Scheduler.

```
dd_task_list** get_completed_dd_task_list(void);
```

Overdue / get_overdue_dd_task_list

Return a reference of the Overdue Task List dd_task_list struct from DD Scheduler.

```
dd_task_list** get_overdue_dd_task_list(void);
```

2.2 User Task

User tasks run for a predetermined execution time (busy while loop) and are scheduled by Task Generators. After completing their executions they send a delete() request to the DD-Scheduler to become suspended, and if they are periodic, they are un-suspended by their respective task generator to run again.

2.3 Task Generator

This implementation uses dedicated task generator for each user task as per one of the recommended approaches from the lab manual:

Figure 4: Possible implementations for generating DD-Tasks. [1]

Deadline-Task Generators are scheduled at the start and use the create() function to interface with DD Scheduler to schedule their respective user tasks. Once the message is sent, in the format of a dd_message, it then waits for a response back from DD Scheduler in a message queue for a successful grant request and then suspends itself where it waits to be unsuspended by the software timer to schedule a task again.

Note: Task Generators use software timers to schedule periodic tasks. For aperiodic tasks, the timer can be set to a one-shot timer.

A flowchart of the workflow how a Task Generator, called A, would work is demonstrated below:

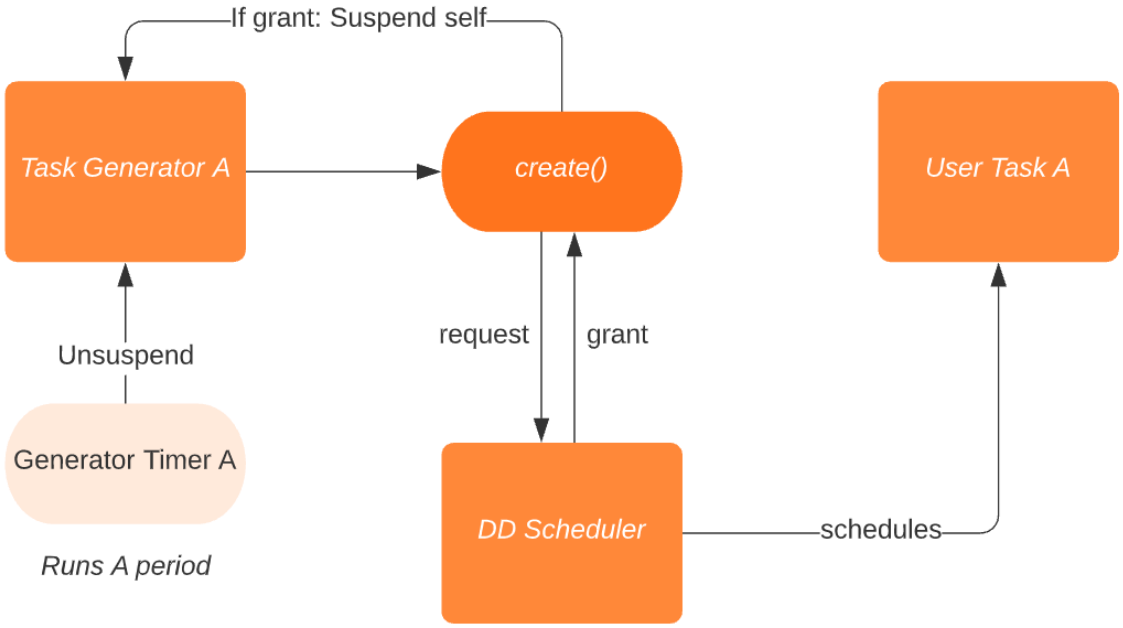


Figure 5: Flow chart of how the Deadline-Task Generator works.

2.4 Monitor Task

Monitor Task is an auxiliary F-Task responsible for reporting the following system information:

- Number of active DD-Tasks
- Number of completed DD-Tasks
- Number of overdue DD-Tasks

Monitor Task collects this information from the DD Scheduler by using get_active_dd_task_list, get_complete_dd_task_list, and get_overdue_dd_task_list functions. It then reports the number of tasks in each list by printing it to the console.

Monitor Task is allowed to execute even if there are active or overdue tasks, thus, has priority one less than DD Scheduler in order not to be interrupted by other tasks.Monitor Task uses a software timer to run at X intervals to report the task lists and system information.

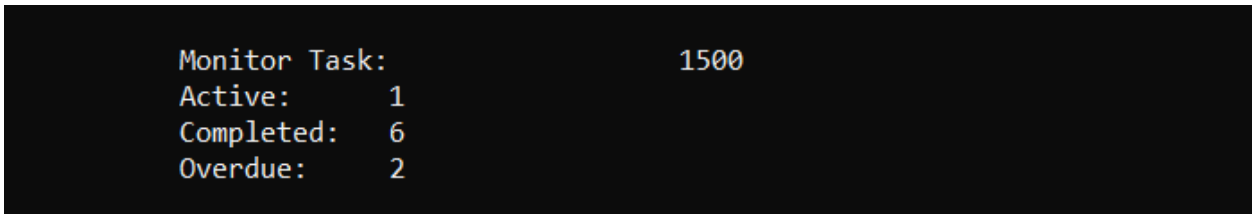


Fig 6: Example output of Monitor Task

3.0 Discussion

The program was tested on the online ECE 455 lab computers using TrueSTUDIO, mainly using ws14-16. Towards the end, my home machine was then used for developing the code due to ws14-16 or other lab machines being widely in-use and unavailable. Visual Studio 2019 is installed to accomplish this and runs FreeRTOS with the same configuration from the lab computer.

