Deadline-Driven Scheduler

Amaan Makhani
[redacted]
April 28, 2021
B02

1. Introduction

This project was aimed at designing a Deadline-Driven Scheduler (DDS). This scheduler was intended to manage tasks which occur in a dynamic environment. Each task managed has a hard deadline. The algorithm behind this scheduler was the Earliest Deadline First (EDF) algorithm. The EDF algorithm was chosen as it is aimed to maximize processor utilization and might be more able to produce a feasible schedule. EDF is not built into the FreeRTOS system so this scheduler is a customized solution built on top of the scheduler. [1]

Two types of tasks exist with the DDS:

1. DD-Task: is a task managed by the DDS. Tasks may be periodic or aperiodic.
2. F-Task: is a task managed by FreeRTOS.

The F-task scheduled to execute is based on a changing priority base. The priorities are managed using a sorted active list. The first task in the active list is the one executing as it has it's priority elevated prior to its execution. This results in the FreeRTOS scheduler only executing that task.

Numerous F-Tasks listed below are needed in order to test and provide the functionality of the DDS.

1. Deadline-Driven Scheduler: This task implements the EDF algorithm, controls the priorities of user-defined F-tasks, and maintains numerous lists to keep track of DD-Tasks.

2. User-Defined Tasks: This task consists of user provided deadline-sensitive code, however, in the testing of this system the code executes a set of delay code.

3. Deadline-Driven Task Generator: This task periodically creates DD-Tasks that the DDS needs to schedule.

4. Monitor Task: The monito task is responsible for extracting information from the DDS and reporting required scheduling information.

The F-tasks other than the DDS exist only to test the system, therefore, they are independent of the DDS.

2. Design Document

The design document shown below in figure 1 details the tasks, queues, functions, timers, and functions needed to implement the solution. This document was created prior to working on the project code. A more detailed document is included in the system overview section.

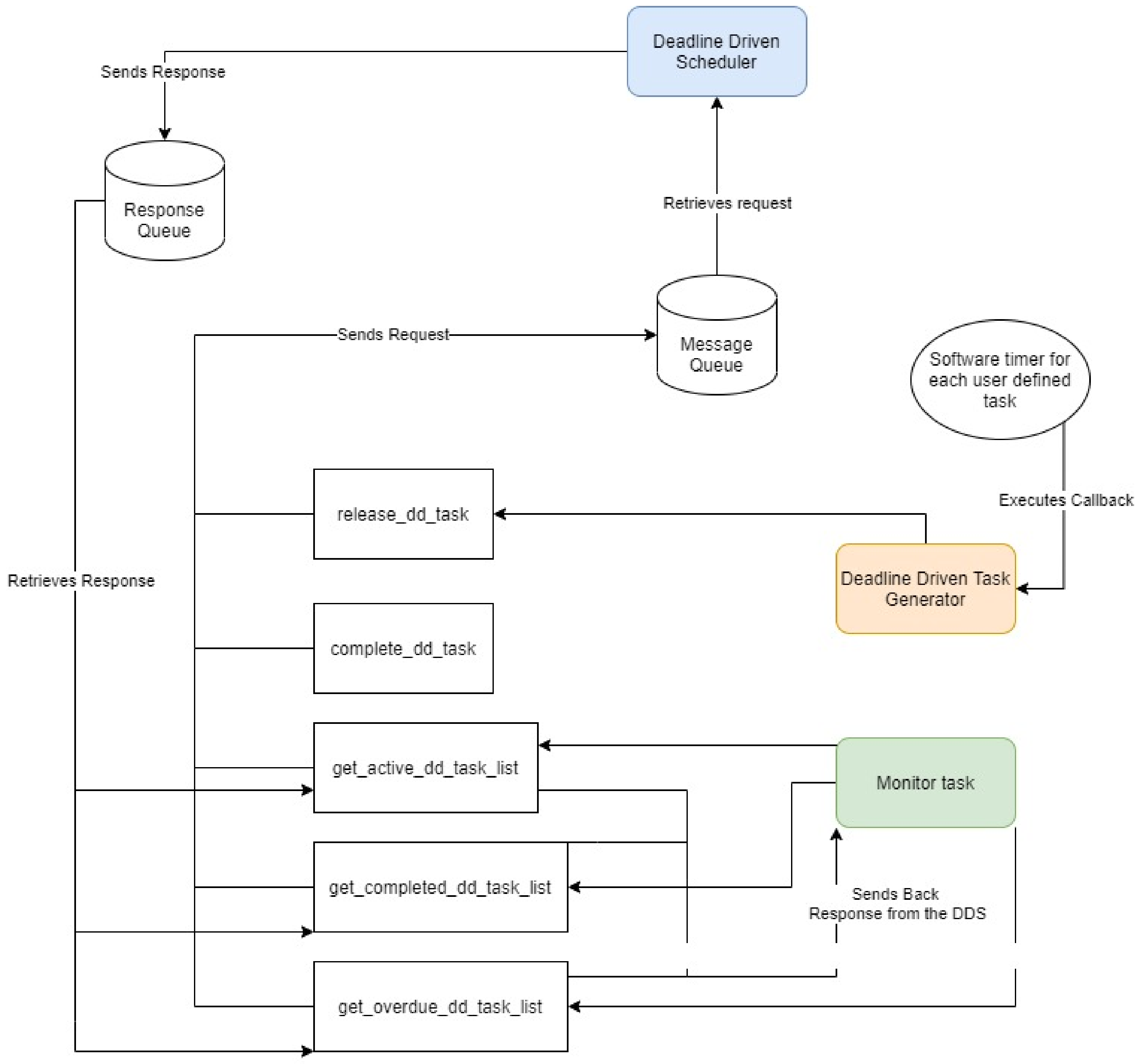


Figure 1: Design document

3. Discussion and Design Solution

This section further details the subsets of the solution and discusses the implementation of the solution as a whole.

3.1. System Overview

The system overview document is shown below, this details the interactions between F-Tasks, queues, and the DDS functions. This was a diagram created after the code was written and builds on the design document.

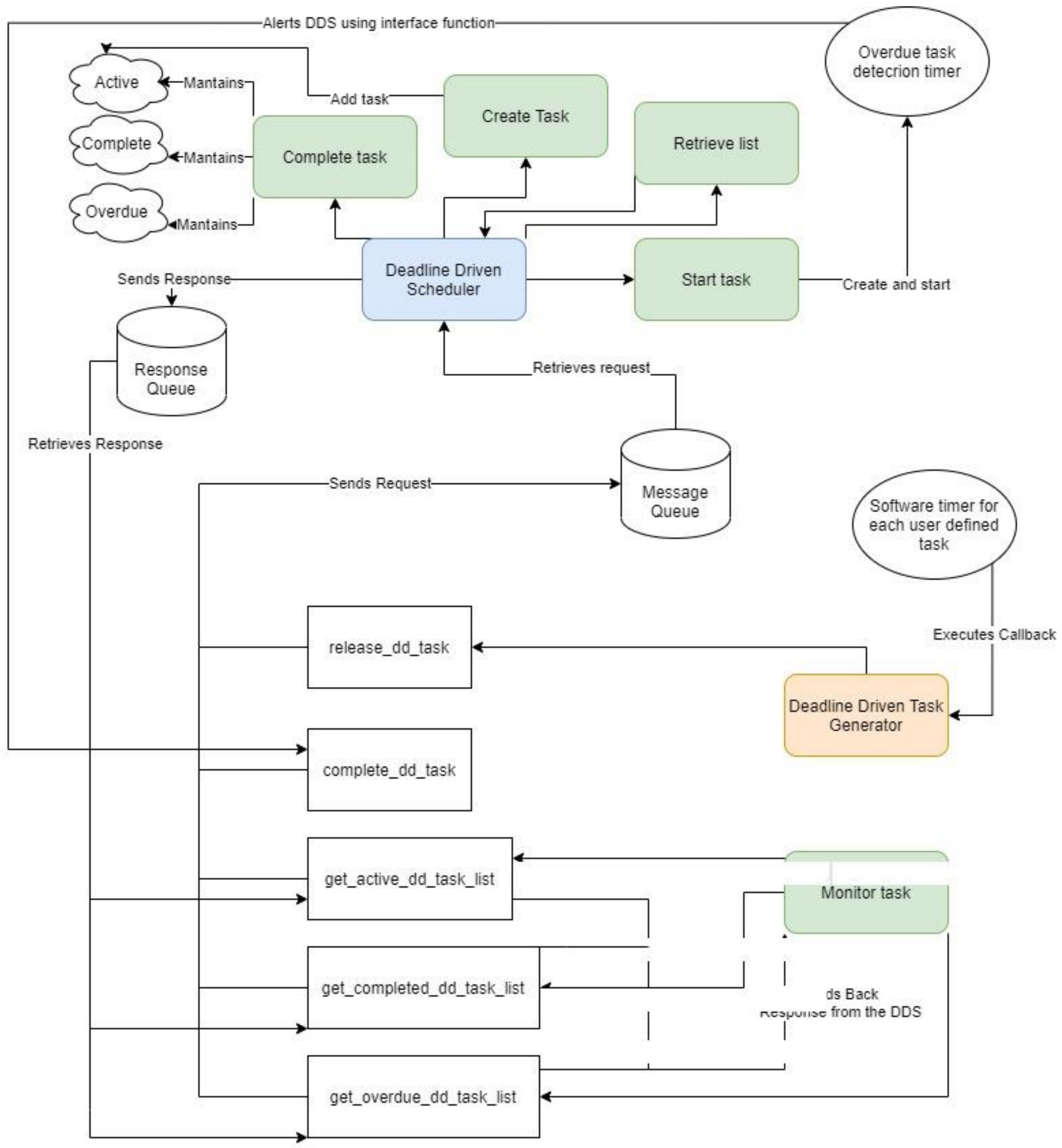


Figure 2: System Overview

The following functional requirements were to be met [1]:

- The DDS must correctly execute the three given test benches using EDF scheduling. These test benches are detailed in section 4.
- The Monitor Task must regularly report the number of active, completed, and overdue DD-Tasks. These values must be printed to the console.

