University of Victoria
Department of Electrical and Computer Engineering
ECE 455 Spring 2023
B01

Lab Project
Project 2: Deadline Driven Scheduler

Merlin M'Cloud - V0094624
Jordan Atchison - V00915743
April 4, 2023

Introduction

The goal of this project is to implement a deadline driven scheduler (DDS) using FreeRTOS. The deadline driven scheduler uses earliest deadline first scheduling (EDF) to dynamically change task priorities based on which has the soonest absolute deadline. A task with the soonest deadline will be given a priority of 'high' and all other tasks are given priorities of 'low'. These tasks are referred to within this project as deadline-driven tasks (DD-tasks).

FreeRTOS does not support deadline driven scheduling; therefore, the algorithm must be built on top of the existing FreeRTOS scheduler. Four FreeRTOS tasks (F-tasks) are used to implement and test the DDS: deadline-driven scheduler task, user-defined tasks, deadline-driven task generator, and the monitor task. Here is a list of F-tasks in order of priority:

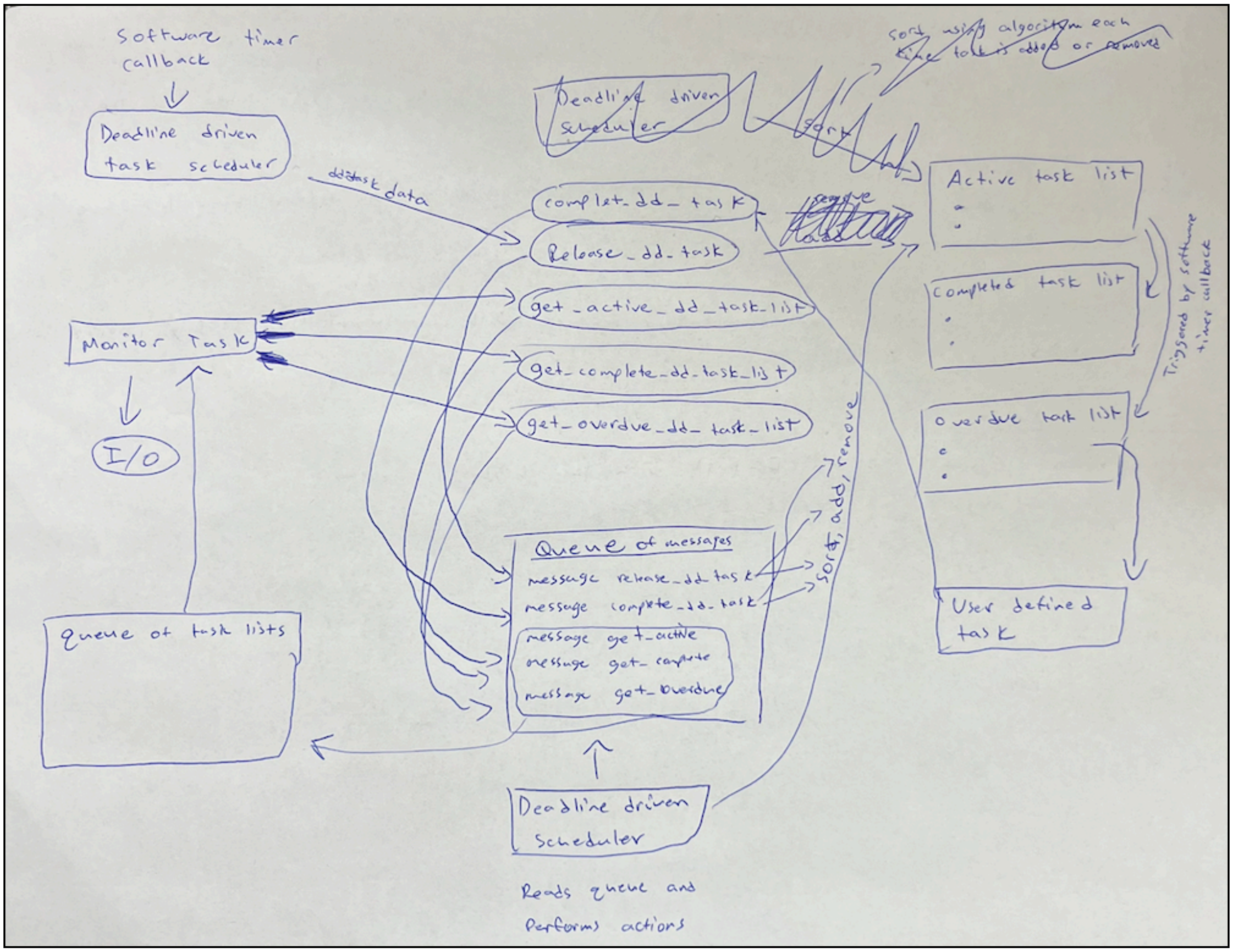
- The scheduler task has the highest priority and reads from a queue of messages sent from the other three F-tasks. It is responsible for shifting the priorities of the DD-tasks based on their deadlines, creating new DD-tasks, and removing them from the active list once they finish or become overdue. This has the highest task priority.
- The monitor task periodically reports information on active tasks, completed tasks, and overdue tasks. This has the second highest task priority.
- The task generator periodically creates new DD-tasks using a software timer. The timer will interrupt the current task.
- User-defined tasks act as a shell for DD-tasks, allowing them to be actively run. These have the lowest priority.

