UNIVERSITY OF VICTORIA

Department of Electrical and Computer Engineering
ECE 455 - Real Time Computer Systems Design
Project

Project 2 Report

Spring 2022
Report Submitted on April 5, 2022

To: Dr Amirali Baniasadi
Laboratory Section B01

Spencer Davis - V00759537
Mustafa Wasif - V00890184

Table of Contents

List of Figures and Tables

Figures

Tables

1. Introduction

The objective of this project was to design and implement on the STM32F4 microcontroller [1] an earliest-deadline-first (EDF) scheduler to dynamically manage tasks having hard deadlines [2]. FreeRTOS was used to implement this scheduler. FreeRTOS does not natively support EDF scheduling, so the scheduler was built on top of the existing FreeRTOS scheduler. The design implemented follows the proposed design presented in the lab manual: deadline-driven tasks (DD-Tasks) are represented and managed as structs; then when a DD-Task needs to execute, it is instantiated as a corresponding FreeRTOS task (F-Task), and the priority of that F-Task is raised above the F-Tasks corresponding to other DD-Tasks. A monitor task periodically reports the number of active, complete, and overdue user-defined tasks. The main components of this system are as follows:

- DD_Task_Generator_Task: this generates periodic tasks and notifies the DDS_Task whenever a task is released.
- DDS_Task: this receives from the xDDS_Queue all DD-Tasks released by the DD_Task_Generator_Task; sorts DD-Tasks by ascending order of absolute deadline (ie applies the EDF algorithm); manages lists of active, completed, and overdue DD-Tasks; controls the execution of DD-Tasks by dynamically raising and lowering the priorities of the corresponding F-Tasks; and responds to requests from the Monitor_Task for the current sizes of the lists of active, completed, and overdue DD-Tasks.
- Task lists: three linked lists of DD_Task structs; they are used to store and manage the struct representations of all active, completed, and overdue DD-Tasks. The task lists are internal to the DDS_Task. The EDF algorithm is implemented by sorting the items in the active_list.
- Monitor_Task: this periodically requests from the DDS_Task the current sizes of the task lists.
- UD_Task_1, UD_Task_2, and UD_Task_3: these are F-Tasks which each execute for a duration defined by the execution times of periodic tasks 1, 2, and 3, respectively. In general, UD_Task_N is used to execute every instance of periodic task N.

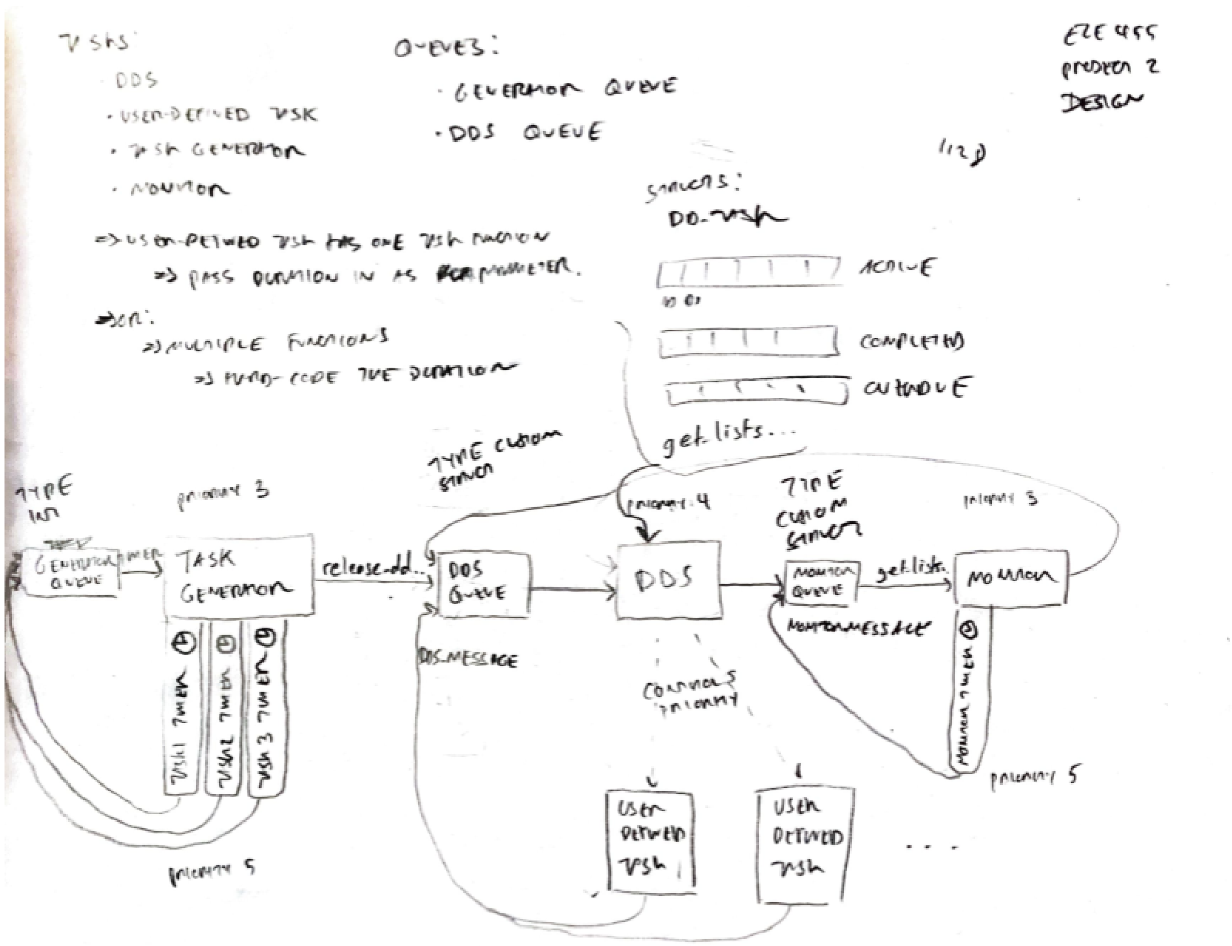
Inter-task communication is performed exclusively using queues. This report presents in detail our design solution, including explanations of all F-Tasks, data structures, and algorithms used. It also presents the output of our system for each of the three assigned test benches, discusses challenges we faced and the solutions we found, and outlines the limitations of our system and possible improvements.