The following technical requirements were to be met [1]:

- Auxiliary tasks must only interface with the DD Scheduler's via its interface functions.
- Tasks must be used to control code execution.
- Queues must be used for inter-task communications. Global variables may NOT be used for this purpose.
- Software timers must be used for generating time-based events.
- No task notifications may not be used.
- The system must accommodate aperiodic tasks. However, aperiodic tasks will not be tested or demonstrated.

3.2. DD Tasks and Task Lists

A DD-Task is a data structure. This data structure holds the handle (pointer) of a corresponding user-defined F-Task. The data structure also holds the task type, task id, the release time, the absolute deadline, and lastly the completion time.

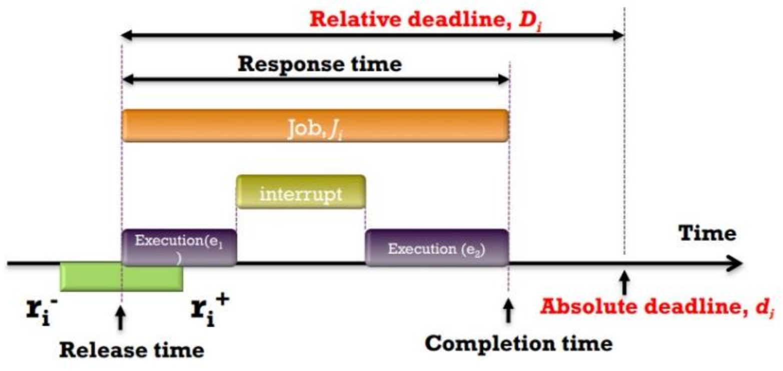


Figure 3: Real time task definitions

As shown above in figure 3, the release time is the instant when the task is released, the completion time is the instant the task is completed/killed, and the absolute deadline is the instant at which the execution must be finished. The execution time of the DD-Task not known to the DDS.

The DDS maintains three lists:
1. Active Task List
A list of DD-Tasks that needs to schedule. This list is sorted when tasks are added or removed.
2. Completed Task List
A list of DD-Tasks which have completed execution before their deadlines.
3. Overdue Task List
A list of DD-Tasks which have missed their deadlines.

The lists are implemented using the below linked list structure.

```c
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 node{
    dd_task* task;
    struct node* next;
} node;
```

To create a list element the function below is used.

```c
node* createNode(dd_task* task){
    node* newNode = (node*)auto_malloc(sizeof(node));
    newNode->task = task;
    newNode->next = NULL;
    return newNode;
}
```

The below processes for the DDS to maintain the list are also shown below.

- Adding a task
    1. Find the location the nodes deadline belongs in (earliest deadline first).
    2. Rearrange pointers in the linked list to insert node.

```c
bool addToTheSortedList(node** list, dd_task* task){
    task->release_time = getAbsoluteTime();
    node* newNode = createNode(task);
    node* listNode = *list;
    if(*list == NULL){
        *list = newNode;
        return true;
    }
    if(listNode->task->absolute_deadline > task->absolute_deadline){
        newNode->next = listNode;
        *list = newNode;
        return true;
    }
    while (listNode->next != NULL){
        if(listNode->next->task->absolute_deadline > task->absolute_deadline){
            newNode->next = listNode->next;
            listNode->next = newNode;
            return false;
        }
        listNode = listNode->next;
    }
    listNode->next = newNode;
    return false;
}
```

- Removing a completed task
  1. Locate the first task with the corresponding task id.
  2. Remove the node.

```
node* removeNode(node** list, int task_id){
    node* removedNode = NULL;
    if((*list)->task->task_id == task_id){
        removedNode = *list;
        vTaskSuspend((*list)->task->t_handle);
        *list = (*list)->next;
        removedNode->task->completion_time = getAbsoluteTime();
        return removedNode;
    }
    node* listNode = *list;
    while (listNode->next != NULL) {
        if(listNode->next->task->task_id == task_id){
            removedNode = listNode->next;
            vTaskSuspend(removedNode->task->t_handle);
            listNode->next = listNode->next->next;
            removedNode->task->completion_time = getAbsoluteTime();
            return removedNode;
        }
        listNode = listNode->next;
    }
    removedNode->task->completion_time = getAbsoluteTime();
    return removedNode;
}
```

  3. Add the removed node to the end of the completed list.

```
void addToTheEndOfTheList(node** list, node* newNode){
    newNode->next = NULL;
    node* listNode = *list;
    if(*list == NULL){
        *list = newNode;
        return;
    }
    while (listNode->next != NULL){
        listNode = listNode->next;
    }
    listNode->next = newNode;
}
```

- Removing an overdue task
  1. Locate the first task with the corresponding task id.
  2. Remove the node.

```
node* removeNode(node** list, int task_id){
    node* removedNode = NULL;
    if((*list)->task->task_id == task_id){
        removedNode = *list;
        vTaskSuspend((*list)->task->t_handle);
        *list = (*list)->next;
        removedNode->task->completion_time = getAbsoluteTime();
        return removedNode;
    }
    node* listNode = *list;
    while (listNode->next != NULL) {
        if(listNode->next->task->task_id == task_id){
            removedNode = listNode->next;
            vTaskSuspend(removedNode->task->t_handle);
            listNode->next = listNode->next->next;
            removedNode->task->completion_time = getAbsoluteTime();
            return removedNode;
        }
        listNode = listNode->next;
    }
    removedNode->task->completion_time = getAbsoluteTime();
    return removedNode;
}
```

  3. Add the removed node to the end of the overdue list.

```
void addToTheEndOfTheList(node** list, node* newNode){
    newNode->next = NULL;
    node* listNode = *list;
    if(*list == NULL){
        *list = newNode;
        return;
    }
    while (listNode->next != NULL){
        listNode = listNode->next;
    }
    listNode->next = newNode;
}
```

- Starting a task

1. Take the node at the head of the active list and ensure it's deadline hasn't already passed.
2. If the deadline hasn't passed:
    a. Start a timer to go of at the moment the task must be done.
    b. Remove the suspension of the task to allow it to get processer time.
3. If the deadline has passed:
    a. Remove the node at the head of the list and try starting the next node at the head if such node exists.

```
// DDS scheduler helper
void startTask(bool *isRunning, int *taskRunning){
    if(activeListHead == NULL){
        return;
    }
    // Create timer to stop overdue tasks
    if(activeListHead->task->absolute_deadline - getAbsoluteTime() > 0){
        if(overdueDetectionTimer == NULL){
            overdueDetectionTimer = xTimerCreate("overdueTaskDetection", pdMS_TO_TICKS
(1 + (activeListHead->task-
>absolute_deadline) - getAbsoluteTime()), pdFALSE, (void*)activeListHead->task-
>task_id, taskIsOverdue);
        } else {
            vTimerSetTimerID(overdueDetectionTimer, (void*)activeListHead->task-
>task_id);
            xTimerChangePeriod(overdueDetectionTimer, pdMS_TO_TICKS (1 + (activeListHea
d->task->absolute_deadline) - getAbsoluteTime()), 0);
        }
        vTaskResume(activeListHead->task->t_handle);
        xTimerReset(overdueDetectionTimer, 0);
        *isRunning = true;
        *taskRunning = activeListHead->task->task_id;
    } else {
        // Deadline is already passed
        // Add to the over due list and start another task if one exists
        addToTheEndOfTheList(&overdueListHead, removeNode(&activeListHead, activeListHe
ad->task->task_id));
        startTask(isRunning, taskRunning);
    }
}
```

3.3. DD Scheduler Functionality

The DDS is the highest priority F-task. This task is kept in the suspended state until one of it's interface functions are used. These interface functions are as follows:

1. release_dd_task
2. complete_dd_task
3. get_active_dd_task_list
4. get_completed_dd_task_list
5. get_overdue_dd_task_list

These interface functions are further explained in section 3.7. The auxiliary/test tasks only have access to the DDS through these interface functions. This includes the restriction of access to data structure used by the DDS.

The DDS is blocked until a message is written to its queue. Once a request/message is received the DDS services this request. The priority of the DDS is the highest so it will be the only task executed. The messages the DDS services are described below:

Message from release_dd_task

1. The DDS assigns the task struct a release time.
2. Adds the DD-Task to the Active Task List sorted by the task's deadline.
3. Start the first task in the active list by setting its priority high so it is the only task executing.

Message from complete_dd_task

1. The DDS assigns the task a completion time.
2. The task is then removed from the active list and the list must maintain it's sorted orders.
3. the task is then added to the completed task list.
4. Start the first task in the active list by setting its priority high so it is the only task executing.

Message from get_active_dd_task_list

The DDS sends the active task list to a return queue.

Message from get_completed_dd_task_list

The DDS sends the completed task list to a return queue.

Message from get_overdue_dd_task_list

The DDS sends the overdue task list to a return queue.

When starting the highest priority active task a software timer is created to issue a callback when the absolute deadline has occurred. This callback uses the complete task interface but provides it with a late flag. This task is then stopped and moved to the overdue task list. The task is stopped because these tasks are considered hard deadline ones which means the results are irrelevant after the deadline. This was the most effective internal method to check if a task has missed a deadline. This supports both aperiodic and periodic tasks. If the scheduler only executed periodic tasks, then checking if the previous task for completion could be performed. Due to need for support of both tasks the chosen approach for overdue detection was the software timer. This timer system is also not affected by task creation jitter. Jitter occurs when multiple tasks are created at the exact same time. This does not delay the detection of the overdue task as the software timer's duration is set at the time of creation and is set to execute its callback exactly at the absolute deadline of the task. Since timers are serviced right away the DDS is able to deal with these tasks in a faster manner. If the absolute deadline is already passed the task is never executed and instead directly sent to the over task list. The priority of the DDS scheduler is 2.