Design Solution

The scheduler makes use of a message queue read by the deadline-driven scheduler task (F-task). Different message types are sent to the queue by the other F-tasks and invoke processes within the scheduler. Our design process involved separating the functionality into their corresponding message types and implementing each scheduler functionality one by one.

Design Document

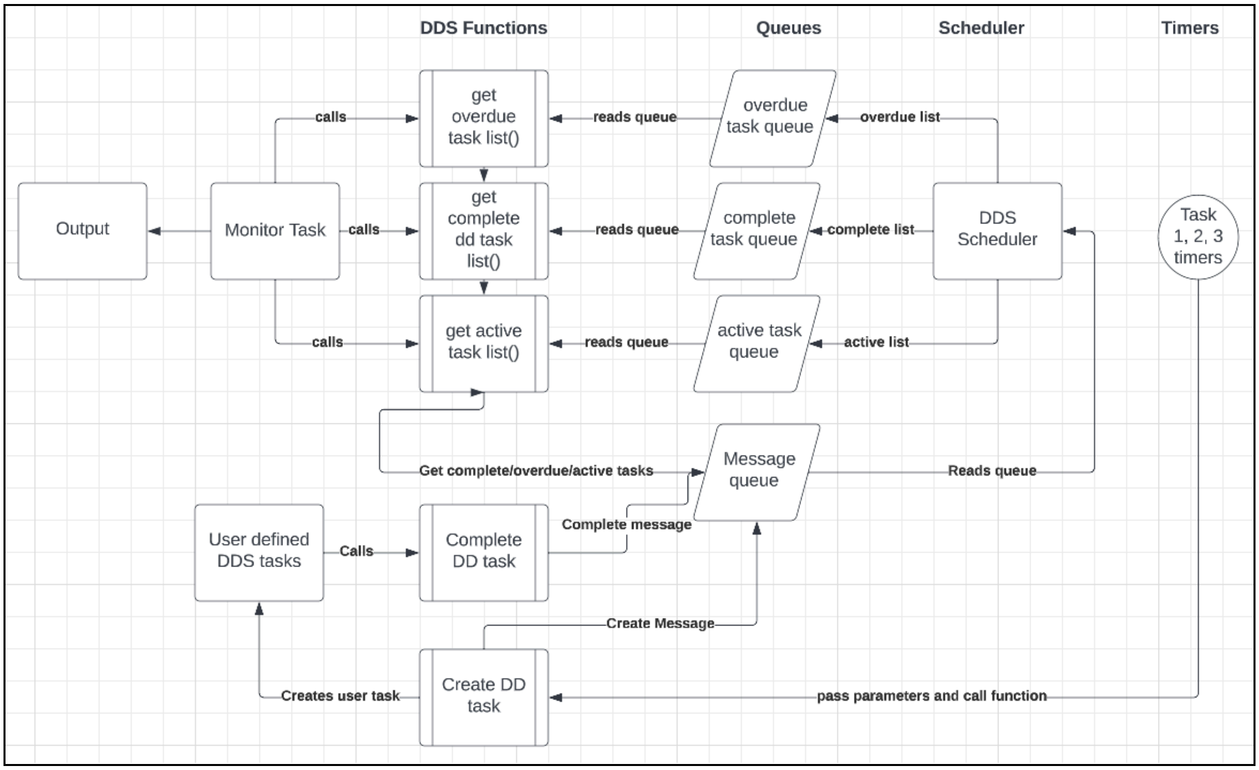
This is the initial design document that we created. It is a high level illustration of how data moves through the scheduler and the dependencies between scheduler functionality.



