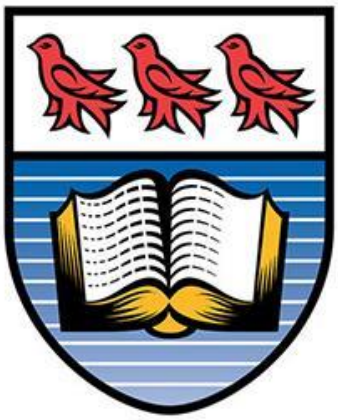
University of Victoria Department of Electrical and Computer Engineering



ECE 455 Project 2: Deadline-Driven Scheduler

Date of Submission: April 1, 2026

Clifton Tollefson - V01004341
Khephren Gould - V01012827

Table of Contents

1. Introduction

The goal of this project was to design and implement a custom scheduler. The specific type of scheduler considered was a deadline-driven scheduler that implements the EDF (Earliest Deadline First) scheduling scheme. In EDF, tasks are scheduled based on how far away their absolute deadline occurs in time. Tasks with the closest deadline are assigned the highest priority. EDF is a dynamic scheduling algorithm. This means it can schedule tasks without necessarily knowing their parameters (execution time, phase, period, etc.) in advance. In an ideal case (no self suspension or scheduler overhead), the test in equation 1 is a necessary and sufficient condition to ensure the schedulability of a set of tasks:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1 \quad (1)$$

Where n is the number of tasks, e_i is the task execution time, and p_i is the task period. In this lab, test benches for which this condition is satisfied and not satisfied were used to test the scheduler [1].

2. System Overview

A block diagram showing the high level design of the system is shown in Figure 1:

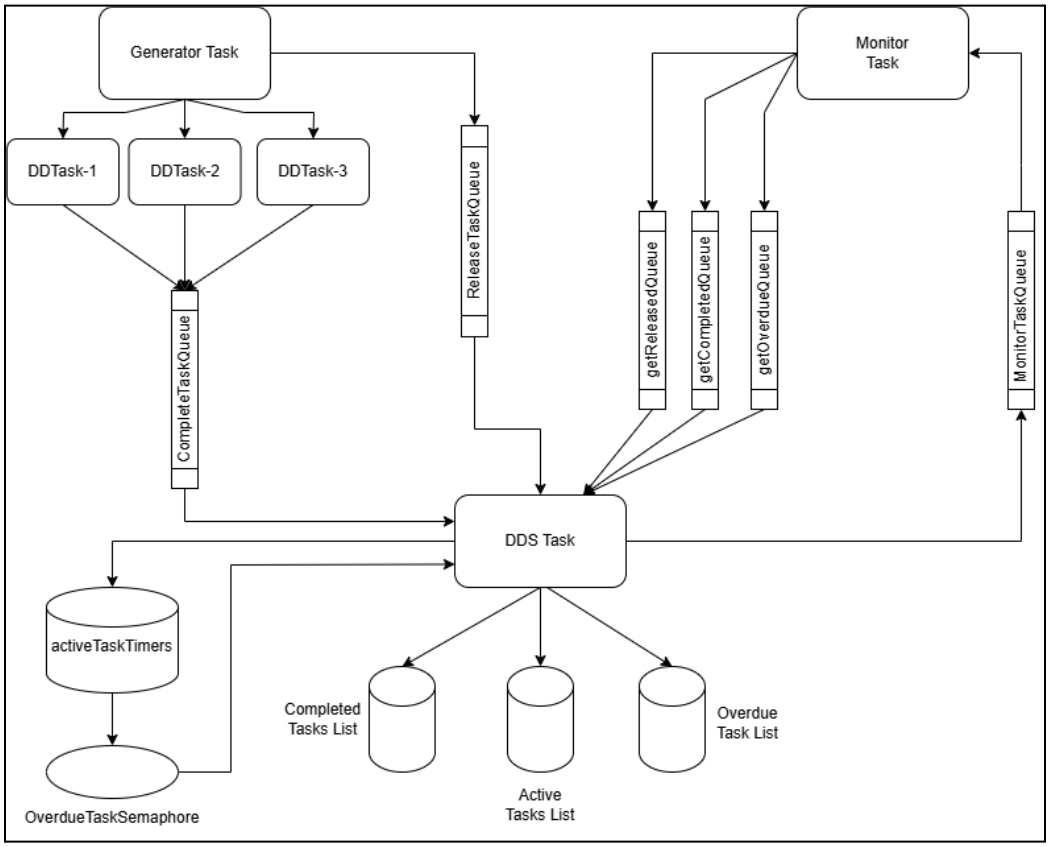


Figure 1: System Overview

In total, the system consists of three core tasks and any number of added user-defined tasks. For the testbenches applied, the number of user-defined DD tasks is 3. All inter-task communication is implemented with queues. There are six total

queues employed for this purpose. Additional resources consist of timers and a semaphore to ensure tasks are released according to their periods and overdue tasks are detected. The task generator uses a single timer to periodically release all user-defined DD Tasks. The DDS dynamically creates a timer for each task released to detect overdue tasks. The timer callbacks give a semaphore to the DDS to signal that a task must be moved to the overdue task list.

Finally, important modifications were made to FreeRTOSconfig.h in order to accommodate the needs of the project. First, the heap used was changed from heap1.c to heap4.c. This was required as heap1.c does not implement of the free function included in the c standard library. Free is required due to the data structure (a linked list) employed to track the status of the tasks. When a node is no longer required, the memory allocated to it must be returned to the operating system. Otherwise, the heap will eventually be filled. Additionally, configMAX_PRIORITY was set to 5. This allowed for task priorities to be set as follows:

1. FreeRTOS native scheduler - 5
2. DDS - 4
3. Task generator - 4
4. Earliest deadline task - 3
5. All other DD tasks - 2

The complete listing for FreeRTOSconfig.h can be found in Listing 8.

2. Design Solution

The design was divided into three modules: the dds, the task generator, and the monitor task. Each module is described individually.

2.1 deadline-driven Scheduler:

The algorithm employed for the DDS Task is shown in Figure 2.

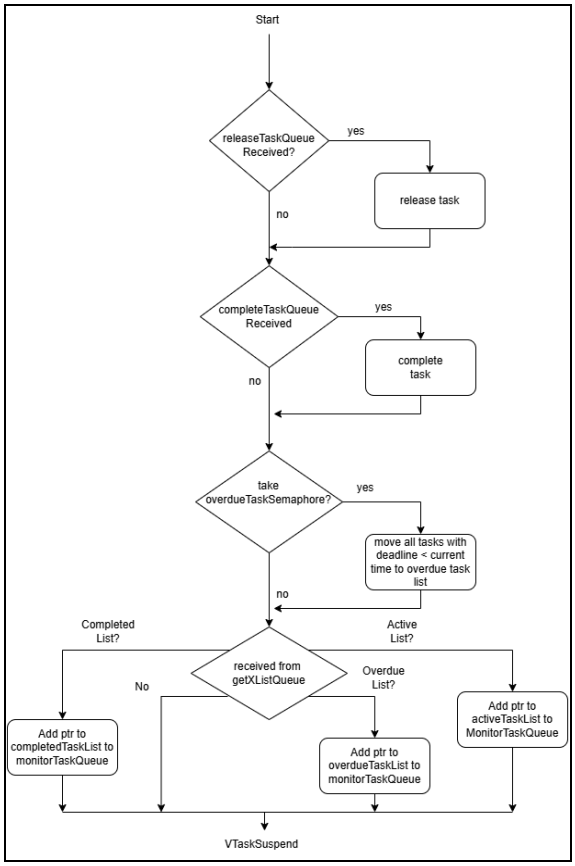


Figure 2: DDS Flowchart

The DDS is implemented as an FTask with the highest priority. This means that it will starve all other tasks whenever it is not blocked or suspended. It is only resumed by the API functions it exposes and it suspends itself after completing one iteration to minimize its effect on other tasks (minimize the scheduler overhead) accordingly. To minimize jitter, the DDS checks for tasks to be released in the releasedTaskQueue first. If the DDS receives an item from the queue, it releases a task. The function exposed by the DDS to add to this queue, along with the logic employed when a response is received are shown in Figures 3 and 4 respectively.

```
void release_dd_task(TaskHandle_t t_handle,task_type type, int task_id, int absolute_deadline){
        dd_task task_struct =  {
                .t_handle = t_handle,
                .type = type,
                .task_id = task_id,
                .release_time =  xTaskGetTickCount(),
                .absolute_deadline = absolute_deadline
        };
        printf("event: %d %s released at %d, task id:
%d\n",event_id,pcTaskGetName(t_handle),task_struct.release_time,task_id);
        event_id++;
        xQueueSend(released_task_queue,&task_struct,0);
        vTaskResume(dds_task_handle);
}
```

Figure 3: release_dd_task function

```
//release task has been called
if(xQueueReceive(released_task_queue,&temp_task,0) != errQUEUE_EMPTY){
        TimerHandle_t handle =  xTimerCreate("timer", temp_task.absolute_deadline -
```

```
xTaskGetTickCount(),0,temp_task.task_id, task_overdue_callback);
        xTimerStart(handle,0);
        timer_struct temp = {
                .handle = handle,
                .timer_id = temp_task.task_id
        };
        timer_array[timer_count++] = temp;
        vTaskResume(temp_task.t_handle);
        released_task_list = insertAtHead(released_task_list,temp_task);
        //increment the list size variable every time a task is released
        task_list_size++;

        //sort the list
        released_task_list = sort_list(released_task_list,task_list_size);
}
```