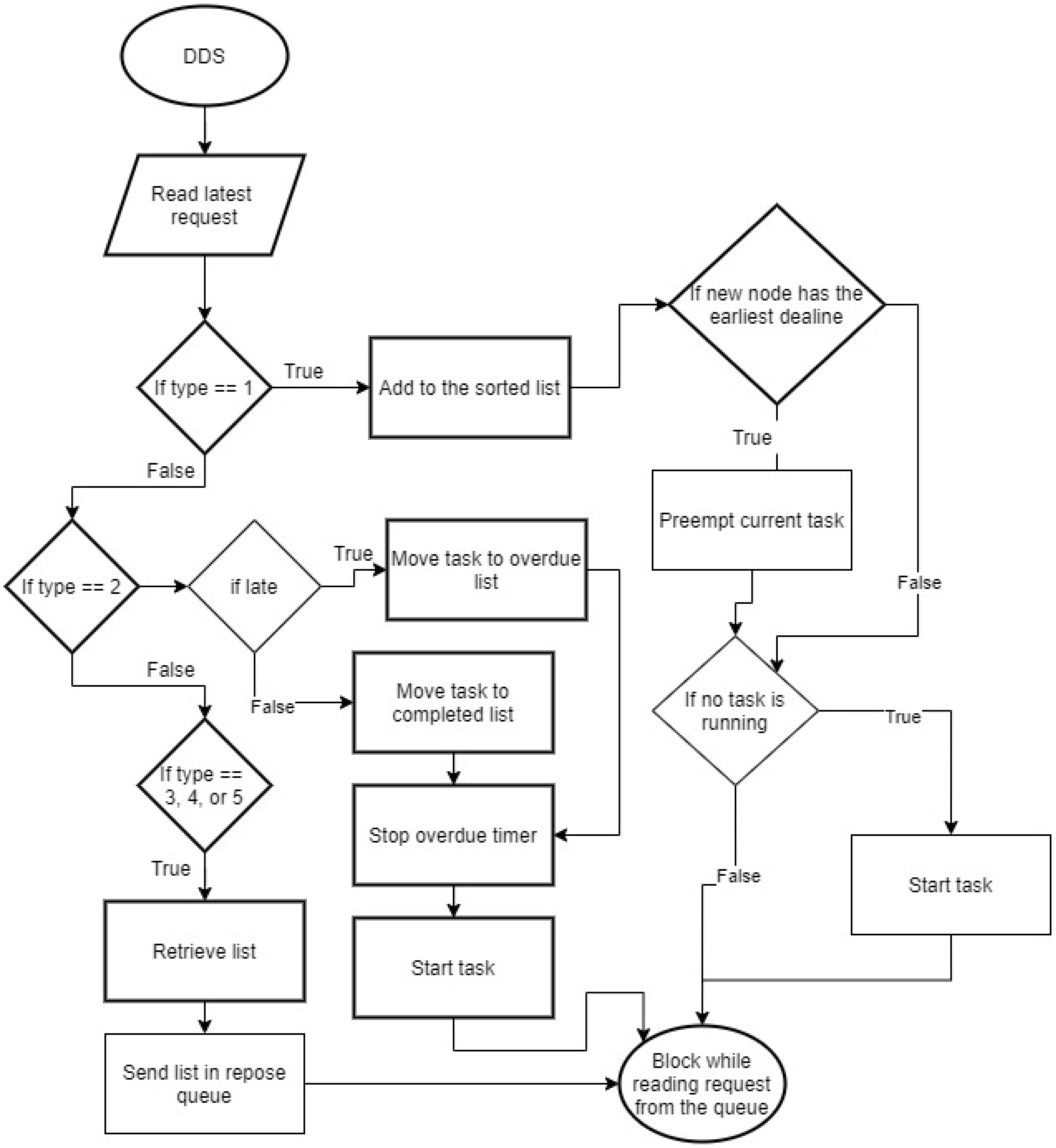


Figure 4: Flow chart of how the DD Scheduler works

The code for the scheduler processing of requests is shown below.

```
// DDS scheduler task
static void ddsSchedulerTask(void *pvParameters){
    request receivedRequest;
    response returnMessage;
    bool isRunning = false;
    int taskRunning = -1;
    while(1)
    {
        xQueueReceive(ddsQueue, &receivedRequest, pdMS_TO_TICKS(portMAX_DELAY));
        if(receivedRequest.reqType == 1){
            if(taskRunning != receivedRequest.task->task_id){
                vTaskSuspend(receivedRequest.task->t_handle);
            }
            if(addToTheSortedList(&activeListHead, receivedRequest.task)){
                if(activeListHead->next != NULL){
                    vTaskSuspend(activeListHead->next->task->t_handle);
                }
                isRunning = false;
                if(overdueDetectionTimer != NULL){
                    xTimerStop(overdueDetectionTimer, 0);
                }
            }
            if(!isRunning){
                startTask(&isRunning, &taskRunning);
            }
        } else if(receivedRequest.reqType == 2 && !receivedRequest.isOverdue){
            xTimerStop(overdueDetectionTimer, 0);
            addToTheEndOfTheList(&completedListHead, removeNode(&activeListHead, receiv
edRequest.task_id));
            startTask(&isRunning, &taskRunning);
        } else if(receivedRequest.reqType == 2 && receivedRequest.isOverdue){
            addToTheEndOfTheList(&overdueListHead, removeNode(&activeListHead, received
Request.task_id));
            startTask(&isRunning, &taskRunning);
        } else if(receivedRequest.reqType == 3){
            returnMessage.list = activeListHead;
            xQueueSend(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
        } else if(receivedRequest.reqType == 4){
            returnMessage.list = completedListHead;
            xQueueSend(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
        } else if(receivedRequest.reqType == 5){
            returnMessage.list = overdueListHead;
            xQueueSend(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
        }
    }
}
```

3.4. User Defined Tasks

In a practical context a user defined function would preform some type of operation. For testing purposes this function just implements a delay loop. This delay loop was taken from the source as described below in the code. This delay bases the number of loop executions based on the number of ticks needed to delay for a certain duration of milliseconds. For debugging use the red led is turned on for task 1, the amber light for task 2, and the green light for task 3. This is used for a indication of where the code execution is currently at. Once complete the task notifies the scheduler using the complete_dd_task interface. The execution time is passed in as a parameter when created, however, this execution time is not known to the scheduler and therefore cannot be used for calculations. The priority of the user defined function is always 2, however, when not executing it is suspended. This was done due to problem I occurred during debugging. Tasks with a priority of 0 still received processor time which cut into to accuracy of the delay code. So instead of elevating priorities the DDS uses the task unblock and block feature to manage these tasks.

```
// User defined tasks
static void userDefinedTask1(void *pvParameters){
    int executionTime = 0;
    while(1){
        executionTime = (int) pvParameters;
        GPIO_SetBits(GPIOC, 0);
        GPIO_ResetBits(GPIOC, 1);
        GPIO_ResetBits(GPIOC, 4);
        // Code for the delay counter from
        // https://stm32f4-discovery.net/2014/09/precise-delay-counter/
        // 400 used as multiplier instead of 1000, as using 1000 produced a very imprec
ise delay
        executionTime = 400 * executionTime * multiplier - 10;
        while (executionTime > 0){
            executionTime--;
        }
        complete_dd_task(1, false);
    }
}
```

The lines of code in orange above indicate which lines change based on the user defined task number. To reduce the complexity, I chose not to create a structure to hold task information meaning both the execution time and the task id had to be passed in. This was handled buy hard wiring these lines in to each user defined task uniquely. These design choices were made because the sacrifice of readability in my opinion was not worth the connivance that creating another data structure would provide.

3.5. DD Task Generator

The task generator is used for testing purposes and only creates periodic tasks for the test benches. The user defined function is unique for each task type. This eliminates the need for the task id to be passed in. The task generator, however, is shared between all tasks but a different user defined function is created for each task based on its id. Aperiodic tasks are supported the only difference being they are only created once. For periodic tasks a delay is used to execute a new tasks creation, the delay is proportional to the task's period. This period is one of the many parameters passed in using the task generator struct as shown below.

The release dd_task is then continually called when the delay is finished. A single generator was used because the only change between the task types was the generation of a different user defined task handles. The user defined task is also passed with the execution time which is never accessed by the scheduler.

Below the code is shown to create the defined test bench and start a software timer to use the task generator as a callback. The task generator then uses the saved information for the corresponding to the task to use the release task interface.

```
    // Execute test bench
    if(TEST_BENCH == 1){
        generator_information task1 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask1, "userDefinedTask1", configMINIMAL_STACK_SIZE, (vo
id*)95, 2, &task1.t_handle);
        taskInformation1 = task1;
        Task1Generator = xTimerCreate("ddsTask1Generator", pdMS_TO_TICKS(500), pdTRUE,
(void*)1, ddsTaskGenerator);
        generator_information task2 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask2, "userDefinedTask2", configMINIMAL_STACK_SIZE, (vo
id*)150, 2, &task2.t_handle);
        taskInformation2 = task2;
        Task2Generator = xTimerCreate("ddsTask2Generator", pdMS_TO_TICKS(500), pdTRUE,
(void*)2, ddsTaskGenerator);
        generator_information task3 = {NULL, 0, 750};
        xTaskCreate(userDefinedTask3, "userDefinedTask3", configMINIMAL_STACK_SIZE, (vo
id*)250, 2, &task3.t_handle);
        taskInformation3 = task3;
        Task3Generator = xTimerCreate("ddsTask3Generator", pdMS_TO_TICKS(750), pdTRUE,
(void*)3, ddsTaskGenerator);
    } else if(TEST_BENCH == 2){
        generator_information task1 = {NULL, 0, 250};
        xTaskCreate(userDefinedTask1, "userDefinedTask1", configMINIMAL_STACK_SIZE, (vo
id*)95, 2, &task1.t_handle);
        taskInformation1 = task1;
        Task1Generator =xTimerCreate("ddsTask1Generator", pdMS_TO_TICKS(250), pdTRUE, (
void*)1, ddsTaskGenerator);
        generator_information task2 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask2, "userDefinedTask2", configMINIMAL_STACK_SIZE, (vo
id*)150, 2, &task2.t_handle);
        taskInformation2 = task2;
        Task2Generator = xTimerCreate("ddsTask2Generator", pdMS_TO_TICKS(500), pdTRUE,
(void*)2, ddsTaskGenerator);
        generator_information task3 = {NULL, 0, 750};
        xTaskCreate(userDefinedTask3, "userDefinedTask3", configMINIMAL_STACK_SIZE, (vo
id*)250, 2, &task3.t_handle);
```

```
        taskInformation3 = task3;
        Task3Generator = xTimerCreate("ddsTask3Generator", pdMS_TO_TICKS(750), pdTRUE,
(void*)3, ddsTaskGenerator);
    } else if(TEST_BENCH == 3){
        generator_information task1 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask1, "userDefinedTask1", configMINIMAL_STACK_SIZE, (vo
id*)100, 2, &task1.t_handle);
        taskInformation1 = task1;
        Task1Generator = xTimerCreate("ddsTask1Generator", pdMS_TO_TICKS(500), pdTRUE,
(void*)1, ddsTaskGenerator);
        generator_information task2 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask2, "userDefinedTask2", configMINIMAL_STACK_SIZE, (vo
id*)200, 2, &task2.t_handle);
        taskInformation2 = task2;
        Task2Generator = xTimerCreate("ddsTask2Generator", pdMS_TO_TICKS(500), pdTRUE,
(void*)2, ddsTaskGenerator);
        generator_information task3 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask3, "userDefinedTask3", configMINIMAL_STACK_SIZE, (vo
id*)200, 2, &task3.t_handle);
        taskInformation3 = task3;
        Task3Generator = xTimerCreate("ddsTask3Generator", pdMS_TO_TICKS(500), pdTRUE,
(void*)3, ddsTaskGenerator);
    }
    // Start the initial tasks for time 0
    ddsTaskGenerator(Task1Generator);
    ddsTaskGenerator(Task2Generator);
    ddsTaskGenerator(Task3Generator);
    // Start the task generators
    xTimerStart(Task1Generator, 0);
    xTimerStart(Task2Generator, 0);
    xTimerStart(Task3Generator, 0);
\
```