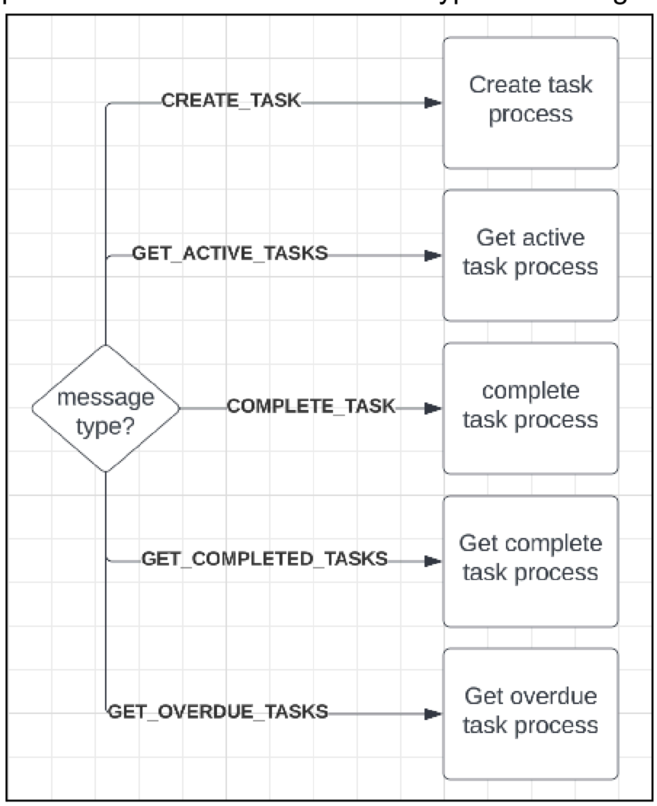
Overall, much of the design document is the same as the final design. The only difference is how we handled creating tasks. We didn't use an F-task for the functionality of creating a task, instead we used a software timer callback so it would be more accurate.

System Overview

Below is a system overview that outlines the interactions between F-tasks, queues, timers, and DDS functions.



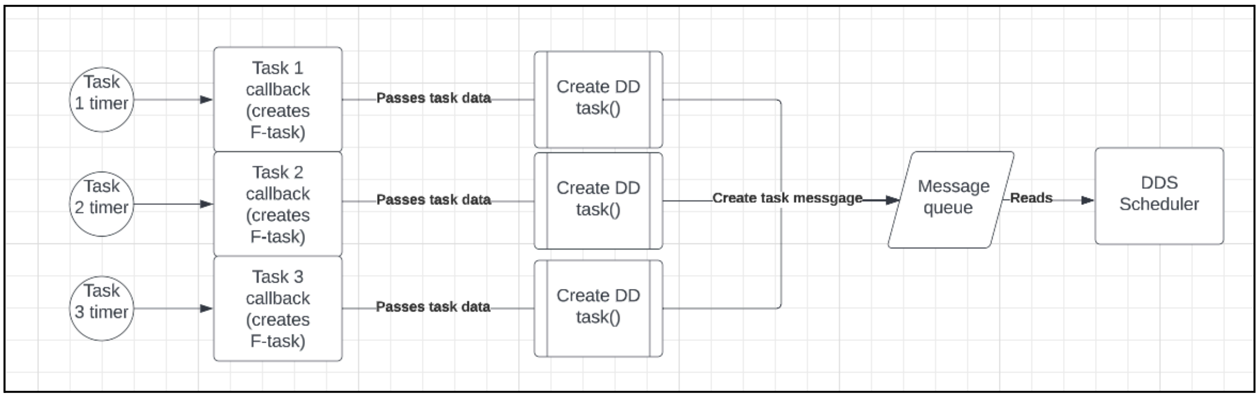
The scheduler itself at a high level is pretty simple. It reads messages from a queue and performs an action based on the type of message. This is shown in the following flow chart.



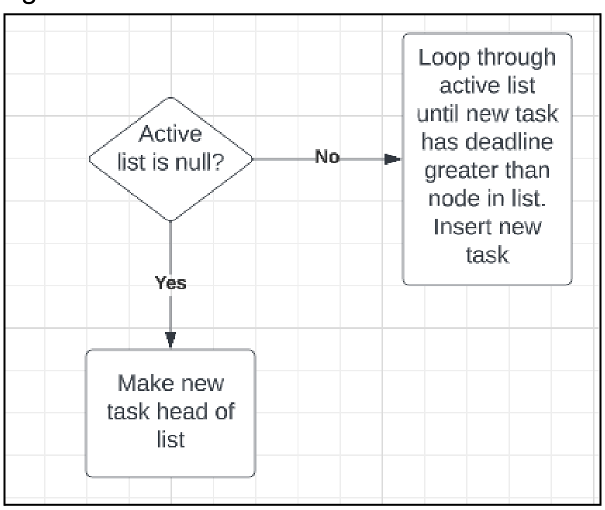
In the following sections, each of the five DDS functions are explained.