Performance of the DDS is similar on lab machines and the machine used to demonstrate during the lab demo. After running the test benches multiple times, sometimes FreeRTOS executed multiple tasks in a single tick. Adding print statements minimized this by adding a little overhead where tasks, such as generator tasks used up a whole tick to print

3.1 System Evaluation

For evaluating the system, as per the lab manual, test benches 1, 2, and 3 are integrated into the source code with their respective reference running times. The functionality of the DDS is tested using each of the three test benches in Table 3.

Table 3: DDS Test Benches.

	Test Bench #1		*Test Bench #2*		*Test Bench #3*	
Task	**Execution Time (ms)**	**Period (ms)**	**Execution Time (ms)**	**Period (ms)**	**Execution Time (ms)**	**Period (ms)**
t_1	95	500	95	250	100	500
t_2	150	500	150	500	200	500
t_3	250	750	250	750	200	500

3.2 Testing/Results

This section contains the results obtained after performing the lab demo.

Test Bench #1:

Below is the table containing the measured and expected times of task releases and completions in milliseconds (ms).

```
C:\Users\gmail\Documents\FreeRTOSv202012.00\FreeRTOS\Demo\WIN32-MSVC\Debug\RTOSDemo.exe
Event #          Event          Measured Time (ms)      Expected Time (ms)
================================================================================
   1      Task 1 released              1                       0
   2      Task 2 released              2                       0
   3      Task 3 released              3                       0
   4      Task 1 complete              99                      95
   5      Task 2 complete              249                     245
   6      Task 3 complete              499                     495
   7      Task 1 released              500                     500
   8      Task 2 released              500                     500
   9      Task 1 complete              595                     595
   10     Task 2 complete              745                     745
   11     Task 3 released              750                     750
   12     Task 3 complete              1000                    1000
   13     Task 1 released              1000                    1000
   14     Task 2 released              1000                    1000
   15     Task 1 complete              1095                    1095
   16     Task 2 complete              1245                    1245
   17     Task 1 released              1500                    1500
```

Monitor Task Results at one HyperPeriod

```
         Monitor Task:                1500
         Active:      0
         Completed:   8
         Overdue:     0
```

It can be seen that every task is released and completed at or close to the expected times with no overdue tasks. (Note: Monitor Task runs before new tasks are scheduled at 1500ms).

Test Bench #2:

Below is the table containing the measured and expected times of task releases and completions in milliseconds (ms).

C:\Users\gmail\Documents\FreeRTOSv202012.00\FreeRTOS\Demo\WIN32-MSVC\Debug\RTOSDemo.exe

Event #	Event	Measured Time (ms)	Expected Time (ms)
1	Task 1 released	1	0
2	Task 2 released	2	0
3	Task 3 released	3	0
4	Task 1 complete	99	95
5	Task 2 complete	249	245
6	Task 1 released	250	250
7	Task 1 complete	345	345
8	Task 2 released	500	500
9	Task 1 released	500	500
10	Task 3 complete	594	590
11	Task 1 complete	689	685
12	Task 3 released	750	750
13	Task 1 released	750	750
14	Task 2 complete	839	835
15	Task 1 complete	934	930
16	Task 2 released	1000	1000
17	Task 1 released	1000	1000
18	Task 1 complete	1095	1095
19	Task 1 released	1250	1250
20	Task 3 complete	1279	1245
21	Task 2 complete	1429	1425
22	Task 1 released	1500	1500
23	Task 2 released	1500	1500
24	Task 3 released	1500	1500

Monitor Task Results at one Hyper-Period:

```
Monitor Task:                1500
Active:      1
Completed:   10
Overdue:     0
```

At the first hyper-period, there is a currently active task.

Allowing the monitor task to continue executing at the next two hyper-periods, we can see that this active task at the hyper-period becomes overdue and is then added to the overdue list.

Monitor Task Results at two Hyper-Period:

```
Monitor Task:                3000
Active:      1
Completed:   20
Overdue:     1
```

Monitor Task Results at three Hyper-Period:

```
Monitor Task:                4500
Active:      1
Completed:   30
Overdue:     2
```

Looking closely at a hyper-period of 1500 ms, task 1 will arrive 1500 / 250 = 5 times, task 2 will arrive 1500 / 500 = 3 times, and task 3 will arrive 1500 / 750 = 2 times.