A new task handle is created for each new task id. This is done as the user defined tasks have different execution times and it was complicated when passing in both the task id and execution time to have a shared user defined task. That means a task handle is created and deleted every time a DD-Task is released and completed. The DDS generator is created as a callback function for a software timer. This allows it to release a task based on the calling timers id. That indicates the type of task to be used.

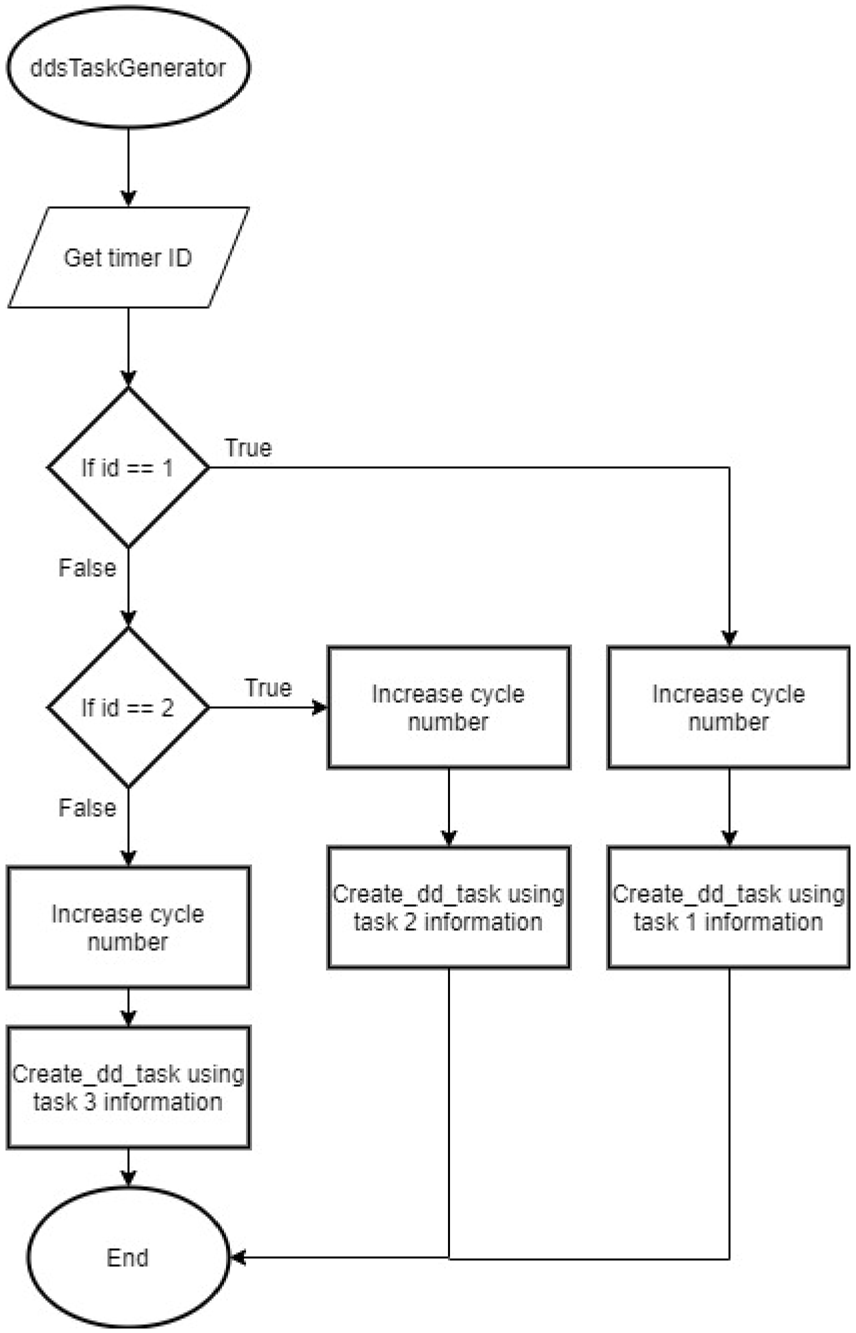


Figure 5: Flow chart of how the Deadline-Task Generator works

```
// Deadline driven task generator
static void ddsTaskGenerator(TimerHandle_t xTimer){
    uint32_t task_id = (uint32_t)pvTimerGetTimerID(xTimer);
    if(task_id == 1){
        taskInformation1.cycleNum += 1;
        create_dd_task(taskInformation1.t_handle, PERIODIC, task_id, taskInformation1.p
eriod * taskInformation1.cycleNum);
    } else if(task_id == 2){
        taskInformation2.cycleNum += 1;
        create_dd_task(taskInformation2.t_handle, PERIODIC, task_id, taskInformation2.p
eriod * taskInformation2.cycleNum);
    } else if(task_id == 3){
        taskInformation3.cycleNum += 1;
        create_dd_task(taskInformation3.t_handle, PERIODIC, task_id, taskInformation3.p
eriod * taskInformation3.cycleNum);
    }
}
```

3.6. Monitor Task

The only task used to report the results of the scheduler during testing is the monitor task.This tasks uses the get_active_dd_task_list, get_complete_dd_task_list, and get_overdue_dd_task_list interface functions to access the scheduler recorded task information. The monitor task then prints out the number of active DD-tasks, the number of completed DD-tasks, and the number of overdue DD-tasks. The monitor task is implemented as a software timer callback. This allows for reporting even if a active task is executing. Using the test bench information, the monitor task executes every 500 ms this was chosen as it is a multiple of the hyper period and was needed for information presented in the system evaluation section. This also reduces the overhead introduced by the monitoring task. The priority of the monitor task function is 3 since it is a software interrupt.

```
// Monitor task
static void monitorTask(void *pvParameters){
    printf("Active list length: %d\n", getListLength(get_active_dd_task_list()));
    printf("Completed list length: %d\n", getListLength(get_complete_dd_task_list()));
    printf("Overdue list length: %d\n\n", getListLength(get_overdue_dd_task_list()));
}
```

3.7. Interface Functions

The auxiliary/test tasks only have access to the DDS through these interface functions. A request struct is defined as the following. The struct and flag may or may not be used based on the type of the request. The types are noted below.

1- release_dd_task
2- complete_dd_task
3- get_active_dd_task_list
4- get_completed_dd_task_list
5- get_overdue_dd_task_list

A response struct is used in the case of getting a list to return the queried list from the DDS. This holds the head of the list and is sent to the response queue.

```
// DDS queue and request data structure
typedef struct request {
    int reqType;
    dd_task* task;
    uint32_t task_id;
    bool isOverdue;
} request;
xQueueHandle ddsQueue;

// DDS response queue and data structure
typedef struct response {
    node* list;
} response;
xQueueHandle responseQueue;
```

3.7.1. release_dd_task

This interface receives the information that is used to create a new task. That is the interface receives the user defined task handle/pointer to the function, the task type (aperiodic or periodic), the task id, and the absolute deadline. This information is then used to create a new struct to represent the task. The release time and the completion time are not given as those are added by the DDS as described in the DDS scheduler section. The struct is then sent as a message to a queue, this message is then received by the DDS. The original function signature is given below.

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

The signature and the code below was used. There were no changes that occurred in the function declaration. However, create was used instead of release when writing the code. This was intended to help my comprehension of the code.

```
void create_dd_task(TaskHandle_t t_handle, task_type type, uint32_t task_id, uint32_t absolute_deadline){
    request newRequest;
    dd_task* newTask = (dd_task*)auto_malloc(sizeof(dd_task));
    newTask->type = type;
    newTask->t_handle = t_handle;
    newTask->task_id = task_id;
    newTask->absolute_deadline = absolute_deadline;
    newRequest.task = newTask;
    newRequest.reqType = 1;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
}
```

3.7.2. complete_dd_task

This interface receives the Id of the task that has been completed and a late status flag. If the late flag is set to true, the task will be stopped and moved to the overdue task list. If the late flag is not set to true, the task will be added to completed task list. The list is transversed till a task with the respective ID is found. Both the flag and the ID are packaged in a struct and sent to a queue, this message is then received by the DDS.