Figure 4: release task logic

As shown in Figure 3, the release_dd_task function receives the task parameters as arguments and creates a data structure to allow efficient transfer of all parameters using a queue. The function adds the newly created dd_task struct to the appropriate queue. Before returning, the function resumes (unsuspends) the DDS so it can process the item in the queue. The receiving logic (in Figure 4) consists of first creating a timer. This timer acts like a watchdog that monitors for the task going overdue. The timer is started immediately and implements a callback that triggers the DDS to search the active task list for overdue task. The only way to prevent the callback from running is by stopping the timer which occurs when the corresponding DD Task completes. Since the watchdog timers are created dynamically as tasks are released, an array (timer_array) is used to store all the handles corresponding to these watchdog timers. After creating the timer, the DDS resumes the F-task corresponding to the handle stored in the DD task struct allowing it to begin execution after the dds suspends. Next, the dds inserts the newly received dd task into the released task queue using a standard linked list function to insert a node at the head position. Finally, the dds sorts the list to adjust task priorities based on the newly received task. The sort function implements the selection sort algorithm for linked lists as shown in Figure 5:

```
static dd_task_node* sort_list(dd_task_node* head, int list_size){
	if(list_size == 0){
		return head;
	}
	dd_task_node* new_head = NULL;
	for(int i = list_size; i > 0; i--){
		dd_task_node* current = head;
		dd_task_node* prev = NULL;
		dd_task_node* before_furthest = NULL;
		dd_task_node* furthest_deadline = NULL;
		while(current != NULL){
			if(furthest_deadline == NULL || (current->task.absolute_deadline >
furthest_deadline->task.absolute_deadline)){
				before_furthest = prev;
				furthest_deadline = current;
			}
			//clear all task priorities back to 2
			vTaskPrioritySet(current->task.t_handle,2);
			prev = current;
			current = current->next;
		}
		if(before_furthest == NULL){
			head = furthest_deadline->next;
		}else{
			before_furthest->next = furthest_deadline->next;
		}
		furthest_deadline->next = new_head;
		new_head = furthest_deadline;
	}
	//set the task at the front of the list (earliest deadline) to 3
	vTaskPrioritySet(new_head->task.t_handle,3);
	return new_head;
}
```

Figure 5: Selection Sort Algorithm

The sorting algorithm works by repeatedly finding the task with the furthest deadline in the list provided. While sorting the list, the algorithm resets the priorities of all tasks to the base priority of 2. After finding the item with the furthest deadline, the item is moved to the front of a new list and removed from the old list. When the outer loop resumes, the furthest deadline out of all tasks still present in the provided list is found and added to the front of the new list. As the loop continues to iterate tasks with earlier deadlines are progressively found and added to the new list. At the last iteration, the furthest deadline found will be the earliest deadline and is placed at the front of the new list accordingly. When complete, the sort algorithm assigns the task at the front of the new list to be a higher priority than the rest and returns the new list.

Due to the presence of the nested loop, the algorithm will be $O(n^2)$ in time (where n is the list length). Potential improvements are discussed in the sections below. A flowchart for the algorithm is provided in Figure 6 below:

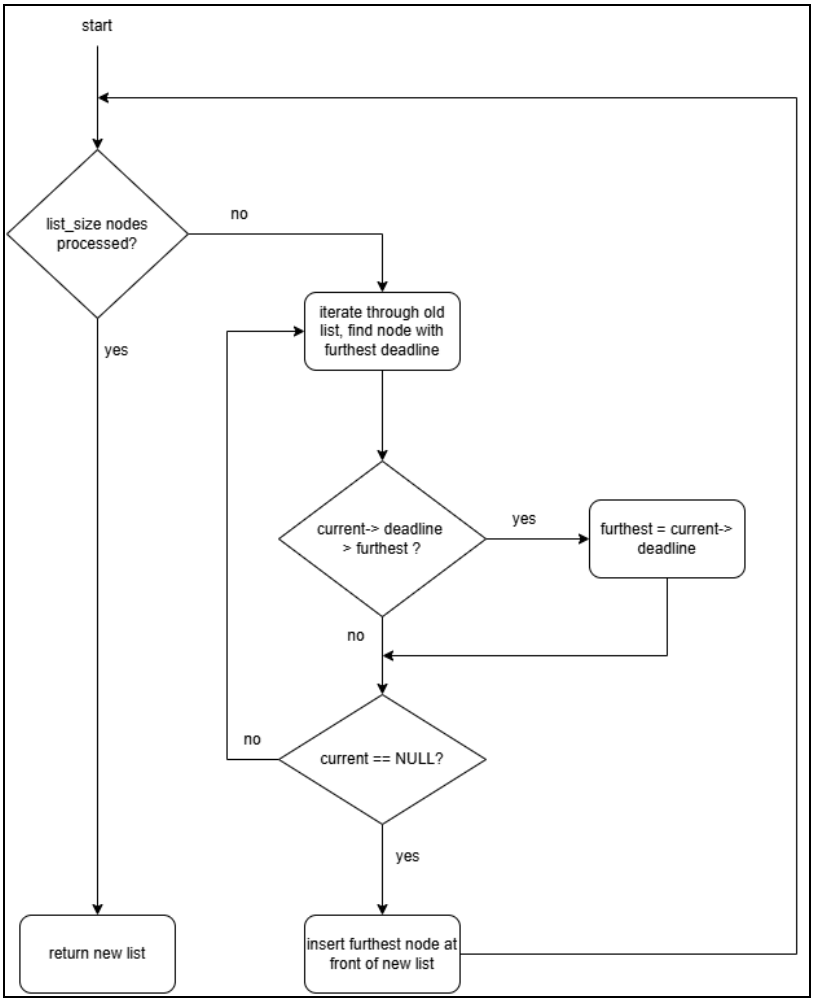


Figure 6: Selection Sort Algorithm Flowchart

The function exposed by the dds to complete tasks, along with the logic employed to process newly completed tasks are shown in Figures 7 and 8 respectively.

```
//Function called by user tasks to signal completion to the dds
void complete_dd_task(uint32_t task_id){
	xQueueSend(completed_task_queue, &task_id,0);
	vTaskResume(dds_task_handle);
}
```

Figure 7: Complete dd task API

```
//complete task has been called
if(xQueueReceive(completed_task_queue,&temp_id,0) != errQUEUE_EMPTY){
//remove from the released_task
	timer_struct temp_array[20] = {};
	int j = 0;
	for(int i = 0; i < timer_count; i++){
		if(timer_array[i].timer_id == temp_id){
			xTimerDelete(timer_array[i].handle,0);
		}else{
			temp_array[j] = timer_array[i];
			j++;
		}
	}
	timer_count--;
```

```
        for(int i = 0; i < timer_count; i++){
                timer_array[i] = temp_array[i];
        }

        dd_task removed_task = removebyId(temp_id,&released_task_list);
        printf("event: %d %s complete at %d, task id:
%d\n",event_id,pcTaskGetName(removed_task.t_handle),xTaskGetTickCount(),removed_task.task_id);
        event_id++;
        removed_task.completion_time = xTaskGetTickCount();
        completed_task_list = insertAtHead(completed_task_list,removed_task);


        vTaskDelete(removed_task.t_handle);
        //decrement the list size variable every time a task completes
        task_list_size--;
        //sort the list to assign the next task to run
        released_task_list = sort_list(released_task_list,task_list_size);
}
```

Figure 8: DDS Complete Task Logic

The complete_dd_task api function simply receives the id of the task to be marked as completed and forwards it to a queue being monitored by the dds. It resumes the dds to allow it to process the item in the queue. When the dds receives an item from the completed_task_queue it starts by deleting the watchdog timer associated with the task. This ensures the timer callback will not run and the task will not be marked as overdue. Next, the dds employs a removeByID function that finds the task with the provided id , removes it from the released task list, and returns the removed task. The return value of this function is provided to the insertAtHead function which places it at the front of the completed tasks list. Following this, the released task is sorted again to account for the completed task being removed. The removeByID function is shown in Figure 9.

```
//takes a list and id. searches list for task with the id and removes the task
static dd_task removebyId(int id,dd_task_node** list){
//find list node with the provided id
        dd_task_node* current = *list;
        dd_task_node* prev = NULL;
        dd_task temp_task;
        while(current != NULL){
                if(current->task.task_id == id){
                        temp_task = current->task;
                        if(prev == NULL){
                                *list = current->next;
                        }else{
                                prev->next = current->next;
                        }
                        free(current);
                        current = NULL;
                }else{
                        prev = current;
                        current = current->next;
                }
        }
        return temp_task;
}
```

Figure 9: removeByID linked list function

The logic for handling overdue tasks is shown in Figures 10 and 11.

```
void task_overdue_callback(TimerHandle_t xTimer){
	int temp =  (int)pvTimerGetTimerID(xTimer);
	timer_struct temp_array[20] = {};
	int j = 0;
	for(int i = 0; i < timer_count; i++){
		if(timer_array[i].timer_id == temp){
			xTimerDelete(timer_array[i].handle,0);
		}else{
			temp_array[j] = timer_array[i];
			j++;
		}
	}
	timer_count--;
	for(int i = 0; i < timer_count; i++){
		timer_array[i] = temp_array[i];
	}
	xSemaphoreGive(overdue_task_semaphore);
	vTaskResume(dds_task_handle);
}
```

Figure 10: Watchdog timer callback

```
if(xSemaphoreTake(overdue_task_semaphore,0) == pdTRUE){
	dd_task temp = removebyDeadline(&released_task_list);
	overdue_task_list = insertAtHead(overdue_task_list,temp);
	task_list_size--;
	released_task_list = sort_list(released_task_list,task_list_size);
}
```

Figure 11: DDS Overdue task logic

The callback function is assigned as the timer callback for all watchdog timers created. In the callback, the timer that triggered the callback is deleted (so as to not waste heap space) and a semaphore is given. Before returning, the callback unsuspends the DDS.