Task 1: 95ms * 5 times = 475ms

Task 2: 150ms * 3 times = 450ms

Task 3: 250ms * 2 times = 500ms

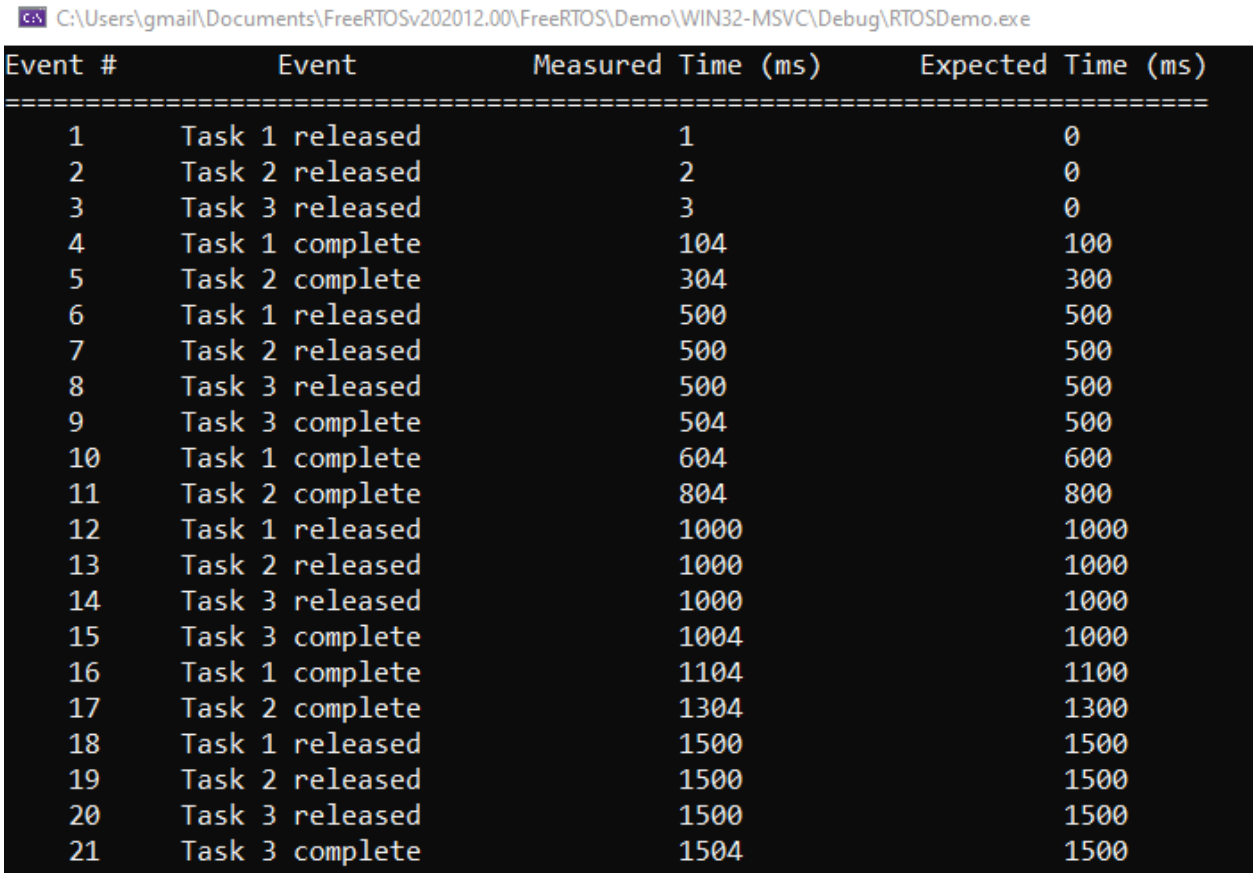
Adding all the run times will equal to a total running time of 475 + 450 + 500 = 1425 ms.

```
20      Task 3 complete              1279                    1245
21      Task 2 complete              1429                    1425
```

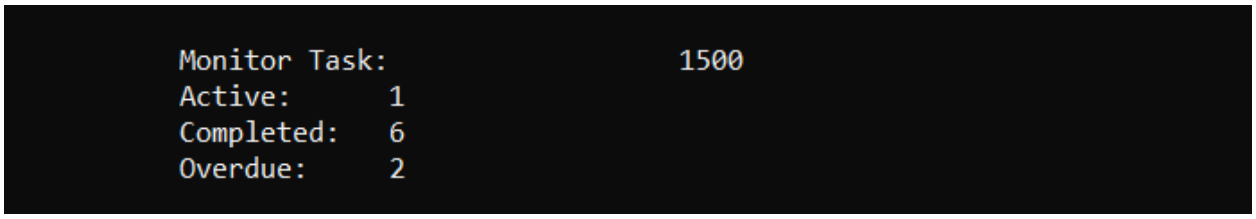
However, by inspection we can see that there is overhead during task release/completion times and this adds up towards the end causing one of the tasks to be overdue.

Test Bench #3:

Below is the table containing the measured and expected times of task releases and completions in milliseconds (ms).



Monitor Task Results at one 1500ms Interval:



Monitor Task Results at two 1500ms Interval:

```
Monitor Task:                3000
Active:      1
Completed:   12
Overdue:     5
```

Monitor Task Results at three 1500ms Interval:

```
Monitor Task:                4500
Active:      1
Completed:   18
Overdue:     8
```

Test bench has a hyper-period of 500ms, but would require perfect release/completion times for all three tasks as they all take a combined 500ms complete in a 500ms hyper-period.