The original function signature is given below.

void delete_dd_task(uint32_t task_id);

The signature was then changed to the below signature. This was done as late tasks would have an overdue timer that would occur and needed to complete a task. To simplify the understand of the requests a late flag indicates a the overdue timer is attempting to complete a task. Complete was also used instead of delete to enhance my comprehension of the code.

```
void complete_dd_task(uint32_t task_id, bool isOverdue){
    request newRequest;
    newRequest.reqType = 2;
    newRequest.task_id = task_id;
    newRequest.isOverdue = isOverdue;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
}
```

3.7.3. get_active_dd_task_list

The interface requests the active task list from the DDS. This is requested by setting the request value to 3. The struct is then packaged and sent to a queue, this message is then received by the DDS. Once the DDS receives the request it returns the list. The request returns the list by value. This is done because the monitor task which uses this interface should not be able to modify the original list by getting a reference to the head of the list. This is a design decision for simplicity during debugging.

The original function signature is given below.

**dd_task_list get_active_dd_task_list(void);

The signature and the code below was used. There were no changes that occurred in the function declaration.

```
node* get_active_dd_task_list(void){
    request newRequest;
    newRequest.reqType = 3;
    response returnMessage;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
    xQueueReceive(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
    return returnMessage.list;
}
```

3.7.4. get_completed_dd_task_list

The interface requests the completed task list from the DDS. This is requested by setting the request value to 4. The struct is then packaged and sent to a queue, this message is then received by the DDS. Once the DDS receives the request it returns the list. The request returns the list by value. This is done because the monitor task which uses this interface should not be able to modify the original list by getting a reference to the head of the list. This is a design decision for simplicity during debugging.

The original function signature is given below.

**dd_task_list get_complete_dd_task_list(void);

The signature and the code below was used. There were no changes that occurred in the function declaration.

```
node* get_complete_dd_task_list(void){
    request newRequest;
    newRequest.reqType = 4;
    response returnMessage;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
    xQueueReceive(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
    return returnMessage.list;
}
```

3.7.5. get_overdue_dd_task_list

The interface requests the overude task list from the DDS. This is requested by setting the request value to 5. The struct is then packaged and sent to a queue, this message is then received by the DDS. Once the DDS receives the request it returns the list. The request returns the list by value. This is done because the monitor task which uses this interface should not be able to modify the original list by getting a reference to the head of the list. This is a design decision for simplicity during debugging.

The original function signature is given below.

**dd_task_list get_overdue_dd_task_list(void);

The signature and the code below was used. There were no changes that occurred in the function declaration.

```
node* get_overdue_dd_task_list(void){
    request newRequest;
    newRequest.reqType = 5;
    response returnMessage;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
    xQueueReceive(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
    return returnMessage.list;
}
```

3.8. Task Sorting Algorithm

Task sorting is handled in two different ways. When adding a task, the task is added in a sorted order by it’s absolute deadline. This guarantees when removed the list is still sorted. Due to this fact the removal of a node requires the removal of a node and modifying the next list element of the previous node.

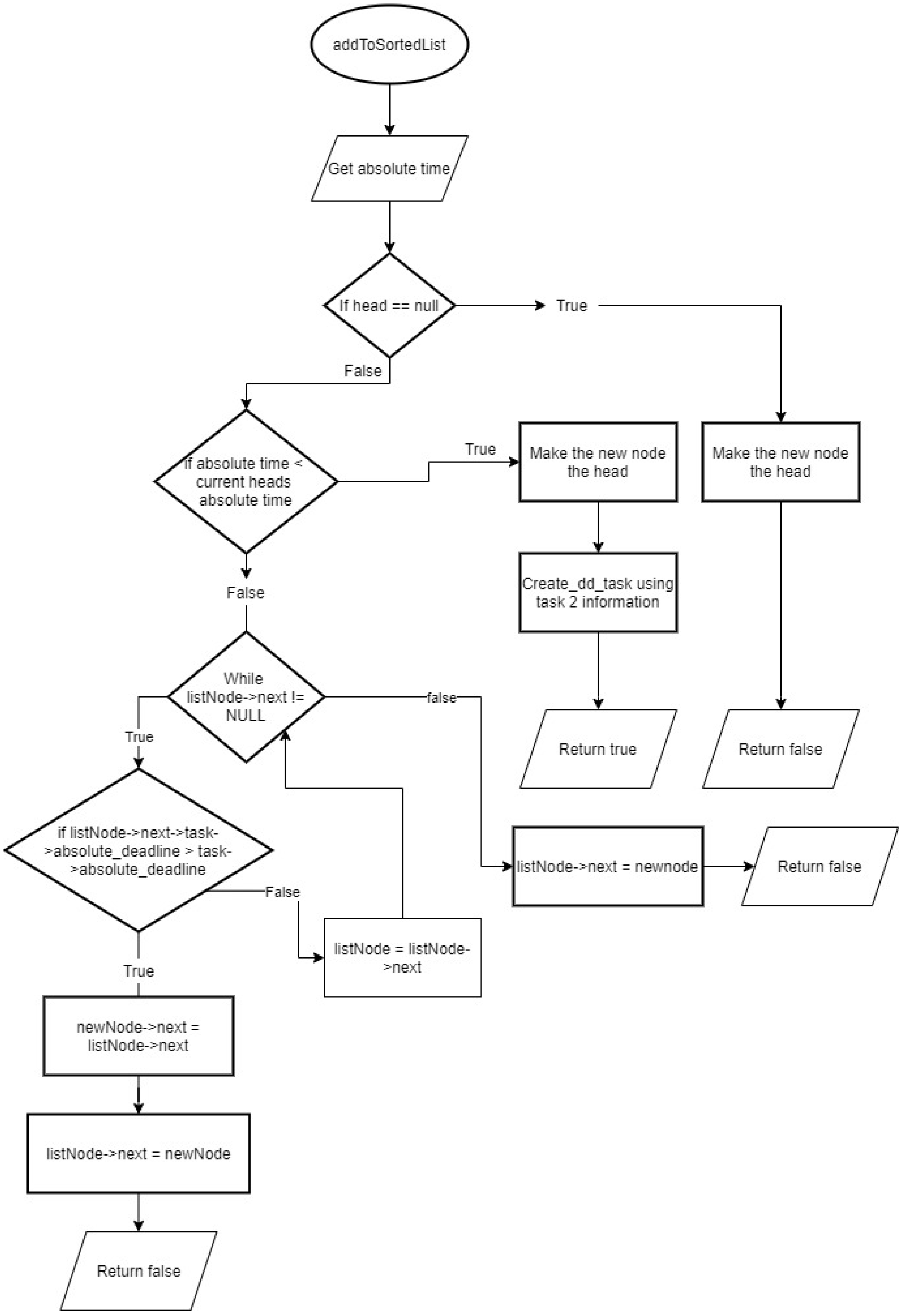

addToSortedList
Get absolute time
If head == null
True
False
Make the new node the head
Return false
if absolute time < current heads absolute time
True
Make the new node the head
Create_dd_task using task 2 information
Return true
False
While listNode->next != NULL
false
listNode->next = newnode
Return false
True
if listNode->next->task->absolute_deadline > task->absolute_deadline
False
listNode = listNode->next
True
newNode->next = listNode->next
listNode->next = newNode
Return false

Figure 6: Flow chart of the DD-Task sorting algorithm for adding a list element

The code is shown below to add a node to a sorted list.

```
bool addToTheSortedList(node** list, dd_task* task){
    task->release_time = getAbsoluteTime();
    node* newNode = createNode(task);
    node* listNode = *list;
    if(*list == NULL){
        *list = newNode;
        return true;
    }
    if(listNode->task->absolute_deadline > task->absolute_deadline){
        newNode->next = listNode;
        *list = newNode;
        return true;
    }
    while (listNode->next != NULL){
        if(listNode->next->task->absolute_deadline > task->absolute_deadline){
            newNode->next = listNode->next;
            listNode->next = newNode;
            return false;
        }
        listNode = listNode->next;
    }
    listNode->next = newNode;
    return false;
}
```

Figure : Flow chart of the DD-Task sorting algorithm for removing a list element

The code is shown below to remove a node from a sorted list.

```
node* removeNode(node** list, int task_id){
    node* removedNode = NULL;
    if((*list)->task->task_id == task_id){
        removedNode = *list;
        vTaskSuspend((*list)->task->t_handle);
        *list = (*list)->next;
        removedNode->task->completion_time = getAbsoluteTime();
        return removedNode;
    }
    node* listNode = *list;
    while (listNode->next != NULL) {
        if(listNode->next->task->task_id == task_id){
            removedNode = listNode->next;
            vTaskSuspend(removedNode->task->t_handle);
            listNode->next = listNode->next->next;
            removedNode->task->completion_time = getAbsoluteTime();
            return removedNode;
        }
        listNode = listNode->next;
    }
    removedNode->task->completion_time = getAbsoluteTime();
    return removedNode;
}
```

4. System Evaluation

Three test benches were provided to test the DDS system, these test benches are shown 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

Table 1: DDS Test Benches

The event table of test bench one is shown below. This compares the perfect expected execution times to the times that were actually produced by the scheduler.