The DDS attempts to take the semaphore on each iteration. If it succeeds in taking the semaphore a timer callback has been triggered indicating a task has gone overdue. The DDS responds by searching the released task list for tasks with an absolute deadline greater than the current time. The function (removeByDeadline) removes these tasks from the released task. This function operates very similarly to the removeByID function. The removed task is inserted at the front of the overdue task list and the release task list is sorted again to ensure removal of the overdue task is reflected in the released task priorities. It is important to note that this solution strictly removes all overdue tasks from the released list and thus if a task goes overdue, it will never complete.

The final function performed by the DDS is to provide functions to access the internal lists. To accomplish this, the DDS monitors three separate queues for requests for the lists. If a request is received, the DDS responds by inserting a pointer to the appropriate list to a response queue. The response queue is shared by all three lists. It's important to note that lists are passed by reference and thus may change if the response is not used promptly. To ensure the expected data is received, the task receiving from the queue should use the list immediately or immediately make a copy of the list data. Although this leads to a worse user experience, copying only the pointer saves scheduler overhead which was deemed more important. This logic is shown in Figure 11.

```
if(xQueueReceive(get_active_tasks_queue,&received_from_queue,0) == pdPASS){
        xQueueSend(monitor_task_queue,&released_task_list,0);
}
if(xQueueReceive(get_completed_tasks_queue,&received_from_queue,0) == pdPASS){
        xQueueSend(monitor_task_queue,&completed_task_list,0);
}
if(xQueueReceive(get_overdue_tasks_queue,&received_from_queue,0) == pdPASS){
        xQueueSend(monitor_task_queue,&overdue_task_list,0);
}
if(received_from_queue){
        vTaskResume(monitor_task_handle);
}
```

Figure 12: Getter function Request Processing

2.2 User-Defined DD Tasks:

The user-defined tasks are the actual deadline-driven tasks that are defined in each test bench. Each user-defined task is a regular FreeRTOS task which runs for the execution time specified by the testbench. To visualize which task is running, we use the integrated LEDs that are on the STM32 board. Task 1 is the green led, task 2 is the blue led and task 3 is the red led.

To ensure that the task is actually running for the specified time, we employ an empty while loop which runs for one clock tick. Before each loop, we store the current tick count in the variable “start_ticks”, we then compare the current tick count with this value and once 1 tick has elapsed, the loop condition is no longer true and thus the loop ends. We then can increment “expired_ticks”. Once the “expired_ticks” have reached the execution time of the task, the task is done and it calls the “complete_dd_tasks()” function and passes its task ID number. Below in Figure 12 is the program listing for task 1. Here we can see that at the beginning of the task the green led is turned on and the blue and red leds are turned off, all other functionality of the function are the same for task 2 and 3 and are shown in Appendix A under the task_generator.c file.

```
void task1(void* pvParameters){
        int id = pvParameters;
        int expired_ticks = 0;
        TickType_t start_ticks;
        STM_EVAL_LEDOn(green_led);
        STM_EVAL_LEDOff(red_led);
        STM_EVAL_LEDOff(blue_led);
        for(;;){
                start_ticks = xTaskGetTickCount();
                while(xTaskGetTickCount() == start_ticks){    //loop to wait 1 tick
                }
                expired_ticks++;
                if(expired_ticks > pdMS_TO_TICKS(TASK1_EXECUTION_TIME)){
                        complete_dd_task(id);
                }
        }
}
```

Figure 13: Program listing of user task 1

DD Task Generator:

The generator task is a freeRTOS task which sits in the dormant state unless prompted to release a new task. We use only 1 generator task with 3 timers, all with the same callback function which resumes the generator task. Each timer is set as a one-shot timer with the period given in the testbench.

Once the timer is triggered, the callback function runs which resumes the generator task. Inside the generator task, it uses the freeRTOS function “xTimerIsTimerActive()” to check if that timer is currently running or not. If the timer is not running, we know that we must release a new task which corresponds to that timer. For example if timer 1 goes off then we need to release a new instance of task 1. The full algorithm is shown below in figure 13 to better illustrate this:

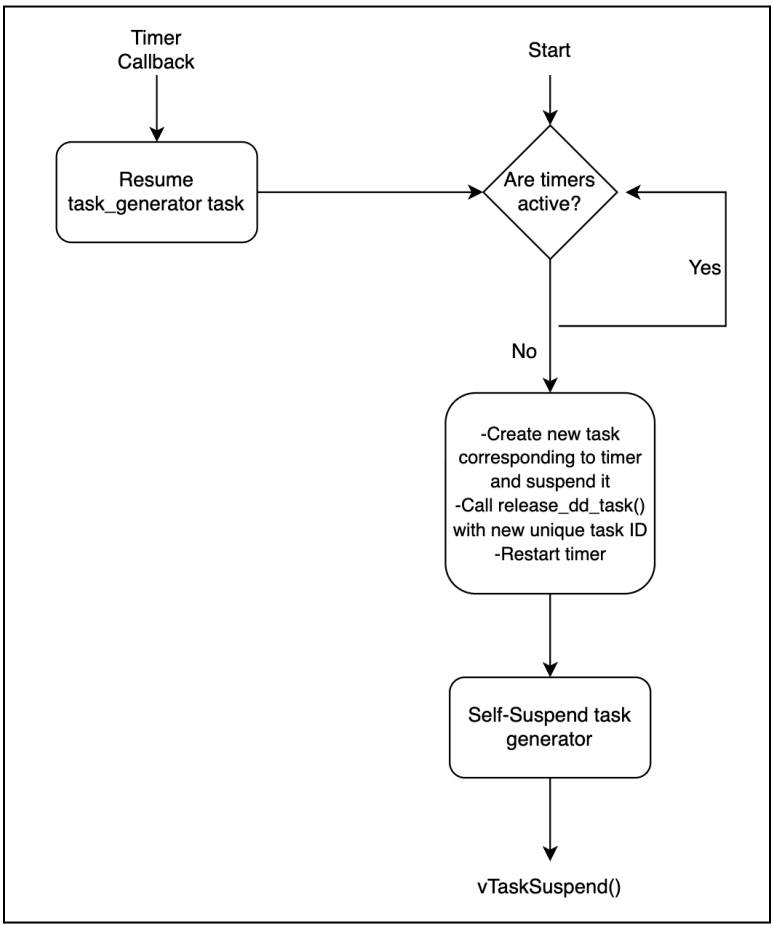


Figure 14: deadline-driven Task Generator flow chart

To release a task, we first must create it using “xTaskCreate”, we reuse the same task listing and handles for each instance of each task, however each task has a unique task id which is stored in the pvParameters to be used in the user task. Once the task is created we immediately suspend it since it is the DDS who officially releases the task. Next to communicate with the DDS, we call “release_dd_task” which was explained in section 2.1. Now we start the timer again which will trigger the next task release.

A possible improvement on our code could be utilising periodic timers which would automatically reload the task each time they expire. This would reduce the jitter experience between the time a timer goes off and when the next one is started. Although the time it takes for the system to create a task and call release_dd_task() is short, it is non-zero meaning as we run our scheduler for longer, the tasks will be released later and later.

Once the generator task has checked each timer and created the necessary tasks, it suspends itself to yield to other tasks. The task will remain dormant until a new timer is triggered which resumes the task once more. The full program listing of the generator task can be seen in figure 14 below:

```
void generator_task_callback(TimerHandle_t xTimer){
	vTaskResume(generator_task_handle);
}

void generator_task(void* pvParameters){
	int timer1_id = TIMER1_ID;
	int timer2_id = TIMER2_ID;
	int timer3_id= TIMER3_ID;
	TimerHandle_t timer_1 = xTimerCreate("Task1
Timer",pdMS_TO_TICKS(TASK1_PERIOD),0,&timer1_id,generator_task_callback);
	TimerHandle_t timer_2 = xTimerCreate("Task2
Timer",pdMS_TO_TICKS(TASK2_PERIOD),0,&timer2_id,generator_task_callback);
	TimerHandle_t timer_3 = xTimerCreate("Task3
Timer",pdMS_TO_TICKS(TASK3_PERIOD),0,&timer3_id,generator_task_callback);
	int task_id = 1;
	for(;;){
		if(!xTimerIsTimerActive(timer_1) && task1_type == PERIODIC){
			xTaskCreate(task1,"task 1", configMINIMAL_STACK_SIZE, task_id,2,&task_handle1);
			vTaskSuspend(task_handle1);
			release_dd_task(task_handle1,task1_type,task_id, xTaskGetTickCount() +
pdMS_TO_TICKS(TASK1_PERIOD));
			task_id++;
			xTimerStart(timer_1,0);
		}
		if(!xTimerIsTimerActive(timer_2) && task2_type == PERIODIC){
			xTaskCreate(task2,"task 2", configMINIMAL_STACK_SIZE, task_id,2,&task_handle2);
			vTaskSuspend(task_handle2);
			release_dd_task(task_handle2,task2_type,task_id, xTaskGetTickCount() +
pdMS_TO_TICKS(TASK2_PERIOD));
			task_id++;
			xTimerStart(timer_2,0);
		}
		if(!xTimerIsTimerActive(timer_3) && task3_type == PERIODIC){
			xTaskCreate(task3,"task 3", configMINIMAL_STACK_SIZE, task_id,2,&task_handle3);
			vTaskSuspend(task_handle3);
			release_dd_task(task_handle3,task3_type,task_id, xTaskGetTickCount() +
pdMS_TO_TICKS(TASK3_PERIOD));
			task_id++;
			xTimerStart(timer_3,0);
		}
		vTaskSuspend(generator_task_handle);
	}
}
```