Create Task

A software timer periodically creates a new instance of a task. It then calls create_dd_task() and sends it the information about the newly created task. create_dd_task() bundles this information into a dd_task struct and sends it to the scheduler via the messenger queue. The scheduler then adds the new task to active_task_list in the correct order. Below is a diagram of how this works.



As part of creating tasks, our Scheduling task sorts the new task into the correct position. This is the only place where we need to sort the tasks. Below is a simple diagram of the sorting algorithm.



Complete task

Each DD-task internally keeps track of how long it has been executing. Since a DD-task is only able to communicate with the schedulers auxiliary functions, when the full execution time has occurred, it will call the scheduler function complete_dd_task. This function will then send a message to the scheduler through the messenger queue. Once the scheduler receives this message, it will remove the task from the active_task_list and add it to the completed_task_list to reflect the newly completed task. It will then use vTaskDelete() to delete the F-task.

Get Active Tasks

The monitor task periodically calls get_active_tasks(). This function then sends a message requesting the head of the active_task_list to the scheduler through the messenger queue. Once the scheduler receives this message, it will send the head of the active_task_list to the active_task_queue, which will then be received by get_active_tasks(). Then, the function will return the head of the list to the monitor task.

Get Completed Tasks

The monitor task periodically calls get_completed_tasks(). This function then sends a message requesting the head of the completed_task_list to the scheduler through the messenger queue. Once the scheduler receives this message, it will send the head of the completed_task_list to the completed_task_queue, which will then be received by get_completed_tasks(). Then, the function will return the head of the list to the monitor task.

Get Overdue Tasks

The monitor task periodically calls get_overdue_tasks(). This function then sends a message requesting the head of the overdue_task_list to the scheduler through the messenger queue. Once the scheduler receives this message, it will send the head of the overdue_task_list to the overdue_task_queue, which will then be received by get_overdue_tasks(). Then, the function will return the head of the list to the monitor task.

Discussion

Overall, our scheduler matched the expected performance with an slight error of a few ms. This is likely due to the additional overhead of the monitor task. Below are the test bench results.

Test bench 1:

Event #	**Event**	**Measured time**	**Expected time**
1	Task 1 released	0	0
2	Task 2 released	0	0
3	Task 3 released	0	0
4	Task 1 completed	98	95
5	Task 2 completed	247	245
6	Task 3 completed	496	495

7	Task 1 released	500	500
8	Task 2 released	500	500
9	Task 1 completed	598	595
10	Task 2 completed	747	745
11	Task 3 released	750	750
12	Task 1 released	1000	1000
13	Task 2 released	1000	1000
14	Task 3 completed	1002	1000
15	Task 1 completed	1097	1095
16	Task 2 completed	1246	1245
17	Task 1 released	1500	1500
18	Task 2 released	1500	1500
19	Task 3 released	1500	1500

Test bench 2:

Event #	**Event**	**Measured time**	**Expected time**
1	Task 1 released	0	0
2	Task 2 released	0	0
3	Task 3 released	0	0
4	Task 1 completed	98	95
5	Task 2 completed	247	245
6	Task 1 released	250	250
7	Task 1 completed	343	340
8	Task 2 released	500	500
9	Task 1 released	500	500
10	Task 3 completed	594	590
11	Task 1 complete	691	685

12	Task 3 released	750	750
13	Task 1 released	750	750
14	Task 2 completed	840	835
15	Task 1 completed	934	930
16	Task 2 released	1000	1000
17	Task 1 released	1000	1000
18	Task 1 completed	1098	1095
19	Task 1 released	1250	1250
20	Task 3 complete	1284	1275
21	Task 3 released	1500	1500
22	Task 2 released	1500	1500

Test bench 3:

Event #	**Event**	**Measured time**	**Expected time**
1	Task 1 released	0	0
2	Task 2 released	0	0
3	Task 3 released	0	0
4	Task 1 completed	103	100
5	Task 3 completed	303	300
6	Task 1 released	500	500
7	Task 2 released	500	500
8	Task 3 released	500	500
9	Task 2 overdue	501	501
10	Task 1 complete	603	600
11	Task 2 completed	805	800
12	Task 1 released	1000	1000
13	Task 2 released	1000	1000

14	Task 3 released	1000	1000
15	Task 3 overdue	1001	1001
16	Task 1 completed	1108	1100
17	Task 2 completed	1308	1300
18	Task 1 released	1500	1500
19	Task 2 released	1500	1500
20	Task 3 released	1500	1500
21	Task 3 overdue	1501	1501

Limitations and Possible Improvements

Looking back at this project, we could have made the development process easier by gaining a better understanding of the theory and documentation behind FreeRTOS before starting development. Although we did create a design document at the start of the project, we had a poor understanding of how some FreeRTOS components worked, which made it difficult to plan properly. We did eventually learn all of the necessary documentation as we worked our way through the process, but it would have been easier to do this at the beginning.