The above results taken at 1500ms intervals, to be consistent with previous test benches, further indicate that since the DDS does not run in a perfect environment and has overhead, that tasks will start becoming overdue.

4.0 Limitations and Possible Improvements

Limitations in this project include the number of tasks, generators and software timers based on the system memory. ECE 455 lab computers sometimes did not allow for scheduling all the tasks. Potential improvements can be made by adding new features to create a more comprehensive deadline-driven scheduling system such as adding more types of tasks and interfacing with physical hardware for real life demonstration of the system.

5.0 Summary

The aim of this project is to design and implement a Deadline-Driven Scheduler (DDS) using FreeRTOS tasks, queues, and timers.

There are four major components in this system. They are the Deadline-Driven Scheduler, User-Defined Tasks, Deadline-Driven Task Generator, and Monitor Task.

My design in the end matched the expected results and completed the three test benches from the lab manual. Overall the lab project has been a fun and enjoyable experience that supported the class material and assignments, and contributed to my learning and understanding of the course.

6.0 Appendix

Source code

```
// Kutay Cinar
// V00******

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

/* Kernel includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "semphr.h"

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

```
#define PRIORITY_MAX            4
#define PRIORITY_HIGH           2
#define PRIORITY_MIN            1

#define MONITOR                         pdMS_TO_TICKS(12500)

/*-------------------- DD Task Structs ----------------------*/

typedef enum task_type task_type;

enum task_type {
      PERIODIC,
      APERIODIC
};

typedef struct dd_task dd_task;

struct dd_task {
      TaskHandle_t t_handle;
      task_type type;
      uint32_t task_id;
      uint32_t release_time;
      uint32_t absolute_deadline;
      uint32_t completion_time;
      uint32_t period;
};

typedef struct dd_task_list dd_task_list;

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

/*------------------ Message Structs ----------------------*/

typedef enum message_type message_type;

enum message_type {
      create,
      delete,
      active,
      completed,
      overdue,
};
```

```c
typedef struct dd_message dd_message;

struct dd_message {
	message_type type;
	dd_task task;
};

/*---------------------- Queues ----------------------------*/

xQueueHandle xMessageQueue;
xQueueHandle xReplyQueue;
xQueueHandle xListQueue;

/*----------------- Task Prototypes -------------------------*/

TaskHandle_t xDDS;
TaskHandle_t xMonitor;

void DD_Scheduler(void* pvParameters);
void Monitor_Task(void* pvParameters);

/*--------------- Functions Prototypes ---------------------*/

void create_dd_task(TaskHandle_t t_handle,
	task_type type,
	uint32_t task_id,
	uint32_t absolute_deadline
);

void delete_dd_task(uint32_t task_id);

dd_task_list** get_active_dd_task_list(void);
dd_task_list** get_completed_dd_task_list(void);
dd_task_list** get_overdue_dd_task_list(void);

void myDDSInit(void);
void myTestInit(void);

void vCallbackTask1(TimerHandle_t xTimer);
void vCallbackTask2(TimerHandle_t xTimer);
void vCallbackTask3(TimerHandle_t xTimer);
void vCallbackMonitor(TimerHandle_t xTimer);

TimerHandle_t xTimerUser1;
TimerHandle_t xTimerUser2;
```

```
TimerHandle_t xTimerUser3;
TimerHandle_t xTimerMonitor;

/*--------------- Test Tasks Prototypes ---------------------*/

void vUserTask1(void* pvParameters);
void vUserTask2(void* pvParameters);
void vUserTask3(void* pvParameters);

void vTaskGenerator1(void* pvParameters);
void vTaskGenerator2(void* pvParameters);
void vTaskGenerator3(void* pvParameters);

/*---------------- Test Tasks Handles ---------------------*/

TaskHandle_t xUserTask1;
TaskHandle_t xUserTask2;
TaskHandle_t xUserTask3;

TaskHandle_t xTaskGenerator1;
TaskHandle_t xTaskGenerator2;
TaskHandle_t xTaskGenerator3;

/*----------------- Test Task Bench 1 -------------------------*/

#define T1_EXEC                         pdMS_TO_TICKS(95)
#define T1_PERIOD                       pdMS_TO_TICKS(500)

#define T2_EXEC                         pdMS_TO_TICKS(150)
#define T2_PERIOD                       pdMS_TO_TICKS(500)

#define T3_EXEC                         pdMS_TO_TICKS(250)
#define T3_PERIOD                       pdMS_TO_TICKS(750)

int expectedUser1[] = { 95, 595, 1095, -1 };
int expectedUser2[] = { 245, 745, 1245, -1 };
int expectedUser3[] = { 495, 1495, -1 };


/*----------------- Test Task Bench 2 -------------------------*/

#define T1_EXEC                         pdMS_TO_TICKS(95)
#define T1_PERIOD                       pdMS_TO_TICKS(250)