Figure 15: Program listing of task generator task

Monitor Task

The monitor task, prints out diagnostics to monitor the operation of the dds and the task generator. It operates by periodically making requests for the internal lists stored by the DDS using `get_active_dd_task_list()`, `get_overdue_dd_task_list()` , and `get_completed_dd_task_list()` respectively. The overhead introduced by the monitor task using the scheme illustrated by the code in Figure 16. Note that the listSize is a simple $O(n)$ order algorithm that iterates through the list a single time and counts the number of nodes.

```
void monitor_task(void* pvParameters){
        dd_task_node* active_task_list;
        dd_task_node* completed_task_list;
        dd_task_node* overdue_task_list;
        monitor_task_timer = xTimerCreate("Monitor Task Timer",HYPERPERIOD,pdTRUE,0,monitor_task_callback);
        xTimerStart(monitor_task_timer,0);
        for(;;){
                if(xQueueReceive(monitor_task_queue,&active_task_list,0)){
                        printf("\nNumber of active tasks DD-Tasks: %d\n",listSize(active_task_list));
                }
                if(xQueueReceive(monitor_task_queue,&completed_task_list,0)){
                        printf("Number of completed tasks DD-Tasks: %d\n",listSize(completed_task_list));
                }
                if(xQueueReceive(monitor_task_queue,&overdue_task_list,0)){
                        printf("Number of overdue tasks DD-Tasks: %d\n\n",listSize(overdue_task_list));
                }
                vTaskSuspend(monitor_task_handle);
        }

}
```

Figure 16: Monitor Task Logic

The period is enforced by using a timer set to autoreload and set to a period of one hyperperiod which is defined along with the other testbench parameters. The timer callback is shown in Figure 17.

```
void monitor_task_callback(TimerHandle_t xTimer){
        get_active_dd_task_list();
        get_overdue_dd_task_list();
        get_completed_dd_task_list();
       vTaskResume(monitor_task_handle);
        vTaskResume(dds_task_handle);
}
```

Figure 17: Monitor Task Callback

The callback must resume the dds to ensure the requests are processed as soon as they are made. This ensures minimal jitter between when the requests are made (every hyperperiod) and when the data is received and displayed.

Discussion

To test the functionality for our design solution, 3 test benches were provided in the lab manual. By comparing the output of our scheduler to the expected scheduling results, we can assess the performance and validate our results. The test benches can be seen in figure 18 below.

	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

Figure 18: DDS Test Benches provided in lab manual

Testbench 1:

Table 1: Testbench 1 Event Trace

Event ID	Event Description	Time	Expected Time
0	task 1 released	0	0
1	task 2 released	1	0
2	task 3 released	2	0
3	task 1 complete	96	95
4	task 2 complete	247	245
5	task 3 complete	498	495
6	task 1 released	501	500
7	task 2 released	503	500
8	task 1 complete	597	595
9	task 2 complete	748	745
10	task 3 released	753	750
11	task 1 released	1003	1000
12	task 3 complete	1004	1000
13	task 2 released	1004	1000
14	task 1 complete	1101	1095
15	task 2 complete	1252	1245

Table 2: Testbench 1 Results

-	Measured	Expected
Number of active tasks DD-Tasks	0	0
Number of completed tasks DD-Tasks	8	8
Number of overdue tasks DD-Tasks	0	0

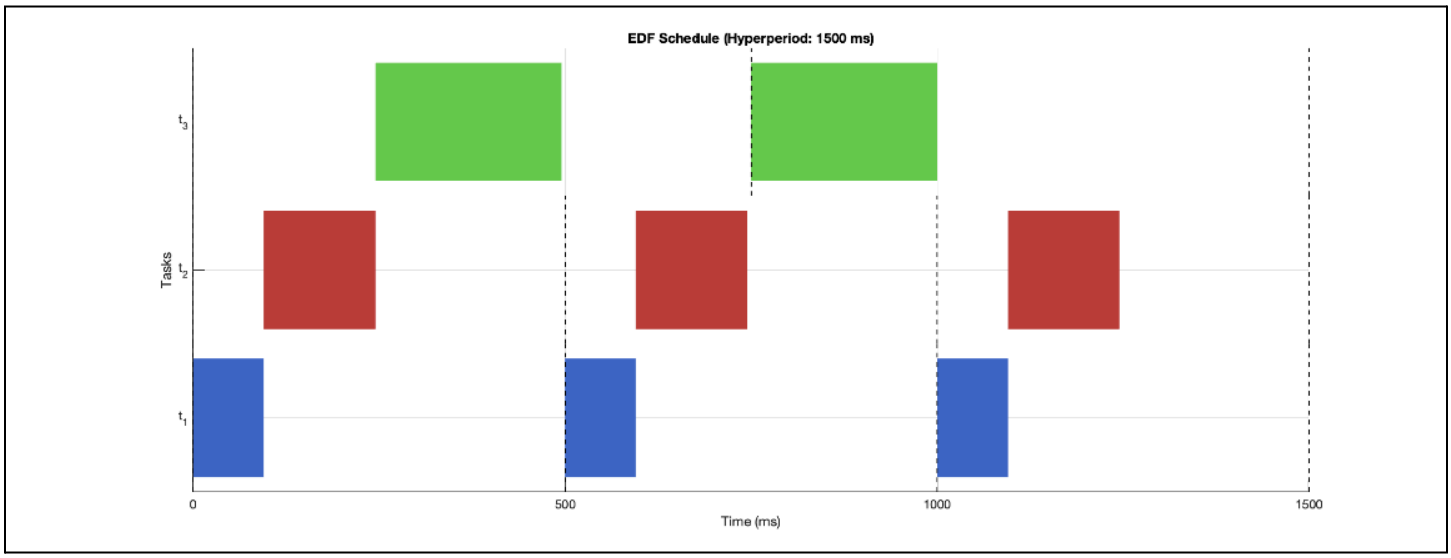


Figure 18: Testbench 1 scheduling diagram

The utilization for testbench 1 (equation 1 from above) is:

$$U = \frac{95}{500} + \frac{150}{500} + \frac{250}{750} = 0.823 = 82.3\%$$

Since $U \leq 1$ is a necessary and sufficient condition for schedulability under EDF, we would expect no overdue tasks after one hyperperiod. This is reflected in the results of the testbench.

Testbench 2:

Table 3: Testbench 2 Event Trace

Event ID	Event Description	Time	Expected Time
0	task 1 released	0	0
1	task 2 released	1	0
2	task 3 released	2	0
3	task 1 complete	96	95
4	task 2 complete	247	245
5	task 1 released	252	250
6	task 1 complete	348	345
7	task 2 released	503	500
8	task 1 released	504	500
9	task 3 complete	594	590
10	task 1 complete	690	685
11	task 3 released	754	750
12	task 1 released	756	750
13	task 2 complete	841	835
14	task 1 complete	937	930
15	task 2 released	1005	1000
16	task 1 released	1008	1000
17	task 1 complete	1104	1095
18	task 1 released	1260	1250
19	task 3 complete	1284	1275
20	task 2 complete	1435	1425

Table 4: Testbench 2 Results

-	Measured	Expected
Number of active tasks DD-Tasks	0	0
Number of completed tasks DD-Tasks	10	10

Number of overdue tasks DD-Tasks	1	1

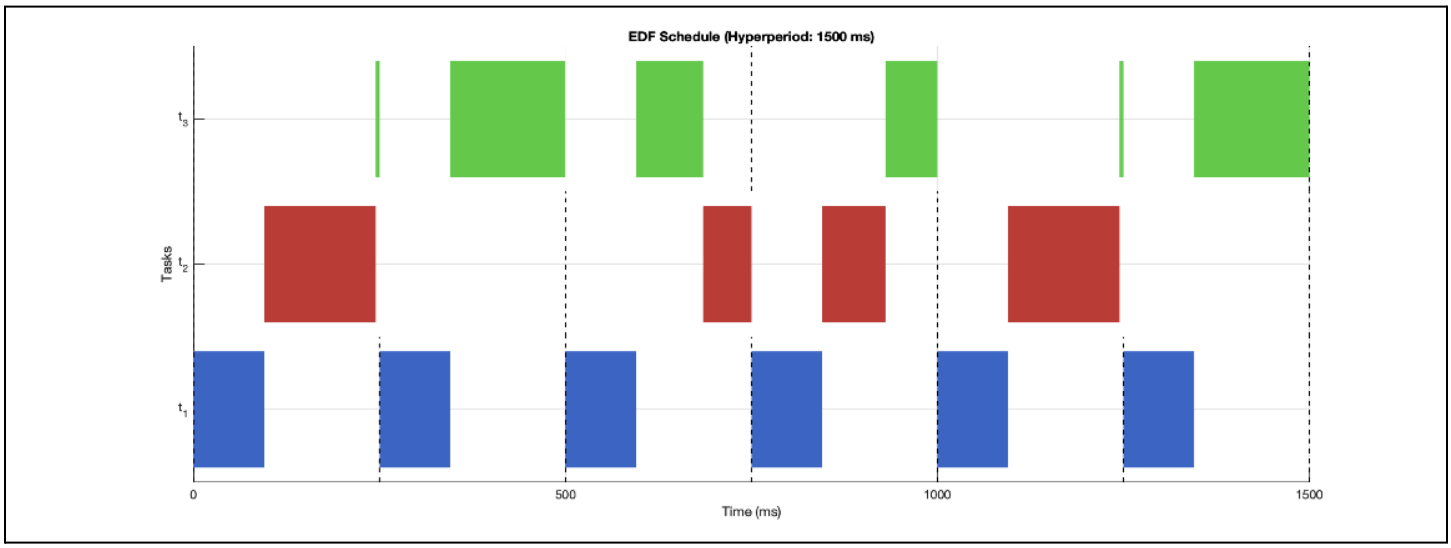


Figure 19: Testbench 2 scheduling diagram

The utilization rate for this test bench is shown below:

$$U = \frac{95}{250} + \frac{150}{500} + \frac{250}{750} = 1.013 = 101.3\%$$

Since the utilization rate is above 100%, this testbench is not schedulable under EDF. This is reflected in our results since we have 1 overdue task after the first hyperperiod.