2. Design Solution

2.1 Initial Design

Before implementation, we created a candidate design by analyzing the project requirements. We created an initial design document by hand; this document can be seen in Figure 1.

Figure 1. Initial design document.



This initial design outlines the F-Tasks, queues, timers, and data structures used, and the relationships between them. The task generator, the DDS, and the monitor all wait on respective queues for messages indicating the occurrence of some event to which they must respond. The task generator waits on a queue to receive notification that a timer corresponding to the release period of a task has expired; it then notifies the DDS of the corresponding task release. The DDS waits on a queue to receive notification that a task has been released, a task has finished executing, or the monitor has requested the current sizes of the task lists; in response, the DDS manages the task lists and schedules the next DD-Task, or sends the list sizes to the monitor. The DDS stores and manages the task lists internally, and schedules them by modifying the priorities of the corresponding F-Tasks. And the monitor waits on a queue for notification that its request timer has expired, or to receive from the DDS the aforementioned list sizes following a request.

Our initial priority scheme can be seen in Table 1.

Table 1. Priority scheme in initial design.

Component	**Priority**
Timers	5
DDS	4
Task generator	3
Monitor	3
Currently-scheduled user-defined F-Task	2
Idle user-defined F-Task	1

The reasoning for this prioritization was as follows: the timers should have the highest priority, to prevent the delay of time-based events; the DDS should have the next-highest priority, to prevent it from waiting for the task generator or monitor, since this could artificially delay scheduling; the task generator and monitor should have equally the next-highest priority, so they will not block the DDS, but can otherwise still perform in a timely manner the time-critical responsibilities of generating tasks and reporting the system state; and the currently-scheduled user-defined F-Task should have the next-highest priority, so it always blocks the other user-defined F-Tasks having lower priority.

This prioritization scheme was modified in the final design; details about this can be found in section 2.3.

The DDS presents a set of interface functions through which other F-Tasks interact with it; no F-Task ever directly accesses the internal state of the DDS – the interface functions are always used. The lab manual suggests using the following five interface functions:

- release_dd_task()
- complete_dd_task()
- get_active_dd_task_list()
- get_completed_dd_task_list()
- get_overdue_dd_task_list()

In our initial design, we decided to combine the last three listed functions into a single function, since the monitor always wants to obtain the number of active, completed, and overdue tasks together; then by using a single interface function for this purpose, we can improve readability and maintainability. This decision persisted into our final design.

We also decided in our initial design that rather than passing to the monitor either references to, or copies of, the task lists, we would implement the DDS to maintain variables storing the current sizes of these lists, by updating them whenever the lists are modified; then whenever the monitor needs to obtain the list sizes, the sizes have already been computed, thereby reducing overhead. If we instead send to the monitor copies of, or references to, the task lists, the monitor must repeatedly scan through the lists to determine their sizes. This decision persisted into our final design.

All time is measured in number of ticks since the scheduler started, which is equivalent to milliseconds, since the tick rate of the system is set to 1000 Hz. This value is obtained using xTaskGetTickCount().