#define T2_EXEC                         pdMS_TO_TICKS(150)
#define T2_PERIOD                       pdMS_TO_TICKS(500)
```

```c
#define T3_EXEC                         pdMS_TO_TICKS(250)
#define T3_PERIOD                       pdMS_TO_TICKS(750)

int expectedUser1[] = {95, 345, 685, 930, 1095, -1 };
int expectedUser2[] = {245, 835, 1245, -1 };
int expectedUser3[] = {590, 1425, -1 };


/*----------------- Test Task Bench 3 -------------------------*/

#define T1_EXEC                         pdMS_TO_TICKS(100)
#define T1_PERIOD                       pdMS_TO_TICKS(500)

#define T2_EXEC                         pdMS_TO_TICKS(200)
#define T2_PERIOD                       pdMS_TO_TICKS(500)

#define T3_EXEC                         pdMS_TO_TICKS(200)
#define T3_PERIOD                       pdMS_TO_TICKS(500)

int expectedUser1[] = { 100, 600, 1100, -1 };
int expectedUser2[] = { 300, 800, 1300, -1 };
int expectedUser3[] = { 500, 1000, 1500, -1 };

/*-------------------------------------------------------------*/

uint32_t taskID1 = 10000;
uint32_t taskID2 = 20000;
uint32_t taskID3 = 30000;

int eventid = 0;

/*-------------------------------------------------------------*/

void main_dds(void)
{
	/* Initialize DD Scheduler and Monitor Task */
	myDDSInit();

	/* Create the tasks from Test Bench (Choose from 1, 2 or 3) */
	myTestInit();

	/* Start the tasks and timer running. */
	vTaskStartScheduler();

	for (;;);
```

```c
}

/*-----------------------------------------------------------*/

void myDDSInit()
{
	// Initialize DD Scheduler Message and Reply Queue
	xMessageQueue = xQueueCreate(100, sizeof(dd_message));
	xReplyQueue = xQueueCreate(100, sizeof(uint32_t));
	xListQueue = xQueueCreate(1, sizeof(dd_task_list**));

	// Create DD Scheduler with max priority
	xTaskCreate(DD_Scheduler, "DDS", configMINIMAL_STACK_SIZE, NULL,
PRIORITY_MAX, &xDDS);

	// Monitor Task and its timer
	xTaskCreate(Monitor_Task, "MON", configMINIMAL_STACK_SIZE, NULL,
PRIORITY_MAX - 1, &xMonitor);
	vTaskSuspend(xMonitor);
	xTimerMonitor = xTimerCreate("TMR_MON", pdMS_TO_TICKS(MONITOR), pdTRUE, 0,
vCallbackMonitor);
	xTimerStart(xTimerMonitor, 0);
}

/*-----------------------------------------------------------*/

void myTestInit(void)
{
	// User Task creation and suspended right after
	xTaskCreate(vUserTask1, "T1", configMINIMAL_STACK_SIZE, NULL, PRIORITY_MIN,
&xUserTask1);
	vTaskSuspend(xUserTask1);
	xTaskCreate(vUserTask2, "T2", configMINIMAL_STACK_SIZE, NULL, PRIORITY_MIN,
&xUserTask2);
	vTaskSuspend(xUserTask2);
	xTaskCreate(vUserTask3, "T3", configMINIMAL_STACK_SIZE, NULL, PRIORITY_MIN,
&xUserTask3);
	vTaskSuspend(xUserTask3);

	// Generator Task creations
	xTaskCreate(vTaskGenerator1, "G1", configMINIMAL_STACK_SIZE, NULL,
PRIORITY_HIGH , &xTaskGenerator1);
	xTaskCreate(vTaskGenerator2, "G2", configMINIMAL_STACK_SIZE, NULL,
PRIORITY_HIGH , &xTaskGenerator2);
	xTaskCreate(vTaskGenerator3, "G3", configMINIMAL_STACK_SIZE, NULL,
PRIORITY_HIGH , &xTaskGenerator3);
```

```
	// Timers
	xTimerUser1 = xTimerCreate("TMR1", T1_PERIOD, pdTRUE, 0, vCallbackTask1);
	xTimerUser2 = xTimerCreate("TMR2", T2_PERIOD, pdTRUE, 0, vCallbackTask2);
	xTimerUser3 = xTimerCreate("TMR3", T3_PERIOD, pdTRUE, 0, vCallbackTask3);

	xTimerStart(xTimerUser1, 0);
	xTimerStart(xTimerUser2, 0);
	xTimerStart(xTimerUser3, 0);
}

/*-----------------------------------------------------------*/

void vCallbackTask1(TimerHandle_t xTimer)
{
	vTaskResume(xTaskGenerator1);
}

void vCallbackTask2(TimerHandle_t xTimer)
{
	vTaskResume(xTaskGenerator2);
}

void vCallbackTask3(TimerHandle_t xTimer)
{
	vTaskResume(xTaskGenerator3);
}

void vCallbackMonitor(TimerHandle_t xTimer)
{
	vTaskResume(xMonitor);
}

/*-----------------------------------------------------------*/

void Monitor_Task(void* pvParameters)
{
	TickType_t currentTick;

	for (;;)
	{
		currentTick = xTaskGetTickCount();
		printf("\n	\t   Monitor Task: \t\t  %d\n", (int)currentTick);
		dd_task_list** activeTasks = get_active_dd_task_list();
		dd_task_list** completedTasks = get_completed_dd_task_list();
		dd_task_list** overdueTask = get_overdue_dd_task_list();
```