Event #	Event	Measured Time (ms)	Expected Time (ms)
1	Task 1 released	0	0
2	Task 2 released	0	0
3	Task 3 released	0	0
4	Task 1 complete	95	95
5	Task 2 complete	245	245
6	Task 3 complete	495	495
7	Task 1 released	500	500
8	Task 2 released	500	500
9	Task 1 complete	595	595
10	Task 2 complete	745	745
11	Task 3 released	750	750
12	Task 3 complete	1000	1000
13	Task 1 released	1000	1000
14	Task 2 released	1000	1000
15	Task 1 complete	1095	1095
16	Task 2 complete	1245	1245
17	Task 1 released	1500	1500
18	Task 2 released	1500	1500
19	Task 3 released	1500	1500

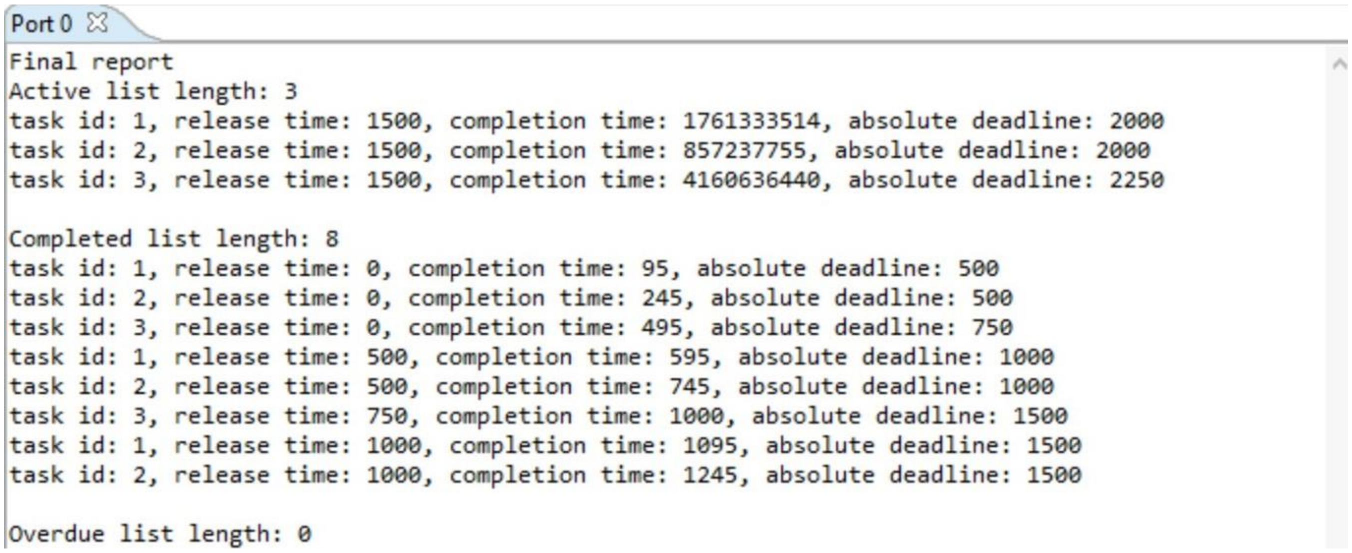
Table 2: Table of DDS events generated for Test Bench #1

The table below shows the number of elements in each list. This compares the perfect expected execution to the execution produced by the scheduler.

Type	Measured	Expected
Number of active DD-Tasks	3	3
Number of completed DD-Tasks	10	10
Number of overdue DD-Tasks	1	1

Table 3: Table of Monitor Task outputs for Test Bench #2.

Reports of all three test benches were taken from the lab system and shown below. These images detail all three lists maintained by the scheduler to give a accurate picture of the schedulers performance and execution.

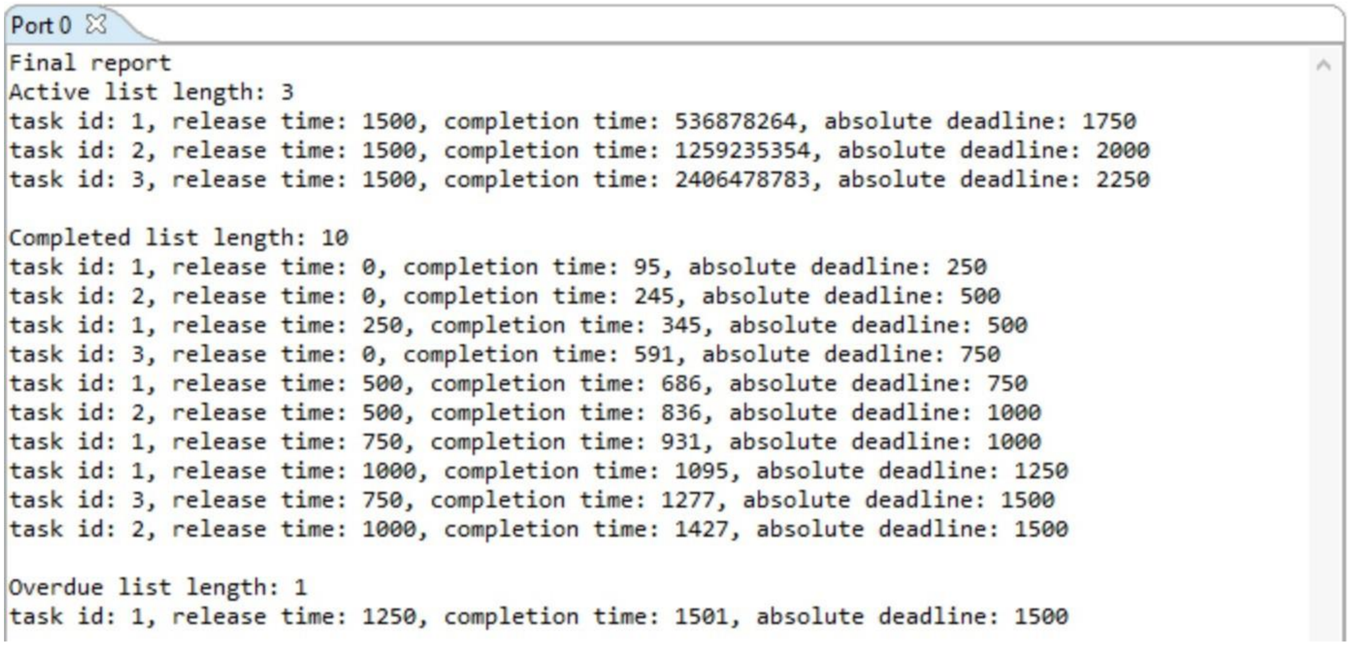


```
Final report
Active list length: 3
task id: 1, release time: 1500, completion time: 1761333514, absolute deadline: 2000
task id: 2, release time: 1500, completion time: 857237755, absolute deadline: 2000
task id: 3, release time: 1500, completion time: 4160636440, absolute deadline: 2250

Completed list length: 8
task id: 1, release time: 0, completion time: 95, absolute deadline: 500
task id: 2, release time: 0, completion time: 245, absolute deadline: 500
task id: 3, release time: 0, completion time: 495, absolute deadline: 750
task id: 1, release time: 500, completion time: 595, absolute deadline: 1000
task id: 2, release time: 500, completion time: 745, absolute deadline: 1000
task id: 3, release time: 750, completion time: 1000, absolute deadline: 1500
task id: 1, release time: 1000, completion time: 1095, absolute deadline: 1500
task id: 2, release time: 1000, completion time: 1245, absolute deadline: 1500

Overdue list length: 0
```

Figure 7: Detailed list for test bench 1

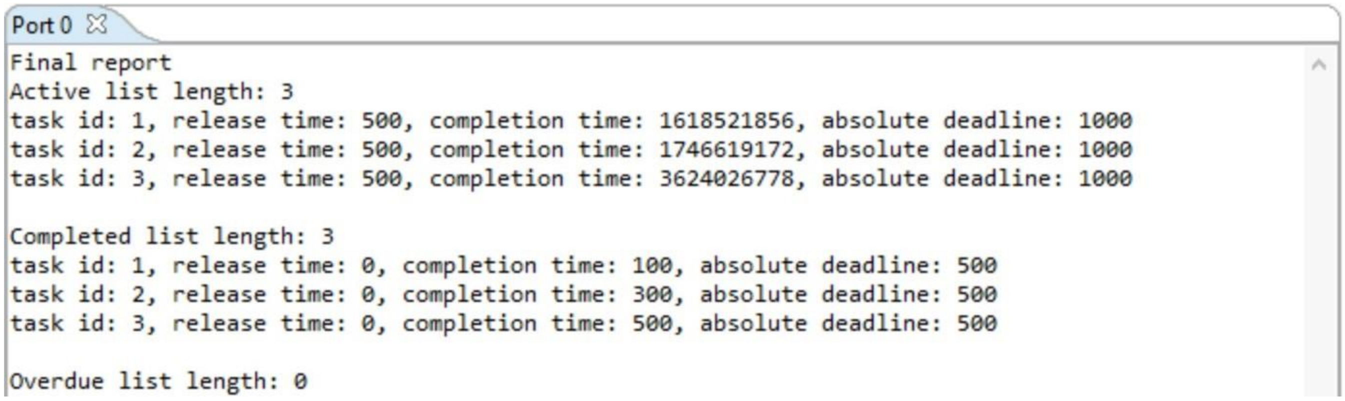


```
Final report
Active list length: 3
task id: 1, release time: 1500, completion time: 536878264, absolute deadline: 1750
task id: 2, release time: 1500, completion time: 1259235354, absolute deadline: 2000
task id: 3, release time: 1500, completion time: 2406478783, absolute deadline: 2250

Completed list length: 10
task id: 1, release time: 0, completion time: 95, absolute deadline: 250
task id: 2, release time: 0, completion time: 245, absolute deadline: 500
task id: 1, release time: 250, completion time: 345, absolute deadline: 500
task id: 3, release time: 0, completion time: 591, absolute deadline: 750
task id: 1, release time: 500, completion time: 686, absolute deadline: 750
task id: 2, release time: 500, completion time: 836, absolute deadline: 1000
task id: 1, release time: 750, completion time: 931, absolute deadline: 1000
task id: 1, release time: 1000, completion time: 1095, absolute deadline: 1250
task id: 3, release time: 750, completion time: 1277, absolute deadline: 1500
task id: 2, release time: 1000, completion time: 1427, absolute deadline: 1500

Overdue list length: 1
task id: 1, release time: 1250, completion time: 1501, absolute deadline: 1500
```

Figure 8: Detailed list for test bench 2



Port 0

```
Final report
Active list length: 3
task id: 1, release time: 500, completion time: 1618521856, absolute deadline: 1000
task id: 2, release time: 500, completion time: 1746619172, absolute deadline: 1000
task id: 3, release time: 500, completion time: 3624026778, absolute deadline: 1000

Completed list length: 3
task id: 1, release time: 0, completion time: 100, absolute deadline: 500
task id: 2, release time: 0, completion time: 300, absolute deadline: 500
task id: 3, release time: 0, completion time: 500, absolute deadline: 500

Overdue list length: 0
```

Figure 9: Detailed list for test bench 3

5. DDS scheduler performance

The above tables show how the DDS scheduler I implemented functions. After evaluating the first test bench it can be seen that the DDS always produced the same release and completion times that were expected. It shows that the user defined tasks were designed accurately as there were no discrepancies between task execution time.

After evaluating the second test bench it can be seen that the DDS created the tasks and managed them as expected. This test bench was intended to test preemption of a task. The preemption worked correctly as the current executing task was preempted when one with a earlier deadline is created.