Testbench 3:

Table 6: Testbench 3 Event Trace

Event ID	Event Description	Time	Expected Time
0	task 1 released	0	0
1	task 2 released	1	0
2	task 3 released	2	0
3	task 1 complete	101	100
4	task 2 complete	302	300
5	task 1 released	504	500
6	task 2 released	506	500
7	task 3 released	507	500

8	task 1 complete	605	600
9	task 2 complete	806	800

Table 5: Testbench 3 Results

-	Measured	Expected
Number of active tasks DD-Tasks	1	0
Number of completed tasks DD-Tasks	4	6
Number of overdue tasks DD-Tasks	1	0

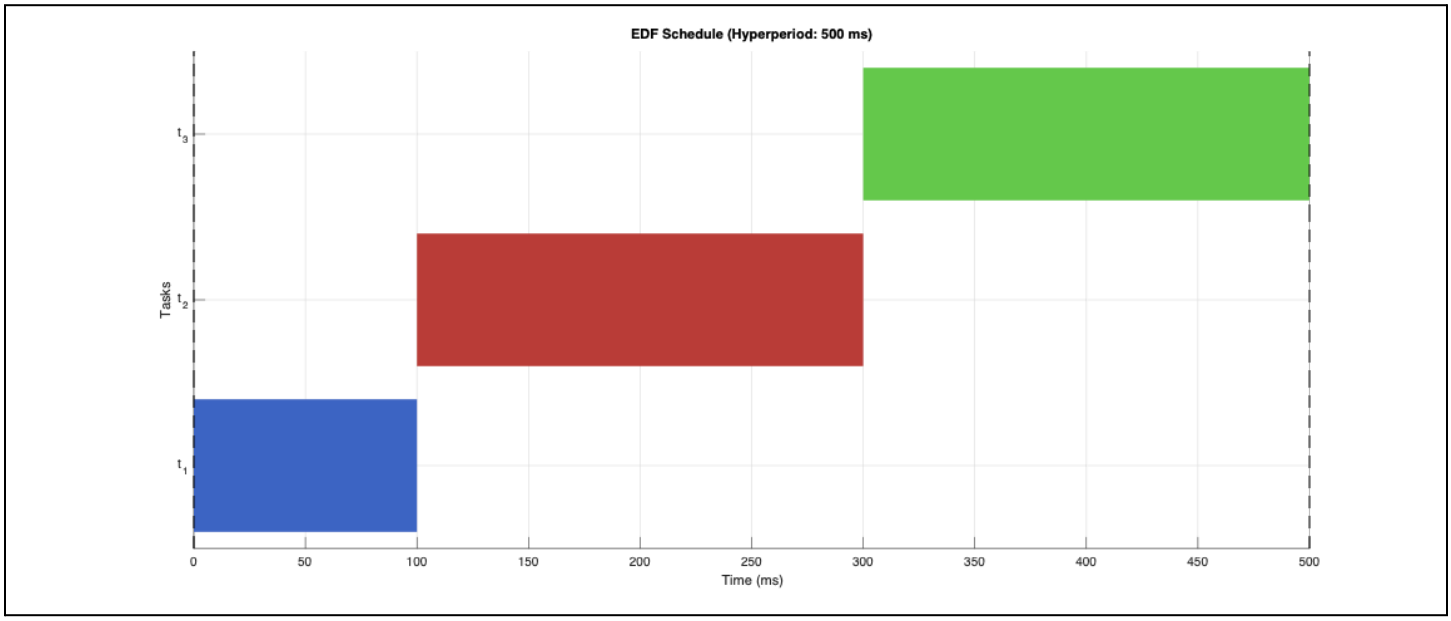


Figure 20: Testbench 3 scheduling diagram

The utilization for testbench 3 is:

$$U = \frac{100}{500} + \frac{200}{500} + \frac{300}{500} = 1 = 100\%$$

Since $U \leq 1$ is a necessary and sufficient condition for schedulability under EDF, we would expect this testbench to be schedulable with an ideal EDF scheduler. However, the results of the testbench show that after two hyperperiods, one overdue task has accumulated. This is due to the scheduler occupying some of the total utilization. This is known as scheduler overhead. This demonstrates one of the reasons why alternative schedulers (such as rate monotonic schedulers) are more popular in practice. Due to the frequent sorting and manipulating of the tasks to be scheduled, EDF schedulers incur a large scheduler overhead which makes predicting schedulability more difficult.

Limitations and Possible Improvements

When designing a deadline (EDF) driven scheduler, the main objective is to minimize scheduler overhead. To achieve this objective, design decisions such as processing requests to release new tasks first were made. However, there are numerous optimizations still left to be made. The first is to improve the efficiency of the sorting algorithm. Since sorting is performed on every released and completed task, the efficiency of this algorithm is critical to the performance of the dds. For the purposes of the lab, the selection sort algorithm was chosen. This was simply due to the designer's familiarity with

the algorithm. Upon conducting further research, algorithms with lower time complexity would be more suitable for this task. Merge sort, for example, is an $O(nlog(n))$ sorting algorithm that works by using a divide and conquer strategy [2].

Another potential optimization is to improve the reliability of the generator task. Prior to the demo, it was observed that on some occasions, the generator task would fail to create tasks properly. When a task is not properly created by the task generator, the DDS never receives a request to release the task and thus the output is not correct. The bug was not a failure in the dds, but rather was due to a timing bug in the task generator that caused it to fail. It is summarized below:

1. As shown in Figure 12, the user-defined task tracks its runtime by running a loop until a single tick has expired
    a. This code has an off by one error. By using > instead of >=, the task erroneously runs for execution time + 1 ticks
    b. In some cases this causes the timer expiry (discussed in the task generator above) to happen one tick later than expected. This causes the generator task to effectively miss the expiry of the timer and not release the task.

The situation described above was present in testbench 2. To fix the bug, the generator was programmed to check for expiry of the task 2 timer before the expiry of the task 1 timer. This allowed more time for the task 1 timer to expire and allowed the generator to function properly but required task 2 to be released before task 1. During the demo, this issue was discussed and the chosen fix was questioned. This fix was decided upon due to time constraints and the assumption that the actual order that tasks are released was not critical. The fix employed is not optimal for multiple reasons including:

1. It makes the output deviate slightly from the ordering in the lab manual making verification more difficult
2. It may not work for testbenches other than the 3 employed for the purposes of the lab

The proposed fix is to simply change the operator used from > to >=.

Another limitation is the difficulty in accommodating a larger task set as well as aperiodic tasks in the task generator. In order to test the dds more rigorously a more capable task generator would be beneficial. As of now, the task generator is limited to only 3 periodic tasks (although the dds was designed to accommodate aperiodic tasks, the task generator was not). This is due to the fact that the generator task creates a fixed number of timers (3) and sets the timers in autoreload mod. In order to make task generation more robust, a better scheme is needed. This could consist of allowing the generator task to accept an array of user-defined tasks and creating timers for each one in a loop. The timer mode (auto-reload or one-shot) could be configured per-task by checking the tasks type enum (PERIODIC or APERIODIC). With this simple addition, the generator task would be much more robust. A phase parameter could also be added to the dds_task struct to allow aperiodic (or periodic) tasks to be released at a user-defined time.

Summary

This project successfully implemented a deadline-driven scheduler using the Earliest Deadline First algorithm on an STM32 microcontroller running FreeRTOS. The system dynamically manages task priorities across three linked lists, released_task_list (all active tasks), completed_task_list (all successfully completed tasks), and overdue_task_list (all tasks that are overdue). All three testbenches were executed, with schedulable task sets producing zero overdue tasks and the over-utilized testbench correctly identifying deadline misses. Test bench 3 highlights the overhead needed to run an EDF scheduler can sometimes cause schedulable task sets in theory to become unschedulable in practice.

Overall, the key design goals were met, the DDS operates at the highest priority and suspends itself between events to minimize scheduling overhead, watchdog timers reliably detect overdue tasks, and the monitor task provides

per-hyperperiod visibility into scheduler state. Known limitations include an off-by-one error in task execution timing. This error is well-understood and carries a straightforward fix such as replacing > with >= which would improve correctness in future iterations.

References

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

[2] A. Ladha, "Lecture 2: Merge Sort and Master Theorem," *CS 3510 Algorithms*, Jan. 11, 2024.