```
			vTaskSuspend(NULL);
		}
}

/*-----------------------------------------------------------*/

void vUserTask1(void* pvParameters)
{
		TickType_t currentTick;
		TickType_t prevTick;
		TickType_t executeTick;

		int* pointer = &expectedUser1;

		for (;; )
		{
			currentTick = xTaskGetTickCount();
			executeTick = T1_EXEC;
			// Busy loop
			while (0 < executeTick)
			{
				prevTick = currentTick;
				currentTick = xTaskGetTickCount();
				if (currentTick != prevTick) {
					executeTick--;
				}
			}

			if (*pointer != -1)
			{
				printf("    %d\t   Task 1 complete\t\t  %d\t\t\t  %d\n",
++eventid, (int)currentTick, *pointer);
				pointer++;
			}
			else
				printf("    %d\t   Task 1 complete\t\t  %d\t\t\t  \n",
++eventid, (int)currentTick);

			delete_dd_task(taskID1);
		}
}

/*-----------------------------------------------------------*/

void vUserTask2(void* pvParameters)
{
```

```
	TickType_t currentTick;
	TickType_t prevTick;
	TickType_t executeTick;
	int* pointer = &expectedUser2;

	for (;; )
	{
		currentTick = xTaskGetTickCount();
		executeTick = T2_EXEC;
		// Busy loop
		while (0 < executeTick)
		{
			prevTick = currentTick;
			currentTick = xTaskGetTickCount();
			if (currentTick != prevTick) {
				executeTick--;
			}
		}

		if (*pointer != -1)
		{
			printf("    %d\t   Task 2 complete\t\t  %d\t\t\t  %d\n",
++eventid, (int)currentTick, *pointer);
			pointer++;
		}
		else
			printf("    %d\t   Task 2 complete\t\t  %d\t\t\t  \n",
++eventid, (int)currentTick);

		delete_dd_task(taskID2);
	}
}

/*-----------------------------------------------------------*/

void vUserTask3(void* pvParameters)
{
	TickType_t currentTick;
	TickType_t prevTick;
	TickType_t executeTick;
	int* pointer = &expectedUser3;

	for (;; )
	{
		currentTick = xTaskGetTickCount();
		executeTick = T3_EXEC;
```

```
		// Busy loop
		while (0 < executeTick)
		{
			prevTick = currentTick;
			currentTick = xTaskGetTickCount();
			if (currentTick != prevTick) {
				executeTick--;
			}
		}

		if (*pointer != -1)
		{
			printf("    %d\t   Task 3 complete\t\t  %d\t\t\t  %d\n",
++eventid, (int)currentTick, *pointer);
			pointer++;
		}
		else
			printf("    %d\t   Task 3 complete\t\t  %d\t\t\t  \n",
++eventid, (int)currentTick);

		delete_dd_task(taskID3);
	}
}

/*-----------------------------------------------------------*/

void vTaskGenerator1(void* pvParameters)
{
	TickType_t currentTick;

	for (;; )
	{
		currentTick = xTaskGetTickCount();

		printf("    %d\t   Task 1 released\t\t  %d\t\t\t  %d\n", ++eventid,
(int)currentTick, T1_PERIOD * (taskID1 - 10000));

		if ((int)currentTick >= 1500)
			printf("");

		create_dd_task(xUserTask1, PERIODIC, ++taskID1, T1_PERIOD * (taskID1 -
10000), T1_PERIOD);
	}
}