After evaluating the third test bench it can be seen that the DDS does not suffer from jitter. Jitter is the delay in task creation events due to multiple tasks being created at the same time. This can be seen due to the tight nature of the third test bench. Regardless of the no slack time the scheduler executes efficiently.

The scheduler was very effective and met the requirements given in the project manual.

6. Summary

Overall, this project worked successfully. The demonstration of the program was a success and met the requirements. This project was able to allow me to get hands on experience with tasks, queues, and software timers. These FreeRTOS features are extremely important in real time systems and this project was able to demonstrate that. This project also allowed me to understand the benefits and challenges of using an EDF scheduling algorithm. Both projects assigned in this course were excellently designed and well managed given the challenges faced throughout this semester.

7. References

[1] LAB MANUAL ECE 455: REAL TIME COMPUTER SYSTEMS DESIGN PROJECT, 1st ed. Victoria: University of Victoria, 2019.

8. Appendix: Project Code

```c
// Standard includes
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <math.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 "../Libraries/STM32F4xx_StdPeriph_Driver/inc/stm32f4xx_gpio.h"

// Controls the test bench execute
// The hyper period value at which the test should be run until
#define TEST_BENCH 1
#define REPORTING_PERIOD 1510

// Multiplier used in delay counter
uint32_t multiplier;

// Timer used for overdue task detection
xTimerHandle overdueDetectionTimer;

// Timers used for monitor task and task generators
xTimerHandle monitorTimer;
xTimerHandle Task1Generator;
xTimerHandle Task2Generator;
xTimerHandle Task3Generator;

// Linked list nodes and task representation data structures
typedef enum task_type {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 node{
    dd_task* task;
    struct node* next;
```

```c
} node;

// Information needed by the task generator based on the test bench
typedef struct generator_information {
    TaskHandle_t t_handle;
    int cycleNum;
    uint32_t period;
} generator_information;

// Stores information for the each test bench
generator_information taskInformation1;
generator_information taskInformation2;
generator_information taskInformation3;

// DDS queue and request data structure
typedef struct request {
    int reqType;
    dd_task* task;
    uint32_t task_id;
    bool isOverdue;
} request;
xQueueHandle ddsQueue;

// DDS response queue and data structure
typedef struct response {
    node* list;
} response;
xQueueHandle responseQueue;

// Scheduler list heads, only accessible by the scheduler
node* activeListHead = NULL;
node* completedListHead = NULL;
node* overdueListHead = NULL;

// Delay counter initialization
void TM_Delay_Init(void) {
    // Code for the delay counter from
    // https://stm32f4-discovery.net/2014/09/precise-delay-counter/
    RCC_ClocksTypeDef RCC_Clocks;
    /* Get system clocks */
    RCC_GetClocksFreq(&RCC_Clocks);
    /* While loop takes 4 cycles */
    /* For 1 us delay, we need to divide with 4M */
    multiplier = RCC_Clocks.HCLK_Frequency / 4000000;
}

// Initialize GPIOs for LED's
void initGPIO(void){
    // Enable GPIOC clock
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    // Initialize GPIO for the traffic light
    GPIO_InitTypeDef TrafficLight_GPIO_Struct;
    // Using pins 0, 1, and 2
    // Red, amber, and green respectively
    TrafficLight_GPIO_Struct.GPIO_Pin = 0b111;
    TrafficLight_GPIO_Struct.GPIO_Mode = GPIO_Mode_OUT;
    TrafficLight_GPIO_Struct.GPIO_OType = GPIO_OType_PP;
    TrafficLight_GPIO_Struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC , &TrafficLight_GPIO_Struct);
}

// Get the current absolute time
int getAbsoluteTime(){
    int absoluteTime = xTaskGetTickCount()/portTICK_PERIOD_MS;
    return absoluteTime;
}

// Linked list functions
void *auto_malloc(size_t size){
    // Execute a safe malloc operation
    void *ptr = NULL;
    ptr = malloc(size);
    if (ptr == NULL) {
        printf("Malloc failed");
        exit(1);
    }
    return ptr;
}

void freeList(node* list){
   node* tmp;
   while (list != NULL){
       tmp = list;
       list = list->next;
       free(tmp->task);
       free(tmp);
    }
}

void freeAllLists(){
    freeList(activeListHead);
    freeList(completedListHead);
    freeList(overdueListHead);
}

node* createNode(dd_task* task){
    node* newNode = (node*)auto_malloc(sizeof(node));
    newNode->task = task;
    newNode->next = NULL;
    return newNode;
```

```
}

void printList(node* list){
    while (list != NULL) {
        printf("task id: %u, release time: %u, completion time: %u, absolute deadline: %u\n"
, list->task->task_id, list->task->release_time, list->task->completion_time, list->task-
>absolute_deadline);
        list = list->next;
    }
}

int getListLength(node* list){
    int length = 0;
    while (list != NULL) {
        length+=1;
        list = list->next;
    }
    return length;
}

bool addToTheSortedList(node** list, dd_task* task){
    task->release_time = getAbsoluteTime();
    node* newNode = createNode(task);
    node* listNode = *list;
    if(*list == NULL){
        *list = newNode;
        return true;
    }
    if(listNode->task->absolute_deadline > task->absolute_deadline){
        newNode->next = listNode;
        *list = newNode;
        return true;
    }
    while (listNode->next != NULL){
        if(listNode->next->task->absolute_deadline > task->absolute_deadline){
            newNode->next = listNode->next;
            listNode->next = newNode;
            return false;
        }
        listNode = listNode->next;
    }
    listNode->next = newNode;
    return false;
}

void addToTheEndOfTheList(node** list, node* newNode){
    newNode->next = NULL;
    node* listNode = *list;
    if(*list == NULL){
        *list = newNode;
```

```c
        return;
    }
    while (listNode->next != NULL){
        listNode = listNode->next;
    }
    listNode->next = newNode;
}

node* removeNode(node** list, int task_id){
    node* removedNode = NULL;
    if((*list)->task->task_id == task_id){
        removedNode = *list;
        vTaskSuspend((*list)->task->t_handle);
        *list = (*list)->next;
        removedNode->task->completion_time = getAbsoluteTime();
        return removedNode;
    }
    node* listNode = *list;
    while (listNode->next != NULL) {
        if(listNode->next->task->task_id == task_id){
            removedNode = listNode->next;
            vTaskSuspend(removedNode->task->t_handle);
            listNode->next = listNode->next->next;
            removedNode->task->completion_time = getAbsoluteTime();
            return removedNode;
        }
        listNode = listNode->next;
    }
    removedNode->task->completion_time = getAbsoluteTime();
    return removedNode;
}

// Scheduler interface functions which are external to the scheduler
// These interfaces send different requests to the scheduler
void create_dd_task(TaskHandle_t t_handle, task_type type, uint32_t task_id, uint32_t absolu
te_deadline){
    request newRequest;
    dd_task* newTask = (dd_task*)auto_malloc(sizeof(dd_task));
    newTask->type = type;
    newTask->t_handle = t_handle;
    newTask->task_id = task_id;
    newTask->absolute_deadline = absolute_deadline;
    newRequest.task = newTask;
    newRequest.reqType = 1;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
}

void complete_dd_task(uint32_t task_id, bool isOverdue){
    request newRequest;
    newRequest.reqType = 2;
    newRequest.task_id = task_id;
    newRequest.isOverdue = isOverdue;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
}

node* get_active_dd_task_list(void){
    request newRequest;
    newRequest.reqType = 3;
    response returnMessage;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
    xQueueReceive(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
    return returnMessage.list;
}

node* get_complete_dd_task_list(void){
    request newRequest;
    newRequest.reqType = 4;
    response returnMessage;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
    xQueueReceive(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
    return returnMessage.list;
}

node* get_overdue_dd_task_list(void){
    request newRequest;
    newRequest.reqType = 5;
    response returnMessage;
    xQueueSend(ddsQueue, &newRequest, pdMS_TO_TICKS(500));
    xQueueReceive(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
    return returnMessage.list;
}

// User defined tasks
static void userDefinedTask1(void *pvParameters){
    int executionTime = 0;
    while(1){
        executionTime = (int) pvParameters;
        GPIO_SetBits(GPIOC, 0);
        GPIO_ResetBits(GPIOC, 1);
        GPIO_ResetBits(GPIOC, 4);
        // Code for the delay counter from
        // https://stm32f4-discovery.net/2014/09/precise-delay-counter/
        // 400 used as multiplier instead of 1000, as using 1000 produced a very imprecise d
elay
        executionTime = 400 * executionTime * multiplier - 10;
        while (executionTime > 0){
            executionTime--;
        }
        complete_dd_task(1, false);
    }
```

```
}

static void userDefinedTask2(void *pvParameters){
    int executionTime = 0;
    while(1){
        executionTime = (int) pvParameters;
        GPIO_ResetBits(GPIOC, 0);
        GPIO_SetBits(GPIOC, 1);
        GPIO_ResetBits(GPIOC, 4);
        // Code for the delay counter from
        // https://stm32f4-discovery.net/2014/09/precise-delay-counter/
        // 400 used as multiplier instead of 1000, as using 1000 produced a very imprecise d
elay
        executionTime = 400 * executionTime * multiplier - 10;
        while (executionTime > 0){
            executionTime--;
        }
        complete_dd_task(2, false);
    }
}

static void userDefinedTask3(void *pvParameters){
    int executionTime = 0;
    while(1){
        executionTime = (int) pvParameters;