Appendix A - Complete Application Code Listings

Listing 1: dds.h

```
#ifndef DDS_H_
#define DDS_H_
#include "../FreeRTOS_Source/include/FreeRTOS.h"
#include "../FreeRTOS_Source/include/queue.h"
#include "../FreeRTOS_Source/include/task.h"
#include "../FreeRTOS_Source/include/semphr.h"
typedef enum {PERIODIC, APERIODIC} task_type ;

typedef struct {
        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_node {
        dd_task task;
        struct dd_task_node* next;
} dd_task_node;
TaskHandle_t dds_task_handle;
QueueHandle_t released_task_queue;
QueueHandle_t completed_task_queue;
QueueHandle_t get_active_tasks_queue;
QueueHandle_t get_completed_tasks_queue;
QueueHandle_t get_overdue_tasks_queue;


SemaphoreHandle_t overdue_task_semaphore;
void get_active_dd_task_list();
void get_completed_dd_task_list();
void get_overdue_dd_task_list();
void release_dd_task(TaskHandle_t t_handle, task_type type, int task_id, int absolute_deadline);
void complete_dd_task(uint32_t task_id);
void dds_task(void * pvParameters);
#endif /* DDS_H_ */
```

Listing 2: dds.c

```
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include "../FreeRTOS_Source/include/timers.h"
#include "../FreeRTOS_Source/include/task.h"
#include "monitor_task.h"
#include "dds.h"

typedef struct{
        TimerHandle_t handle;
        int timer_id;
}timer_struct;

//array storing timers to monitor tasks going overdue
timer_struct timer_array[300] = {};
int timer_count = 0;
//global
int event_id = 0;


//function called by task generator to release a new user task
void release_dd_task(TaskHandle_t t_handle,task_type type, int task_id, int absolute_deadline){
        dd_task task_struct =  {
                .t_handle = t_handle,
                .type = type,
                .task_id = task_id,
                .release_time =  xTaskGetTickCount(),
                .absolute_deadline = absolute_deadline
        };
        printf("event: %d %s released at %d, task id:
%d\n",event_id,pcTaskGetName(t_handle),task_struct.release_time,task_id);
        event_id++;
        xQueueSend(released_task_queue,&task_struct,0);
```

```
        vTaskResume(dds_task_handle);
}

//Function called by user tasks to signal completion to the dds
void complete_dd_task(uint32_t task_id){
        xQueueSend(completed_task_queue, &task_id,0);
        vTaskResume(dds_task_handle);
}


//Private helper functions

//helper function used by the dds to create a new listnode
static dd_task_node* create_task_node(dd_task new_task){
        dd_task_node* new_node = (dd_task_node*)malloc(sizeof(dd_task_node));
        if(new_node == NULL){
                printf("Memory Allocation failed :(");
        }
        new_node->task = new_task;
        new_node->next = NULL;
        return new_node;
}
// helper function to insert a task at the front of the provided list
static dd_task_node* insertAtHead(dd_task_node* old_head, dd_task new_task){
        dd_task_node* new_node = create_task_node(new_task);
        new_node->next = old_head;
        return new_node;
}

//takes a list and id. searches list for task with the id and removes the task
static dd_task removebyId(int id,dd_task_node** list){

//find list node with the provided id
        dd_task_node* current = *list;
        dd_task_node* prev = NULL;
        dd_task temp_task;
        while(current != NULL){
                if(current->task.task_id == id){
                        temp_task = current->task;
                        if(prev == NULL){
                                *list = current->next;
                        }else{
                                prev->next = current->next;
                        }
                        free(current);
                        current = NULL;
                }else{
                        prev = current;
                        current = current->next;
                }
        }
        return temp_task;
}

//takes a list and its size, sorts the list in order of ascending deadline
static dd_task_node* sort_list(dd_task_node* head, int list_size){
        if(list_size == 0){
                return head;
        }
        dd_task_node* new_head = NULL;
```

```
        for(int i = list_size; i > 0; i--){
                dd_task_node* current = head;
                dd_task_node* prev = NULL;
                dd_task_node* before_furthest = NULL;
                dd_task_node* furthest_deadline = NULL;
                while(current != NULL){
                        if(furthest_deadline == NULL || (current->task.absolute_deadline >
furthest_deadline->task.absolute_deadline)){
                                before_furthest = prev;
                                furthest_deadline = current;
                        }
                        //clear all task priorities back to 2
                        vTaskPrioritySet(current->task.t_handle,2);
                        prev = current;
                        current = current->next;
                }
                if(before_furthest == NULL){
                        head = furthest_deadline->next;
                }else{
                        before_furthest->next = furthest_deadline->next;
                }
                furthest_deadline->next = new_head;
                new_head = furthest_deadline;
        }
        //set the task at the front of the list (earliest deadline) to 3
        vTaskPrioritySet(new_head->task.t_handle,3);
        return new_head;
}


void task_overdue_callback(TimerHandle_t xTimer){
        int temp =  (int)pvTimerGetTimerID(xTimer);
        timer_struct temp_array[20] = {};
        int j = 0;
        for(int i = 0; i < timer_count; i++){
                if(timer_array[i].timer_id == temp){
                        xTimerDelete(timer_array[i].handle,0);
                }else{
                        temp_array[j] = timer_array[i];
                        j++;
                }
        }
        timer_count--;
        for(int i = 0; i < timer_count; i++){
                timer_array[i] = temp_array[i];
        }
        xSemaphoreGive(overdue_task_semaphore);
        vTaskResume(dds_task_handle);
}

static dd_task removebyDeadline(dd_task_node** list){

//todo: find list node with the provided id
        dd_task_node* current = *list;
        dd_task_node* prev = NULL;
        dd_task temp_task;
        while(current != NULL){
                if(current->task.absolute_deadline <= xTaskGetTickCount()){
                        temp_task = current->task;
                        if(prev == NULL){
```

```
                        *list = current->next;
                }else{
                        prev->next = current->next;
                }
                vTaskDelete(current->task.t_handle);
                free(current);
                current = NULL;
        }else{
                prev = current;
                current = current->next;
        }
    }
    return temp_task;
}

void get_active_dd_task_list(){
    bool item_to_queue = true;
    xQueueSend(get_active_tasks_queue,&item_to_queue,0);
}

void get_overdue_dd_task_list(){
    bool item_to_queue = true;
    xQueueSend(get_overdue_tasks_queue,&item_to_queue,0);
}

void get_completed_dd_task_list(){
    bool item_to_queue = true;
    xQueueSend(get_completed_tasks_queue,&item_to_queue,0);
}

//main task that implements the dds
void dds_task(void * pvParameters){
    dd_task_node* released_task_list = NULL;
    dd_task_node* completed_task_list = NULL;
    dd_task_node* overdue_task_list = NULL;
    dd_task_node* temp_node;
    dd_task_node* temp_ptr;
    dd_task temp_task;
    int temp_id;
    int task_list_size = 0;
    for(;;){
        //release task has been called
        if(xQueueReceive(released_task_queue,&temp_task,0) != errQUEUE_EMPTY){
                TimerHandle_t handle =  xTimerCreate("timer", temp_task.absolute_deadline -
xTaskGetTickCount(),0,temp_task.task_id, task_overdue_callback);
                xTimerStart(handle,0);
                timer_struct temp = {
                                .handle = handle,
                                .timer_id = temp_task.task_id
                };
                timer_array[timer_count++] = temp;
                vTaskResume(temp_task.t_handle);
                released_task_list = insertAtHead(released_task_list,temp_task);
                //increment the list size variable every time a task is released
                task_list_size++;

                //sort the list
                released_task_list = sort_list(released_task_list,task_list_size);
        }
        //complete task has been called
```

```
                if(xQueueReceive(completed_task_queue,&temp_id,0) != errQUEUE_EMPTY){
                        //remove from the released_task
                        timer_struct temp_array[20] = {};
                        int j = 0;
                        for(int i = 0; i < timer_count; i++){
                                if(timer_array[i].timer_id == temp_id){
                                        xTimerDelete(timer_array[i].handle,0);
                                }else{
                                        temp_array[j] = timer_array[i];
                                        j++;
                                }
                        }
                        timer_count--;
                        for(int i = 0; i < timer_count; i++){
                                timer_array[i] = temp_array[i];
                        }

                        dd_task removed_task = removebyId(temp_id,&released_task_list);
                        printf("event: %d %s complete at %d, task id:
%d\n",event_id,pcTaskGetName(removed_task.t_handle),xTaskGetTickCount(),removed_task.task_id);
                        event_id++;
                        removed_task.completion_time = xTaskGetTickCount();
                        completed_task_list = insertAtHead(completed_task_list,removed_task);


                        vTaskDelete(removed_task.t_handle);
                        //decrement the list size variable every time a task completes
                        task_list_size--;
                        //sort the list to assign the next task to run
                        released_task_list = sort_list(released_task_list,task_list_size);
                }

                if(xSemaphoreTake(overdue_task_semaphore,0) == pdTRUE){
                        dd_task temp = removebyDeadline(&released_task_list);
                        overdue_task_list = insertAtHead(overdue_task_list,temp);
                        task_list_size--;
                        released_task_list = sort_list(released_task_list,task_list_size);
                }
                bool received_from_queue = false;
                if(xQueueReceive(get_active_tasks_queue,&received_from_queue,0) == pdPASS){
                        xQueueSend(monitor_task_queue,&released_task_list,0);

                }
                if(xQueueReceive(get_completed_tasks_queue,&received_from_queue,0) == pdPASS){
                        xQueueSend(monitor_task_queue,&completed_task_list,0);
                }
                if(xQueueReceive(get_overdue_tasks_queue,&received_from_queue,0) == pdPASS){
                        xQueueSend(monitor_task_queue,&overdue_task_list,0);
                }
                if(received_from_queue){
                        vTaskResume(monitor_task_handle);
                }

                vTaskSuspend(dds_task_handle);
        }

}
```