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

```
void vTaskGenerator2(void* pvParameters)
{
	TickType_t currentTick;

	for (;; )
	{
		currentTick = xTaskGetTickCount();

		printf("    %d\t   Task 2 released\t\t  %d\t\t\t  %d\n", ++eventid,
(int)currentTick, T2_PERIOD * (taskID2 - 20000));

		create_dd_task(xUserTask2, PERIODIC, ++taskID2, T2_PERIOD * (taskID2 -
20000), T2_PERIOD);
	}
}

/*-----------------------------------------------------------*/

void vTaskGenerator3(void* pvParameters)
{
	TickType_t currentTick;

	for (;; )
	{
		currentTick = xTaskGetTickCount();

		printf("    %d\t   Task 3 released\t\t  %d\t\t\t  %d\n", ++eventid,
(int)currentTick, T3_PERIOD * (taskID3 - 30000));

		if ((int)currentTick > 1500)
		{
			printf("");
		}
		create_dd_task(xUserTask3, PERIODIC, ++taskID3, T3_PERIOD * (taskID3 -
30000), T3_PERIOD);
	}
}

/*-----------------------------------------------------------*/

dd_task allocateEmptyTask()
{
	dd_task emptyTask;

	// Apply empty values for struct
```

```
	emptyTask.t_handle = NULL;
	emptyTask.type = PERIODIC;
	emptyTask.task_id = 0;
	emptyTask.absolute_deadline = 0;
	emptyTask.completion_time = 0;
	emptyTask.release_time = 0;

	return emptyTask;
}

void DD_Scheduler(void* pvParameters)
{
	TickType_t currentTick = xTaskGetTickCount();

	dd_task_list* activeTaskList = malloc(sizeof(dd_task_list));
	dd_task_list* completedTaskList = malloc(sizeof(dd_task_list));
	dd_task_list* overdueTaskList = malloc(sizeof(dd_task_list));

	dd_message message_received;
	uint32_t reply;

	activeTaskList->task = allocateEmptyTask();
	completedTaskList->task = allocateEmptyTask();
	overdueTaskList->task = allocateEmptyTask();

	activeTaskList->next_task = NULL;
	completedTaskList->next_task = NULL;
	overdueTaskList->next_task = NULL;

	printf("Event # \t Event \t\t Measured Time (ms) \t Expected Time (ms)\n");

printf("================================================================================
\n");

	for (;; )
	{
		currentTick = xTaskGetTickCount();
		// Check for message
		if (xQueueReceive(xMessageQueue, &message_received, portMAX_DELAY))
		{
			dd_task_list* curr, * prev;
			int flag = 1;

			currentTick = xTaskGetTickCount();

			// Stop currently running task
```

```
                    if (activeTaskList->task.t_handle != NULL) {
                          vTaskSuspend(activeTaskList->task.t_handle);
                          vTaskPrioritySet(activeTaskList->task.t_handle,
PRIORITY_MIN);
                    }

                    switch (message_received.type)
                    {
                    case create:
                    {
                          dd_task_list* node = malloc(sizeof(dd_task_list));

                          node->task = message_received.task;
                          node->next_task = NULL;

                          // Check if schedulable
                          if (node->task.absolute_deadline > currentTick +
node->task.period)
                          {
                                curr = overdueTaskList;
                                prev = overdueTaskList;

                                dd_task_list* node = malloc(sizeof(dd_task_list));
                                node->task = activeTaskList->task;
                                node->next_task = NULL;

                                while (curr->next_task != NULL)
                                {
                                      prev = curr;
                                      curr = curr->next_task;
                                }

                                // Add to list at the end
                                curr->next_task = node;
                          }

                          else
                          {
                                curr = activeTaskList;
                                prev = activeTaskList;

                                // TRAVERSE LIST
                                while (curr->next_task != NULL)
                                {
                                      if (node->task.absolute_deadline <
curr->task.absolute_deadline)
```

```
                                        {
                                                node->next_task = curr;
                                                if (prev == curr)
                                                {
                                                        activeTaskList = node;
                                                }
                                                else
                                                {
                                                        prev->next_task = node;
                                                }
                                                flag = 0;
                                                break;
                                        }
                                        prev = curr;
                                        curr = curr->next_task;
                                }

                                if (flag)
                                {
                                        // EMPTY LIST
                                        if (curr->task.absolute_deadline == 0)
                                                activeTaskList = node;

                                        // ADD ELEMENT TO LEFT
                                        else if (node->task.absolute_deadline <
curr->task.absolute_deadline)
                                        {
                                                // ONLY 1
                                                if (prev == curr)
                                                {
                                                        node->next_task = curr;
                                                        activeTaskList = node;
                                                }
                                                // MANY
                                                else
                                                {
                                                        node->next_task = curr;
                                                        prev->next_task = node;
                                                }
                                        }
                                        // ADD ELEMENT TO RIGHT
                                        else
                                                curr->next_task = node;
                                }
                        }
```

```
                        xQueueSend(xReplyQueue, &reply, portMAX_DELAY);
                        break;
                }

                case delete:
                {
                        activeTaskList->task.completion_time = currentTick;

                        // move to other list
                        if (activeTaskList->task.completion_time <
activeTaskList->task.absolute_deadline)
                        {
                                curr = completedTaskList;
                                prev = completedTaskList;

                                dd_task_list* node = malloc(sizeof(dd_task_list));
                                node->task = activeTaskList->task;
                                node->next_task = NULL;

                                // Traverse list
                                while (curr->next_task != NULL)
                                {
                                        prev = curr;
                                        curr = curr->next_task;
                                }

                                // Add to list at the end
                                curr->next_task = node;
                        }

                        else
                        {
                                curr = overdueTaskList;
                                prev = overdueTaskList;

                                dd_task_list* node = malloc(sizeof(dd_task_list));
                                node->task = activeTaskList->task;
                                node->next_task = NULL;

                                // Traverse list
                                while (curr->next_task != NULL)
                                {
                                        prev = curr;
                                        curr = curr->next_task;
                                }
```

```
                                // Add to list at the end
                                curr->next_task = node;
                        }

                        // DELETE FROM ACTIVE LIST
                        curr = activeTaskList;
                        prev = activeTaskList;

                        // ActiveTaskList 1 element
                        if (curr->next_task == NULL)
                        {
                                activeTaskList->task = allocateEmptyTask();
                        }
                        else
                        {
                                activeTaskList->task = curr->next_task->task;
                                activeTaskList->next_task =
curr->next_task->next_task;
                        }
                        xQueueSend(xReplyQueue, &reply, portMAX_DELAY);
                        break;
                }

                case active:

                        xQueueSend(xListQueue, &activeTaskList, portMAX_DELAY);
                        vTaskResume(xMonitor);
                        break;

                case completed:

                        xQueueSend(xListQueue, &completedTaskList,
portMAX_DELAY);
                        vTaskResume(xMonitor);
                        break;

                case overdue:

                        xQueueSend(xListQueue, &overdueTaskList, portMAX_DELAY);
                        vTaskResume(xMonitor);
                        break;

                default:
                        break;

                }
```

```
                    if (activeTaskList->task.absolute_deadline > 0)
                    {
                          if (activeTaskList->task.release_time == 0)
                                activeTaskList->task.release_time = currentTick;

                          vTaskPrioritySet(activeTaskList->task.t_handle,
PRIORITY_HIGH);

                          vTaskResume(activeTaskList->task.t_handle);
                    }

             }
      }
}

void create_dd_task(TaskHandle_t t_handle, task_type type, uint32_t task_id,
uint32_t absolute_deadline, uint32_t period)
{
      int reply;

      // create the dd_task struct
      dd_task task = {
            t_handle,
            type,
            task_id,
            0, // not yet released
            absolute_deadline,
            0, // not yet completed
            period
      };

      // create a task message to be created
      dd_message message = { create, task };

      // send task message to DD queue
      xQueueSend(xMessageQueue, &message, portMAX_DELAY);

      // wait for reply back from DD scheduler
      xQueueReceive(xReplyQueue, &reply, portMAX_DELAY);

      vTaskSuspend(NULL);

}

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