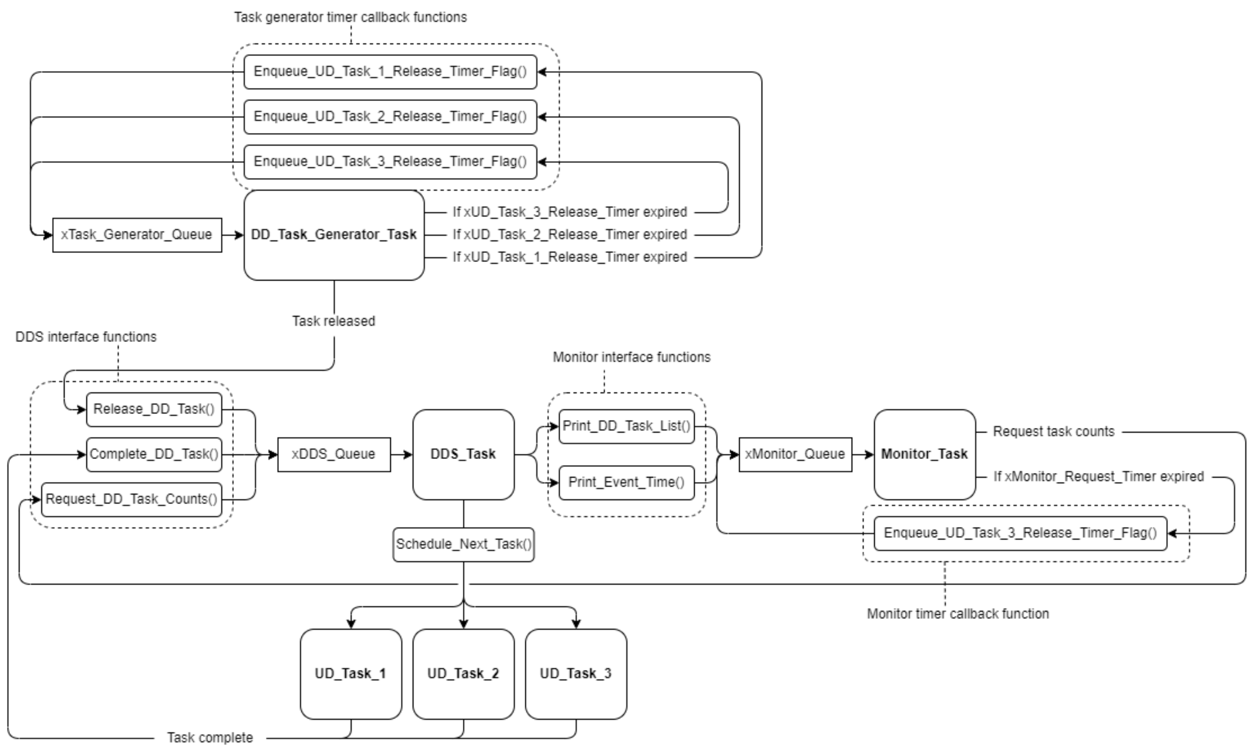
This was not a complete design: the implementation details of each task and the various algorithms therein were left undecided. Also, the points at which task handles would be created, and the points at which xTaskCreate() and vTaskDelete() would be called, were not decided. Due to the complexity of the problem, we decided to first implement skeletons of the major components, as well as the queues for inter-task communication, to better understand the problem space, then use that understanding to make those decisions.

Other than the points mentioned above, the high-level construction of our final design was the same as our initial design. This final design is presented in detail in the following sections.

2.2 Final Design

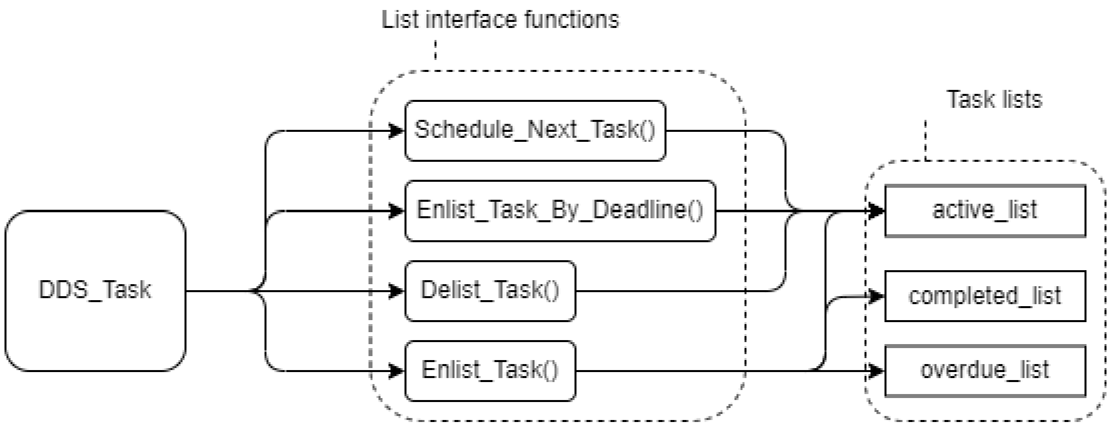
A high-level view of our final design can be seen in Figure 2.