Listing 3: task_generator.h

```
#ifndef TASK_GENERATOR_H_
#define TASK_GENERATOR_H_
#include "../FreeRTOS_Source/include/FreeRTOS.h"
#include "../FreeRTOS_Source/include/task.h"

//testbench1
#define TASK1_PERIOD (500)
#define TASK2_PERIOD (500)
#define TASK3_PERIOD (750)
#define TASK1_EXECUTION_TIME (95)
#define TASK2_EXECUTION_TIME (150)
#define TASK3_EXECUTION_TIME (250)
#define HYPERPERIOD (1500)
//testbench2
//#define TASK1_PERIOD (250)
//#define TASK2_PERIOD (500)
//#define TASK3_PERIOD (750)
//#define TASK1_EXECUTION_TIME (95)
//#define TASK2_EXECUTION_TIME (150)
//#define TASK3_EXECUTION_TIME (250)
//#define HYPERPERIOD (1500)
//testbench3
//#define TASK1_PERIOD (500)
//#define TASK2_PERIOD (500)
//#define TASK3_PERIOD (500)
//#define TASK1_EXECUTION_TIME (100)
//#define TASK2_EXECUTION_TIME (200)
//#define TASK3_EXECUTION_TIME (200)
//#define HYPERPERIOD (500)

#define green_led LED4
#define red_led LED5
#define blue_led LED6

TaskHandle_t generator_task_handle;
void generator_task(void* pvParameters);


#endif /* TASK_GENERATOR_H_ */
```

Listing 4: task_generator.c

```
#include "task_generator.h"
#include "FreeRTOSConfig.h"
#include "dds.h"
#include <stdint.h>
#include <stdio.h>
#include "../FreeRTOS_Source/include/timers.h"
#include "stm32f4_discovery.h"
#define TIMER1_ID (1)
#define TIMER2_ID (2)
#define TIMER3_ID (3)

task_type task1_type = PERIODIC;
task_type task2_type = PERIODIC;
task_type task3_type = PERIODIC;
```

```
TaskHandle_t task_handle1;
TaskHandle_t task_handle2;
TaskHandle_t task_handle3;



void task1(void* pvParameters){
	int id = pvParameters;
	int expired_ticks = 0;
	TickType_t start_ticks;
	STM_EVAL_LEDOn(green_led);
	STM_EVAL_LEDOff(red_led);
	STM_EVAL_LEDOff(blue_led);
	for(;;){
		start_ticks = xTaskGetTickCount();
		while(xTaskGetTickCount() == start_ticks){	//loop to wait 1 tick
		}
		expired_ticks++;
		if(expired_ticks > pdMS_TO_TICKS(TASK1_EXECUTION_TIME)){
			complete_dd_task(id);
		}


	}
}
void task2(void* pvParameters){
	int id = pvParameters;
	int expired_ticks = 0;
	TickType_t start_ticks;
	STM_EVAL_LEDOn(blue_led);
	STM_EVAL_LEDOff(green_led);
	STM_EVAL_LEDOff(red_led);
	for(;;){
		start_ticks = xTaskGetTickCount();
		while(xTaskGetTickCount() == start_ticks){	//loop to wait 1 tick

		}

		expired_ticks++;
		if(expired_ticks > pdMS_TO_TICKS(TASK2_EXECUTION_TIME)){
			complete_dd_task(id);
		}


	}
}
void task3(void* pvParameters){
	int id = pvParameters;
	int expired_ticks = 0;
	TickType_t start_ticks;
	STM_EVAL_LEDOn(red_led);
	STM_EVAL_LEDOff(green_led);
	STM_EVAL_LEDOff(blue_led);
	for(;;){
		start_ticks = xTaskGetTickCount();
		//printf("in task 3 at %d ticks\n", start_ticks);
		while(xTaskGetTickCount() == start_ticks){	//loop to wait 1 tick
		}

		expired_ticks++;
```

```
                if(expired_ticks > pdMS_TO_TICKS(TASK3_EXECUTION_TIME)){
                        //printf("Task 3 completed at %d ticks\n", xTaskGetTickCount());
                        complete_dd_task(id);
                }


        }
}

void generator_task_callback(TimerHandle_t xTimer){
        vTaskResume(generator_task_handle);
}

void generator_task(void* pvParameters){
        int timer1_id = TIMER1_ID;
        int timer2_id = TIMER2_ID;
        int timer3_id= TIMER3_ID;
        TimerHandle_t timer_1 = xTimerCreate("Task1
Timer",pdMS_TO_TICKS(TASK1_PERIOD),0,&timer1_id,generator_task_callback);
        TimerHandle_t timer_2 = xTimerCreate("Task2
Timer",pdMS_TO_TICKS(TASK2_PERIOD),0,&timer2_id,generator_task_callback);
        TimerHandle_t timer_3 = xTimerCreate("Task3
Timer",pdMS_TO_TICKS(TASK3_PERIOD),0,&timer3_id,generator_task_callback);
        int task_id = 1;
        //initial task 1 release
        xTaskCreate(task1,"task 1", configMINIMAL_STACK_SIZE, task_id,2,&task_handle1);
        vTaskSuspend(task_handle1);
        release_dd_task(task_handle1,task1_type,task_id, xTaskGetTickCount() + pdMS_TO_TICKS(TASK1_PERIOD));
        task_id++;
        xTimerStart(timer_1,0);
        //initial task 2 release
        xTaskCreate(task2,"task 2", configMINIMAL_STACK_SIZE, task_id,2,&task_handle2);
        vTaskSuspend(task_handle2);
        release_dd_task(task_handle2,task2_type,task_id, xTaskGetTickCount() + pdMS_TO_TICKS(TASK2_PERIOD));
        task_id++;
        xTimerStart(timer_2,0);
        //initial task 3 release
        xTaskCreate(task3,"task 3", configMINIMAL_STACK_SIZE, task_id,2,&task_handle3);
        vTaskSuspend(task_handle3);
        release_dd_task(task_handle3,task3_type,task_id, xTaskGetTickCount() + pdMS_TO_TICKS(TASK3_PERIOD));
        task_id++;
        xTimerStart(timer_3,0);
        vTaskSuspend(generator_task_handle);
        for(;;){
                if(!xTimerIsTimerActive(timer_2) && task2_type == PERIODIC){
                        xTaskCreate(task2,"task 2", configMINIMAL_STACK_SIZE, task_id,2,&task_handle2);
                        vTaskSuspend(task_handle2);
                        release_dd_task(task_handle2,task2_type,task_id, xTaskGetTickCount() +
pdMS_TO_TICKS(TASK2_PERIOD));
                        task_id++;
                        xTimerStart(timer_2,0);
                }
                if(!xTimerIsTimerActive(timer_1) && task1_type == PERIODIC){
                        xTaskCreate(task1,"task 1", configMINIMAL_STACK_SIZE, task_id,2,&task_handle1);
                        vTaskSuspend(task_handle1);
                        release_dd_task(task_handle1,task1_type,task_id, xTaskGetTickCount() +
pdMS_TO_TICKS(TASK1_PERIOD));
                        task_id++;
                        xTimerStart(timer_1,0);
                }
```

```
                if(!xTimerIsTimerActive(timer_3) && task3_type == PERIODIC){
                        xTaskCreate(task3,"task 3", configMINIMAL_STACK_SIZE, task_id,2,&task_handle3);
                        vTaskSuspend(task_handle3);
                        release_dd_task(task_handle3,task3_type,task_id, xTaskGetTickCount() +
pdMS_TO_TICKS(TASK3_PERIOD));
                        task_id++;
                        xTimerStart(timer_3,0);
                }

                vTaskSuspend(generator_task_handle);


        }

}
```

Listing 5: monitor_task.h

```
#ifndef MONITOR_TASK_H_
#define MONITOR_TASK_H_
#include "../FreeRTOS_Source/include/FreeRTOS.h"
#include "../FreeRTOS_Source/include/task.h"
#include "../FreeRTOS_Source/include/timers.h"
#include "../FreeRTOS_Source/include/queue.h"
QueueHandle_t monitor_task_queue;
TaskHandle_t monitor_task_handle;
void monitor_task_callback(TimerHandle_t xTimer);

void monitor_task(void* pvParameters);

#endif /* MONITOR_TASK_H_ */
```

Listing 6: monitor_task.c

```
#include "monitor_task.h"
#include "dds.h"
#include "task_generator.h"

xTimerHandle monitor_task_timer;

static int listSize(dd_task_node* head){
        dd_task_node* current = head;
        int count = 0;
        while(current != NULL){
                count++;
                current = current->next;
        }
        return count;
}

void monitor_task_callback(TimerHandle_t xTimer){
        get_active_dd_task_list();
        get_overdue_dd_task_list();
        get_completed_dd_task_list();
        vTaskResume(dds_task_handle);

}
```

```
void monitor_task(void* pvParameters){
	dd_task_node* active_task_list;
	dd_task_node* completed_task_list;
	dd_task_node* overdue_task_list;
	monitor_task_timer = xTimerCreate("Monitor Task Timer",HYPERPERIOD,pdTRUE,0,monitor_task_callback);
	xTimerStart(monitor_task_timer,0);
	for(;;){
		if(xQueueReceive(monitor_task_queue,&active_task_list,0)){
			printf("\nNumber of active tasks DD-Tasks: %d\n",listSize(active_task_list));
		}
		if(xQueueReceive(monitor_task_queue,&completed_task_list,0)){
			printf("Number of completed tasks DD-Tasks: %d\n",listSize(completed_task_list));
		}
		if(xQueueReceive(monitor_task_queue,&overdue_task_list,0)){
			printf("Number of overdue tasks DD-Tasks: %d\n\n",listSize(overdue_task_list));
		}
		vTaskSuspend(monitor_task_handle);
	}

}
```