Summary

The purpose of this project was to manually create a task scheduling system using FreeRTOS tasks, timers, and queues. The contents of this project were similar to the contents of what we learned in the lecture portion of the class. This made both sections easier, as we learned necessary information in the lecture section, and then applied the information to this lab project. We had some initial setbacks involving memory management. It was difficult to properly manage pointers as they were constantly getting passed to different functions through queues. This just took some time to figure out, and we were able to make everything functional through basic debugging techniques.

We were able to complete the project with some extra time left, which allowed us to fine tune our task scheduling system, compare our results to the theoretical results, and ensure everything was working correctly.

Appendix (with source code)

```
/* Standard includes. */
#include <stdint.h>
#include <stdio.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"
#include "../FreeRTOS_Source/portable/MemMang/heap_4.c"


#define QUEUE_LENGTH 100
#define CREATE_TASK 0
#define COMPLETE_TASK 1
#define GET_ACTIVE_TASKS 2
#define GET_COMPLETED_TASKS 3
#define GET_OVERDUE_TASKS 4

#define TASK_1_PERIOD 500
#define TASK_2_PERIOD 500
#define TASK_3_PERIOD 500

#define TASK_1_EXEC_TIME 100
#define TASK_2_EXEC_TIME 200
#define TASK_3_EXEC_TIME 200

#define SCHEDULER_PRIORITY (configMAX_PRIORITIES)
#define MONITOR_PRIORITY (configMAX_PRIORITIES - 1)
#define DEFAULT_PRIORITY 1


#define PRINT_TASKS 0

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

```c
static void prvSetupHardware( void );


static void callback_Task_Generator_1();
static void callback_Task_Generator_2();
static void callback_Task_Generator_3();
static void DD_Task_Scheduler( void *pvParameters );
static void DD_Task_Monitor( void *pvParameters );
static void Task1( void *pvParameters );
static void Task2( void *pvParameters );
static void Task3( void *pvParameters );


xQueueHandle message_queue = 0;
xQueueHandle task_return_queue = 0;
xQueueHandle active_tasks_queue = 0;
xQueueHandle completed_tasks_queue = 0;
xQueueHandle overdue_tasks_queue = 0;


xTimerHandle task1_timer;
xTimerHandle task2_timer;
xTimerHandle task3_timer;

typedef enum {PERIODIC,APERIODIC} task_type;

typedef 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;
} dd_task;

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

typedef struct Message {
   uint32_t message_type;
```

```
   dd_task task;
   uint32_t task_id;
} Message;

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); // removed ** before dd_task_list. unsure
if correct
dd_task_list* get_complete_dd_task_list(void);
dd_task_list* get_overdue_dd_task_list(void);
void print_task(dd_task_list* task);

void Delay( void );

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

int main(void)
{
   prvSetupHardware();

   message_queue = xQueueCreate(QUEUE_LENGTH, sizeof(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));
   task_return_queue = xQueueCreate(QUEUE_LENGTH, sizeof(dd_task));

   task1_timer = xTimerCreate("timer1", pdMS_TO_TICKS(TASK_1_PERIOD), pdFALSE,
(void*)0, callback_Task_Generator_1);
   task2_timer = xTimerCreate("timer2", pdMS_TO_TICKS(TASK_2_PERIOD), pdFALSE,
(void*)0, callback_Task_Generator_2);
   task3_timer = xTimerCreate("timer3", pdMS_TO_TICKS(TASK_3_PERIOD), pdFALSE,
(void*)0, callback_Task_Generator_3);

   xTaskCreate( DD_Task_Scheduler, "Scheduler", configMINIMAL_STACK_SIZE, NULL,
SCHEDULER_PRIORITY, NULL);
   xTaskCreate( DD_Task_Monitor, "Monitor", configMINIMAL_STACK_SIZE, NULL,
MONITOR_PRIORITY, NULL);

   xTimerStart(task1_timer, 0);
   xTimerStart(task2_timer, 0);
   xTimerStart(task3_timer, 0);
```