        GPIO_ResetBits(GPIOC, 0);
        GPIO_ResetBits(GPIOC, 1);
        GPIO_SetBits(GPIOC, 4);
        // Code for the delay counter from
        // https://stm32f4-discovery.net/2014/09/precise-delay-counter/
        // 400 used as multiplier instead of 1000, as using 1000 produced a very imprecise d
elay
        executionTime = 400 * executionTime * multiplier - 10;
        while (executionTime > 0){
            executionTime--;
        }
        complete_dd_task(3, false);
    }
}

// Deadline driven task generator
static void ddsTaskGenerator(TimerHandle_t xTimer){
    uint32_t task_id = (uint32_t)pvTimerGetTimerID(xTimer);
    if(task_id == 1){
        taskInformation1.cycleNum += 1;
        create_dd_task(taskInformation1.t_handle, PERIODIC, task_id, taskInformation1.period
 * taskInformation1.cycleNum);
    } else if(task_id == 2){
        taskInformation2.cycleNum += 1;
        create_dd_task(taskInformation2.t_handle, PERIODIC, task_id, taskInformation2.period
 * taskInformation2.cycleNum);
    } else if(task_id == 3){
        taskInformation3.cycleNum += 1;
        create_dd_task(taskInformation3.t_handle, PERIODIC, task_id, taskInformation3.period
 * taskInformation3.cycleNum);
    }
}

// Monitor task
static void monitorTask(void *pvParameters){
    printf("Active list length: %d\n", getListLength(get_active_dd_task_list()));
    printf("Completed list length: %d\n", getListLength(get_complete_dd_task_list()));
    printf("Overdue list length: %d\n\n", getListLength(get_overdue_dd_task_list()));
}

// Cleanup function
void cleanup(){
    printf("Final report\n");
    printf("Active list length: %d\n", getListLength(get_active_dd_task_list()));
    printList(get_active_dd_task_list());
    printf("\nCompleted list length: %d\n", getListLength(get_complete_dd_task_list()));
    printList(get_complete_dd_task_list());
    printf("\nOverdue list length: %d\n", getListLength(get_overdue_dd_task_list()));
    printList(get_overdue_dd_task_list());
    freeAllLists();
    vTaskEndScheduler();
}

// Overdue task detected
void taskIsOverdue(){
    complete_dd_task((int)pvTimerGetTimerID(overdueDetectionTimer), true);
}

// DDS scheduler helper
void startTask(bool *isRunning, int *taskRunning){
    if(activeListHead == NULL){
        return;
    }
    // Create timer to stop overdue tasks
    if(activeListHead->task->absolute_deadline - getAbsoluteTime() > 0){
        if(overdueDetectionTimer == NULL){
            overdueDetectionTimer = xTimerCreate("overdueTaskDetection", pdMS_TO_TICKS (1 +
(activeListHead->task-
>absolute_deadline) - getAbsoluteTime()), pdFALSE, (void*)activeListHead->task-
>task_id, taskIsOverdue);
        } else {
            vTimerSetTimerID(overdueDetectionTimer, (void*)activeListHead->task->task_id);
            xTimerChangePeriod(overdueDetectionTimer, pdMS_TO_TICKS (1 + (activeListHead-
>task->absolute_deadline) - getAbsoluteTime()), 0);
```

```
        }
        vTaskResume(activeListHead->task->t_handle);
        xTimerReset(overdueDetectionTimer, 0);
        *isRunning = true;
        *taskRunning = activeListHead->task->task_id;
    } else {
        // Deadline is already passed
        // Add to the over due list and start another task if one exists
        addToTheEndOfTheList(&overdueListHead, removeNode(&activeListHead, activeListHead-
>task->task_id));
        startTask(isRunning, taskRunning);
    }
}

// DDS scheduler task
static void ddsSchedulerTask(void *pvParameters){
    request receivedRequest;
    response returnMessage;
    bool isRunning = false;
    int taskRunning = -1;
    while(1)
    {
        xQueueReceive(ddsQueue, &receivedRequest, pdMS_TO_TICKS(portMAX_DELAY));
        if(receivedRequest.reqType == 1){
            if(taskRunning != receivedRequest.task->task_id){
                vTaskSuspend(receivedRequest.task->t_handle);
            }
            if(addToTheSortedList(&activeListHead, receivedRequest.task)){
                if(activeListHead->next != NULL){
                    vTaskSuspend(activeListHead->next->task->t_handle);
                }
                isRunning = false;
                if(overdueDetectionTimer != NULL){
                    xTimerStop(overdueDetectionTimer, 0);
                }
            }
            if(!isRunning){
                startTask(&isRunning, &taskRunning);
            }
        } else if(receivedRequest.reqType == 2 && !receivedRequest.isOverdue){
            xTimerStop(overdueDetectionTimer, 0);
            addToTheEndOfTheList(&completedListHead, removeNode(&activeListHead, receivedReq
uest.task_id));
            startTask(&isRunning, &taskRunning);
        } else if(receivedRequest.reqType == 2 && receivedRequest.isOverdue){
            addToTheEndOfTheList(&overdueListHead, removeNode(&activeListHead, receivedReque
st.task_id));
            startTask(&isRunning, &taskRunning);
        } else if(receivedRequest.reqType == 3){
            returnMessage.list = activeListHead;
```

```c
            xQueueSend(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
        } else if(receivedRequest.reqType == 4){
            returnMessage.list = completedListHead;
            xQueueSend(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
        } else if(receivedRequest.reqType == 5){
            returnMessage.list = overdueListHead;
            xQueueSend(responseQueue, &returnMessage, pdMS_TO_TICKS(500));
        }
    }
}

// Main to setup tests and scheduler
int main(){
    initGPIO();
    // Initialize delay counter
    TM_Delay_Init();
    // Create queues
    ddsQueue = xQueueCreate(100, sizeof(request));
    responseQueue = xQueueCreate(100, sizeof(response));
    // Add the queues to the registry, for the benefits while debugging
    vQueueAddToRegistry(ddsQueue, "ddsQueue");
    vQueueAddToRegistry(responseQueue, "responseQueue");
    // Create DDS scheduler
    xTaskCreate(ddsSchedulerTask, "ddsSchedulerTask", configMINIMAL_STACK_SIZE, NULL, 3, NULL);
    // Create monitor F-task
    monitorTimer = xTimerCreate("monitorTask", pdMS_TO_TICKS(500), pdTRUE, 0, monitorTask);
    // Create a timer to end the scheduler after a set duration
    TimerHandle_t reportingPeriod = xTimerCreate("WrapUp", pdMS_TO_TICKS(REPORTING_PERIOD), pdFALSE, 0, cleanup);
    // Execute test bench
    if(TEST_BENCH == 1){
        generator_information task1 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask1, "userDefinedTask1", configMINIMAL_STACK_SIZE, (void*)95, 2, &task1.t_handle);
        taskInformation1 = task1;
        Task1Generator = xTimerCreate("ddsTask1Generator", pdMS_TO_TICKS(500), pdTRUE, (void*)1, ddsTaskGenerator);
        generator_information task2 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask2, "userDefinedTask2", configMINIMAL_STACK_SIZE, (void*)150, 2, &task2.t_handle);
        taskInformation2 = task2;
        Task2Generator = xTimerCreate("ddsTask2Generator", pdMS_TO_TICKS(500), pdTRUE, (void*)2, ddsTaskGenerator);
        generator_information task3 = {NULL, 0, 750};
        xTaskCreate(userDefinedTask3, "userDefinedTask3", configMINIMAL_STACK_SIZE, (void*)250, 2, &task3.t_handle);
        taskInformation3 = task3;
        Task3Generator = xTimerCreate("ddsTask3Generator", pdMS_TO_TICKS(750), pdTRUE, (void*)3, ddsTaskGenerator);
    } else if(TEST_BENCH == 2){
        generator_information task1 = {NULL, 0, 250};
        xTaskCreate(userDefinedTask1, "userDefinedTask1", configMINIMAL_STACK_SIZE, (void*)9
5, 2, &task1.t_handle);
        taskInformation1 = task1;
        Task1Generator =xTimerCreate("ddsTask1Generator", pdMS_TO_TICKS(250), pdTRUE, (void*
)1, ddsTaskGenerator);
        generator_information task2 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask2, "userDefinedTask2", configMINIMAL_STACK_SIZE, (void*)1
50, 2, &task2.t_handle);
        taskInformation2 = task2;
        Task2Generator = xTimerCreate("ddsTask2Generator", pdMS_TO_TICKS(500), pdTRUE, (void
*)2, ddsTaskGenerator);
        generator_information task3 = {NULL, 0, 750};
        xTaskCreate(userDefinedTask3, "userDefinedTask3", configMINIMAL_STACK_SIZE, (void*)2
50, 2, &task3.t_handle);
        taskInformation3 = task3;
        Task3Generator = xTimerCreate("ddsTask3Generator", pdMS_TO_TICKS(750), pdTRUE, (void
*)3, ddsTaskGenerator);
    } else if(TEST_BENCH == 3){
        generator_information task1 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask1, "userDefinedTask1", configMINIMAL_STACK_SIZE, (void*)1
00, 2, &task1.t_handle);
        taskInformation1 = task1;
        Task1Generator = xTimerCreate("ddsTask1Generator", pdMS_TO_TICKS(500), pdTRUE, (void
*)1, ddsTaskGenerator);
        generator_information task2 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask2, "userDefinedTask2", configMINIMAL_STACK_SIZE, (void*)2
00, 2, &task2.t_handle);
        taskInformation2 = task2;
        Task2Generator = xTimerCreate("ddsTask2Generator", pdMS_TO_TICKS(500), pdTRUE, (void
*)2, ddsTaskGenerator);
        generator_information task3 = {NULL, 0, 500};
        xTaskCreate(userDefinedTask3, "userDefinedTask3", configMINIMAL_STACK_SIZE, (void*)2
00, 2, &task3.t_handle);
        taskInformation3 = task3;
        Task3Generator = xTimerCreate("ddsTask3Generator", pdMS_TO_TICKS(500), pdTRUE, (void
*)3, ddsTaskGenerator);
    }
    // Start the initial tasks for time 0
    ddsTaskGenerator(Task1Generator);
    ddsTaskGenerator(Task2Generator);
    ddsTaskGenerator(Task3Generator);
    // Start monitor timer
    //xTimerStart(monitorTimer, 0);
    // Start the task generators
    xTimerStart(Task1Generator, 0);
    xTimerStart(Task2Generator, 0);
    xTimerStart(Task3Generator, 0);
    // Start the reporting period
```

```c
    xTimerStart(reportingPeriod, 0);
    // Start the scheduler
    vTaskStartScheduler();
}

/*-----------------------------------------------------------*/
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. */
    }
}
```

```c
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. */
}
```