Listing 7: main.c

```
/* Standard includes. */
#include <stdint.h>
#include <stdio.h>
#include <stdbool.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"
/*User includes*/
#include "monitor_task.h"
#include "task_generator.h"
#include "dds.h"


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

/*
 * TODO: Implement this function for any hardware specific clock configuration
 * that was not already performed before main() was called.
 */
static void prvSetupHardware( void );

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

uint8_t ucHeap[configTOTAL_HEAP_SIZE];
int main(void)
{
```

```
	STM_EVAL_LEDInit(green_led);
	STM_EVAL_LEDInit(red_led);
	STM_EVAL_LEDInit(blue_led);
	/* Configure the system ready to run the demo.  The clock configuration
	can be done here if it was not done before main() was called. */
	prvSetupHardware();
	get_active_tasks_queue = xQueueCreate(1,sizeof(bool));
	get_completed_tasks_queue = xQueueCreate(1,sizeof(bool));
	get_overdue_tasks_queue = xQueueCreate(1,sizeof(bool));
	monitor_task_queue = xQueueCreate(50, sizeof(dd_task_node*));
	released_task_queue = xQueueCreate(50, sizeof(dd_task));
	completed_task_queue =  xQueueCreate(50, sizeof(int));
	vQueueAddToRegistry(released_task_queue, "Released task queue");
	vQueueAddToRegistry(completed_task_queue, "Completed task queue");
	vQueueAddToRegistry(get_active_tasks_queue,"Active task queue");
	vQueueAddToRegistry(get_completed_tasks_queue,"Completed Task queue");
	vQueueAddToRegistry(get_overdue_tasks_queue,"overdue task queue");

	overdue_task_semaphore = xSemaphoreCreateBinary();
	xTaskCreate(monitor_task,"Monitor Task", configMINIMAL_STACK_SIZE,0,4,&monitor_task_handle);
	xTaskCreate(generator_task,"Generator Task",configMINIMAL_STACK_SIZE,0,3,&generator_task_handle);
	xTaskCreate(dds_task,"DDS Task",configMINIMAL_STACK_SIZE + 10000,0,4,&dds_task_handle);
	/* Start the tasks and timer running. */
	vTaskStartScheduler();

	return 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. */
	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 );
}
```

Listing 8: FreeRTOSconfig.h

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

/*-----------------------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *
 * See http://www.freertos.org/a00110.html.
 *----------------------------------------------------------*/

extern uint32_t SystemCoreClock;
#define configSUPPPORT_DYNAMIC_ALLOCATION 1
#define configAPPLICATION_ALLOCATED_HEAP 1
#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 ) ( 64 * 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

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES         0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Software timer definitions. */
#define configUSE_TIMERS                            1
#define configTIMER_TASK_PRIORITY             ( 3 )
#define configTIMER_QUEUE_LENGTH              5
#define configTIMER_TASK_STACK_DEPTH   ( configMINIMAL_STACK_SIZE * 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet              1
#define INCLUDE_uxTaskPriorityGet             1
#define INCLUDE_vTaskDelete                         1
#define INCLUDE_vTaskCleanUpResources  1
#define INCLUDE_vTaskSuspend                  1
#define INCLUDE_vTaskDelayUntil                     1
#define INCLUDE_vTaskDelay                          1

/* Cortex-M specific definitions. */
#ifdef __NVIC_PRIO_BITS
        /* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
        #define configPRIO_BITS               __NVIC_PRIO_BITS
#else
        #define configPRIO_BITS               4        /* 15 priority levels */
#endif

/* The lowest interrupt priority that can be used in a call to a "set priority"
function. */
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY                     0xf

/* The highest interrupt priority that can be used by any interrupt service
routine that makes calls to interrupt safe FreeRTOS API functions.  DO NOT CALL
INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
PRIORITY THAN THIS! (higher priorities are lower numeric values. */
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY  5

/* Interrupt priorities used by the kernel port layer itself.  These are generic
to all Cortex-M ports, and do not rely on any particular library functions. */
#define configKERNEL_INTERRUPT_PRIORITY               ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 -
configPRIO_BITS) )
/* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY  ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 -
configPRIO_BITS) )
```

```
/* Normal assert() semantics without relying on the provision of an assert.h
header file. */
#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ); }

/* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
standard names. */
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler

#endif /* FREERTOS_CONFIG_H */
```

Appendix B - Project Design Document

deadline-driven Scheduler

Requirements
Implements the EDF algorithm and controls the priorities of user-defined F-tasks from an actively-managed list of DD-Tasks. The DDS is an F-Task with the highest priority and is normally suspended unless a call is made to one of its interface functions. The dds is an F task with the ***highest priority.*** *Since the DDS is independent of the auxiliary tasks, auxiliary tasks should only access the DDS using the interface functions. Furthermore, auxiliary tasks should not have access to any internal data structures used by the DDS.*

1. *Organize DD Tasks into 3 lists*
    a. *Active task list (tasks that are not blocked and need to be scheduled)*
    b. *Completed task list (a list of tasks that have completed before deadline)*
    c. *Overdue Task list (a list of tasks that completed after their deadline)*
2. *Add all completed tasks to the completed task list*
3. *Add all overdue tasks to the overdue task list*
4. *Add newly generated tasks to the active task list.*
    a. *Sort the active task list by deadline (nearest in time to furthest in time)*

5. *Expose 5 functions to the user*
    a. *Release_dd_task*
        i. *A function that receives a task structure*
        ii. *Populates the release time member of the struct*
        iii. *Adds the struct to a queue that is read by the dds task*
    b. *Complete_dd_task*
        i. *Receives the ID of the task that has completed*
        ii. *Adds the ID to a queue*
    c. *Get_active_dd_task_list*
        i. *Sends a message to a queue. Once dds receives the message, dds sends the list in response. The function returns the list after it receives the response*
    d. *Get_completed_dd_task_list*
        i. *Sends a message to a queue. Once dds receives the message, dds sends the list in response. The function returns the list after it receives the response*
    e. *Get_overdue_dd_task_list*
        i. *Sends a message to a queue. Once dds receives the message, dds sends the list in response. The function returns the list after it receives the response*
6. *The DDS should process the messages received from each of the queues discussed above*
    a. *Upon receiving a message from release_dd_task the dds should*
        i. *Assign a release time to the new task*
        ii. *Add the dd task to the active task list*
        iii. *Sort the list by deadline*
        iv. *Set the priorities of the user-defined tasks accordingly*
    b. *Upon receiving a message from complete_dd_task the dds should*
        i. *Assign a completion time to the newly completed dd task*
        ii. *Remove the dd task from the active task list*
        iii. *Add it to the completed task list.*
        iv. *Sort the active task list again now that the task has been removed*
        v. *Set the priorities of the user-defined tasks accordingly*
    c. *Upon receiving a message from any of the get_xxxx_task_list functions*
        i. *The dds should send the respective list to a queue.*

Implementation

1. *When a task is created the dds should start a software timer. In the timer callback, the associated task should be moved to the overdue task list.*
2. *After a single execution, the dds should block itself indefinitely*
3. *When any of the functions exposed to the user are called the functions should unblock the dds before they return.*
4. *Task data structure*

```
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;
}
```

5. *List data structure*

a. *The following data structure should be used for each task list. This is a singly linked list*

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

6. *Function definitions*
   a. *Following will be used*

```
1. void create_dd_task(  TaskHandle_t t_handle,
                         task_type type,
                         uint32_t task_id,
                         uint32_t absolute_deadline,
                         );
2. void delete_dd_task(uint32_t task_id);
3. **dd_task_list get_active_dd_task_list(void);
```

7. *List sorting*
   a. *A sorting algorithm will be required. Ideally the sorting algorithm should be efficient to reduce jitter in the release time of each task since it will run each time a new task is released to the dds*
   b. *Some popular linked list sorting algorithms:*
      i. *Merge sort (good time complexity)*
      ii. *https://www.geeksforgeeks.org/dsa/merge-sort-for-linked-list/*
      iii. *https://www.geeksforgeeks.org/dsa/sorting-a-singly-linked-list/*
      iv. *https://leetcode.com/problems/sort-list/description/*

user-defined Tasks

Requirements:

1. *Can be an empty loop*
2. *Break out of loop when "execution time" amount of ticks have accumulated while running the task (don't use VTaskDelay !!)*
3. *We could turn on an LED to indicate which task we are in.*

Implementation

1. *Turn the LED on?*
2. *Empty loop*
3. *Must call complete_dd_task once complete*

deadline-driven task generator

Requirements:

1. *Responsible for periodically generating tasks*
2. *Prepares all necessary conditions for the task and then calls release_dd_task*
   a. *handle*
   b. *Task_id*
   c. *Task type (periodic/aperiodic)*
   d. *Absolute deadline*

Implementation

1. *May use single generator or multiple generator (1 for each task)*

2. *Create a timer with the value of each task's period.*
3. *When the timer goes off, start collecting the necessary parameters for the release_dd_task*
    a. *ID ~ have a variable that increments each time a task is generated*
    b. *Handle ~ depends on which timer has ended*
    c. *Assign type to periodic/aperiodic (will always be periodic so could hard code potentially)*
    d. *Absolute deadline ~ current time + period of the task*

Monitor task

Requirements

1. *Responsible for reporting the following information:*
    a. *Number of active tasks*
    b. *Number of completed tasks*
    c. *Number of overdue tasks*
2. *Uses the following tasks to access the information*
    a. *get_active_dd_task_list*
    b. *get_complete_dd_task_list*
    c. *Get_overdue_dd_task_list*
3. *Given a period report active completed overdue tasks at the period*

Implementation

1. *A low priority periodic task?*
    a. *Ensure it does not affect timing of user-defined tasks*
        i. *Call all functions in task main loop*
        ii. *Printf data to the monitor*
        iii. *Block for a set time*
    b. *A timer callback*
        i. *Very simple just call all functions in the callback and use printf*