```
   /* Start the tasks */
   vTaskStartScheduler();

   return 0;
}

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

void print_task(dd_task_list* dd_task){
   if(dd_task->task.completion_time == -1){
       printf("Task ID: %d, Release time: %d, Absolute deadline: %d\n",
               dd_task->task.task_id,
               dd_task->task.release_time,
               dd_task->task.absolute_deadline);
   }
   else{
       printf("Task ID: %d, Release time: %d, Absolute deadline: %d, Completion time:
%d\n",
               dd_task->task.task_id,
               dd_task->task.release_time,
               dd_task->task.absolute_deadline,
               dd_task->task.completion_time);
   }
}

/////////////////////////////////////////////////////////////////////////////////////
///
/* Scheduler internal functions */
/* These functions send messages to the queue that are read by the scheduler*/
/////////////////////////////////////////////////////////////////////////////////////
///

void create_dd_task(TaskHandle_t t_handle, task_type type, uint32_t task_id, uint32_t
absolute_deadline){
   vTaskSuspend(t_handle);
   Message message;
   dd_task task;
   task.absolute_deadline = absolute_deadline;
   task.completion_time = -1;
   task.task_id = task_id;
   task.type = type;
```

```
    task.t_handle = t_handle;

    message.message_type = CREATE_TASK;
    message.task = task;

    if(xQueueSend(message_queue, &message, 0)){ //send message to queue
        //printf("message sent.\n");
    }
}

void complete_dd_task(uint32_t task_id){
    dd_task task;

    Message message;
    message.message_type = COMPLETE_TASK;
    message.task_id = task_id;
    if(xQueueSend(message_queue, &message, 500)){
        //printf("message sent.\n");
    }

    if(xQueueReceive(task_return_queue, &task, portMAX_DELAY)){
        vTaskDelete(task.t_handle);
    }
}

dd_task_list* get_active_dd_task_list(void){
    Message message;
    message.message_type = GET_ACTIVE_TASKS;
    if(xQueueSend(message_queue, &message, 500)){
        //printf("message sent.\n");
    }
    dd_task_list* head = (dd_task_list*)pvPortMalloc(sizeof(dd_task_list));
    while(1){
        if(xQueueReceive(active_tasks_queue, &head, 500)){
            return head;
        }
    }
}

dd_task_list* get_complete_dd_task_list(void){
    Message message;
    message.message_type = GET_COMPLETED_TASKS;
```

```c
    if(xQueueSend(message_queue, &message, 500)){
        //printf("message sent.\n");
    }

    dd_task_list* head = (dd_task_list*)pvPortMalloc(sizeof(dd_task_list));
    while(1){
        if(xQueueReceive(completed_tasks_queue, &head, 500)){
            return head;
        }
    }
}

dd_task_list* get_overdue_dd_task_list(void){
    Message message;
    message.message_type = GET_OVERDUE_TASKS;
    if(xQueueSend(message_queue, &message, 500)){
        //printf("message sent.\n");
    }

    dd_task_list* head = (dd_task_list*)pvPortMalloc(sizeof(dd_task_list));
    while(1){
        if(xQueueReceive(overdue_tasks_queue, &head, 500)){
            return head;
        }
    }
}


/////////////////////////////////////////////////////////////////////////////////////////
///
/* F tasks */
/////////////////////////////////////////////////////////////////////////////////////////
///

//User defined DD task
static void Task1(void *pvParameters){
    TickType_t current_t = xTaskGetTickCount();
    TickType_t prev_t = 0;
    TickType_t exec_time = TASK_1_EXEC_TIME/portTICK_PERIOD_MS; //TASK_1_EXEC_TIME is
in ms
    while(exec_time > 0){
        current_t = xTaskGetTickCount();
```

```
        if(current_t != prev_t){
            //if tick has updated, update exec time
            prev_t = current_t;
            exec_time--;
        }
    }
    complete_dd_task(1);

    while(1){
        vTaskDelay(500);
    }
}

static void Task2(void *pvParameters){
    TickType_t current_t = xTaskGetTickCount();
    TickType_t prev_t = 0;
    TickType_t exec_time = TASK_2_EXEC_TIME/portTICK_PERIOD_MS; //TASK_1_EXEC_TIME is
in ms
    while(exec_time > 0){
        current_t = xTaskGetTickCount();
        if(current_t != prev_t){
            //if tick has updated, update exec time
            prev_t = current_t;
            exec_time--;
        }
    }
    complete_dd_task(2);

    while(1){
        vTaskDelay(500);
    }
}

static void Task3(void *pvParameters){
    TickType_t current_t = xTaskGetTickCount();
    TickType_t prev_t = 0;
    TickType_t exec_time = TASK_3_EXEC_TIME/portTICK_PERIOD_MS; //TASK_1_EXEC_TIME is
in ms
    while(exec_time > 0){
        current_t = xTaskGetTickCount();
        if(current_t != prev_t){
            //if tick has updated, update exec time
```