```
void delete_dd_task(uint32_t task_id)
{
	int reply;

	// create the dd_task struct
	dd_task task = { .task_id = task_id };

	// create a task message to be created
	dd_message message = { delete, task };

	// send task message to DD queue
	xQueueSend(xMessageQueue, &message, portMAX_DELAY);

	// wait for reply back from DD scheduler
	xQueueReceive(xReplyQueue, &reply, portMAX_DELAY);

}

/*-----------------------------------------------------------*/

dd_task_list** get_active_dd_task_list()
{
	dd_task_list* activelist;
	dd_message message = { .type = active };

	// Send message to DD scheduler to get correct list
	xQueueSend(xMessageQueue, &message, portMAX_DELAY);
	xQueueReceive(xListQueue, &activelist, portMAX_DELAY);

	dd_task_list* curr = activelist;
	dd_task_list* prev = activelist;

	int count = 0;

	// Print list
	printf("	\t   Active: \t");
	while (curr->next_task != NULL)
	{
		count++;
		prev = curr;
		curr = curr->next_task;
	}

	if (curr->task.task_id != 0)
	{
```

```
                count++;
        }

        printf("%d\n", count);

        return activelist;
}

/*-----------------------------------------------------------*/

dd_task_list** get_completed_dd_task_list()
{
        dd_task_list* completedList;
        dd_message message = { .type = completed };

        // Send message to DD scheduler to get correct list
        xQueueSend(xMessageQueue, &message, portMAX_DELAY);
        xQueueReceive(xListQueue, &completedList, portMAX_DELAY);

        int count = 0;
        int flag = 1;

        dd_task_list* curr = completedList;
        dd_task_list* prev = completedList;

        // Print list
        printf("      \t   Completed: \t");
        while (curr->next_task != NULL)
        {
                if (flag)
                {
                        flag = 0;
                }
                else
                {
                        count++;
                }
                prev = curr;
                curr = curr->next_task;
        }

        if (curr->task.task_id != 0)
        {
                count++;
        }
```

```
	printf("%d\n", count);

	return completedList;
}

/*-----------------------------------------------------------*/

dd_task_list** get_overdue_dd_task_list()
{
	dd_task_list* overdueList;
	dd_message message = { .type = overdue };

	// Send message to DD scheduler to get correct list
	xQueueSend(xMessageQueue, &message, portMAX_DELAY);
	xQueueReceive(xListQueue, &overdueList, portMAX_DELAY);

	int count = 0;
	int flag = 1;

	dd_task_list* curr = overdueList;
	dd_task_list* prev = overdueList;

	// Print list
	printf("	\t   Overdue: \t");
	while (curr->next_task != NULL)
	{
		if (flag)
		{
			flag = 0;
		}
		else
		{
			count++;
		}
		prev = curr;
		curr = curr->next_task;
	}

	if (curr->task.task_id != 0)
	{
		count++;
	}

	printf("%d\n\n", count);

	return overdueList;
```

```
}

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

References

[1] Department of Electrical and Computer Engineering, "Lab Manual ECE 455: Real Time Computer Systems Design Project", University of Victoria, Victoria, 2019.

[2] "ECE455 - Lab Project 2 Slides", University of Victoria, Khaled Kelany, 2021.