```c
            prev_t = current_t;
            exec_time--;
        }
    }
    complete_dd_task(3);

    while(1){
        vTaskDelay(500);
    }
}

//monitor task
static void DD_Task_Monitor( void *pvParameters ){
    vTaskDelay(50);
    while(1){
        int num_active = 0;
        int num_complete = 0;
        int num_overdue = 0;

        dd_task_list* current = (dd_task_list*)pvPortMalloc(sizeof(dd_task_list));
        TickType_t current_t = xTaskGetTickCount();

        printf("\n------------MONITOR TASK------------");
        printf("\ncurrent time: %d \n\n", current_t);

        //active tasks
        current = get_active_dd_task_list();
        printf("ACTIVE TASKS:\n");
        while(current != NULL){
            if(PRINT_TASKS){
                print_task(current);
            }
            num_active++;
            current = current->next_task;
        }

        printf("Number of active tasks: %d\n\n", num_active);

        //completed tasks
        current = get_complete_dd_task_list();
        printf("COMPLETED TASKS:\n");
```

```c
        while(current != NULL){
            if(PRINT_TASKS){
                print_task(current);
            }
            num_complete++;
            current = current->next_task;
        }

        printf("Number of completed tasks: %d\n\n", num_complete);

        //overdue tasks
        current = get_overdue_dd_task_list();
        printf("OVERDUE TASKS:\n");
        //running one too many times, printing garbage
        while(current != NULL){
            if(PRINT_TASKS){
                print_task(current);
            }
            num_overdue++;
            current = current->next_task;
        }
        printf("Number of overdue tasks: %d\n\n", num_overdue);

        vTaskDelay(500);
    }
}

//The actual scheduler
static void DD_Task_Scheduler(void *pvParameters){
   Message message;
   dd_task_list* active_tasks_head = NULL;
   dd_task_list* completed_tasks_head = NULL;
   dd_task_list* overdue_tasks_head = NULL;

   dd_task_list* current;
   dd_task_list* current2;
   dd_task_list* prev;
   uint32_t active_size = 0;
   uint32_t completed_size = 0;
   uint32_t overdue_size = 0;

   while(1){
```

```
        if(xQueueReceive(message_queue, &message, 500) == pdFALSE){
            continue;
        }
        //check for overdue tasks
        TickType_t time_current = xTaskGetTickCount();
        current = active_tasks_head;
        prev = active_tasks_head;

        if(current != NULL){
            if(time_current > current->task.absolute_deadline){
                //TASK IS OVERDUE
                vTaskDelete(current->task.t_handle);
                dd_task_list* new_overdue_task =
(dd_task_list*)pvPortMalloc(sizeof(dd_task_list));

                new_overdue_task->task = current->task;

                if(prev == current){
                    //current is the head of the list
                    active_tasks_head = current->next_task;
                    //free(current);
                }
                else if(current->next_task == NULL){
                    //current is last in list
                    current = NULL;
                    //free(current);
                }else{
                    //current is not last in list
                    prev->next_task = current->next_task;
                    //free(current);
                }

                //if there is a task to work on, start it
                if(active_tasks_head != NULL){
                    vTaskResume(active_tasks_head->task.t_handle);
                }



                //adding task to overdue task list
                new_overdue_task->next_task = NULL;
                current = overdue_tasks_head;
```

```
               if(overdue_tasks_head == NULL){
                   overdue_tasks_head = new_overdue_task;
               }else{
                   while(current->next_task != NULL){
                       current = current->next_task;
                   }
                   current->next_task = new_overdue_task;
               }

               active_size--;
               overdue_size++;
           }
       }

       switch(message.message_type){

       case CREATE_TASK:
           current = active_tasks_head;
           dd_task_list* new_task = (dd_task_list*)pvPortMalloc(sizeof(dd_task_list));
           //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_tasks_head == NULL){
               active_tasks_head = (dd_task_list*)pvPortMalloc(sizeof(dd_task_list));
               active_tasks_head->task = new_task->task;
               active_tasks_head->next_task = NULL;

               //else, list is not empty
           }else{
               //if task to be inserted has lower deadline than head of list, insert
at the front

               if(message.task.absolute_deadline < current->task.absolute_deadline){
                   new_task->next_task = current;
                   active_tasks_head = 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
                while(current->next_task != NULL && (message.task.absolute_deadline
> current->next_task->task.absolute_deadline)){ //A next_task exists
                    current = current->next_task;
                }
                //insert node in correct location
                new_task->next_task = current->next_task;
                current->next_task = new_task;
            }
        }

        active_size++;

        //if there is a task to work on, start it
        if(active_tasks_head != NULL){
            vTaskResume(active_tasks_head->task.t_handle);
        }
        break;

    case COMPLETE_TASK:
        //removing task from active task list
        current = active_tasks_head;
        prev = active_tasks_head;
        dd_task_list* new_completed_task =
(dd_task_list*)pvPortMalloc(sizeof(dd_task_list));
        while(current != NULL){
            if(message.task_id == current->task.task_id){
                new_completed_task->task = current->task;
                new_completed_task->task.completion_time = xTaskGetTickCount();
                //remove from active list
                if(prev == current){
                    //current is the head of the list
                    active_tasks_head = current->next_task;
                }
                else if(current->next_task == NULL){
                    //current is last in list
                    current = NULL;
                }else{
                    //current is not last in list
```

```
                    prev->next_task = current->next_task;
                }
                break;
            }
            prev = current;
            current = current->next_task;
        }

        //if there is a task to work on, start it
        if(active_tasks_head != NULL){
            vTaskResume(active_tasks_head->task.t_handle);
        }

        if(xQueueSend(task_return_queue, &new_completed_task->task, 500)){
            //printf("task to be deleted sent to queue.\n");
        }

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

        active_size--;
        completed_size++;

        break;
    case GET_ACTIVE_TASKS:
        if(xQueueSend(active_tasks_queue, &active_tasks_head, 500)){
            //printf("active list sent to queue.\n");
        }else{
            printf("Failure\n");
        }
        break;
    case GET_COMPLETED_TASKS:
        if(xQueueSend(completed_tasks_queue, &completed_tasks_head, 500)){
```

```c
                //printf("active list sent to queue.\n");
            }
            break;
        case GET_OVERDUE_TASKS:
            if(xQueueSend(overdue_tasks_queue, &overdue_tasks_head, 500)){
                //printf("active list sent to queue.\n");
            }
            break;
        }
    }
}

//Periodically generates new DD tasks
//Calls internal create_dd_task function
static void callback_Task_Generator_1(){
   uint32_t absolute_deadline = xTaskGetTickCount() + TASK_1_PERIOD; //from test bench
(period)
   TaskHandle_t t_handle;
   uint32_t task_id = 1;
   task_type type = PERIODIC;
   xTaskCreate(Task1, "task1", configMINIMAL_STACK_SIZE, NULL, DEFAULT_PRIORITY,
&t_handle); //create task

   vTaskSuspend(t_handle);

   create_dd_task(t_handle, type, task_id, absolute_deadline);
   xTimerStart(task1_timer, 0);
}

static void callback_Task_Generator_2(){
   uint32_t absolute_deadline = xTaskGetTickCount() + TASK_2_PERIOD; //from test bench
(period)
   TaskHandle_t t_handle;
   uint32_t task_id = 2;
   task_type type = PERIODIC;
   xTaskCreate(Task2, "task2", configMINIMAL_STACK_SIZE, NULL, DEFAULT_PRIORITY,
&t_handle); //create task

   vTaskSuspend(t_handle);

   create_dd_task(t_handle, type, task_id, absolute_deadline);
   xTimerStart(task2_timer, 0);
```

```c
}

static void callback_Task_Generator_3(){
   uint32_t absolute_deadline = xTaskGetTickCount() + TASK_3_PERIOD; //from test bench
(period)
   TaskHandle_t t_handle;
   uint32_t task_id = 3;
   task_type type = PERIODIC;
   xTaskCreate(Task3, "task3", configMINIMAL_STACK_SIZE, NULL, DEFAULT_PRIORITY,
&t_handle); //create task

   vTaskSuspend(t_handle);

   create_dd_task(t_handle, type, task_id, absolute_deadline);
   xTimerStart(task3_timer, 0);
}
/*-----------------------------------------------------------*/

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. */
```

```c
    for( ;; );
}
/*-----------------------------------------------------------*/

void vApplicationIdleHook( void )
{
volatile size_t xFreeStackSpace;

    /* The idle task hook is enabled by setting configUSE_IDLE_HOOK to 1 in
    FreeRTOSConfig.h.

    This function is called on each cycle of the idle task.  In this case it
    does nothing useful, other than report the amount of FreeRTOS heap that
    remains unallocated. */
    xFreeStackSpace = xPortGetFreeHeapSize();

    if( xFreeStackSpace > 100 )
    {
        /* By now, the kernel has allocated everything it is going to, so
        if there is a lot of heap remaining unallocated then
        the value of configTOTAL_HEAP_SIZE in FreeRTOSConfig.h can be
        reduced accordingly. */
    }
}
/*-----------------------------------------------------------*/

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 );

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