Figure 2. Final design overview



In this figure, we can see that the configuration of the major components is unchanged from our initial design. However, this diagram shows the final names of each component as they appear in the code, as well as the final interface functions. Notice that in addition to implementing interface functions for the DDS_Task, we also implemented interface functions for the Monitor_Task; this was done to improve readability and maintainability. The function Schedule_Next_Task() is shown as the interface to the user-defined F-Tasks: it is used by the DDS_Task to start the execution of the F-Task corresponding to the DD_Task currently at the head of the active_list. To make Figure 2 more readable, the lists of tasks managed by the DDS_Task are not shown. These lists are manipulated by the DDS_Task using a set of interface functions; these functions and their interactions with the lists can be seen in Figure 3.

Figure 3. Task lists, list interface functions, and the DDS_Task.



The DDS_Task interface functions send DDS_Message structs to the DDS_Task via the xDDS_Queue. The final DDS_Task interface functions are outlined in Table 2. The definition of the DDS_Message struct can be seen at line 23 in the code.

Table 2. DDS_Task interface functions.

Function name	**Role**
Release_DD_Task()	Used by the DD_Task_Generator_Task to notify the DDS_Task of a task release. Packages as a DDS_Message struct all information necessary for the DDS_Task to release a task, and sends it to DDS_Task on the xDDS_Queue.
Complete_DD_Task()	Used by any user-defined F-Task to notify the DDS_Task that it has finished executing. Packages as a DDS_Messaage struct a flag indicating itself as the sender, then sends it to DDS_Task on the xDDS_Queue.
Request_DD_Task_Counts()	Used by the Monitor_Task to request from the DDS_Task the current sizes of the task lists. Packages as a DDS_Messaage struct a flag indicating this request, then sends it to DDS_Task on the xDDS_Queue.

The Monitor_Task interface functions send Monitor_Message structs to the Monitor_Task via the xMonitor_Queue. The Monitor_Task interface functions are outlined in Table 3.

Table 3. Monitor_Task interface functions.

Function name	**Role**
Print_DD_Task_List()	Used by the DDS_Task to send to the Monitor_Task the current sizes of the task lists. Packages these values as a Monitor_Message, then sends it to Monitor_Task on the xMonitor_Queue.
Print_Event_Time()	Used by the DDS_Task to send to inform the Monitor that a task has just been released, has completed on time, or has completed after its deadline (i.e. is overdue). Packages as a Monitor_Message the time, type, and corresponding task number (ie 1, 2 or 3) of the event, then sends it to Monitor_Task on the xMonitor_Queue. This is additional functionality not specified in the lab manual; we implemented this to enhance the reporting capabilities of the Monitor_Task, in order to verify the system's correct behaviour during the test benches.

The behaviour of the list interface functions is outlined in Table 4.

Table 4. List interface functions

Function name	**Role**
Schedule_Next_Task()	Starts or resumes execution of the DD_Task currently at the head of active_list, using the corresponding F-Task. Raises the priority of the F-Task to 2. Lowers the priority of the

	F-Task corresponding to the next-in-line DD_Task to 1.
Enlist_Task_By_Deadline()	Adds a DD_Task struct to a task list. Sorts the DD_Task into position by increasing order of absolute deadline (EDF). If two tasks have the same deadline, then the task which was enlisted first gets priority. Never sorts a PERIODIC task behind an APERIODIC task.
Delist_Task()	Removes the DD_Task at the head of a list.
Enlist_Task()	Adds a DD_Task to the back of a list.

The sorting behaviour of Enlist_Task_By_Deadline() makes the active list a priority queue data structure, sorted in increasing order of absolute deadline; this sorting behaviour is what gives us EDF scheduling, since Schedule_Next_Task() always schedules the DD_Task at the head of the active_list.

2.3 List of all F-Tasks and their priority levels

The final prioritization scheme used is outlined in Table 5.

Table 5. List of all F-Tasks and their priority levels.

Component	**Priority**
Timers	6
DD_Task_Generator_Task	5
DDS_Task	4
Monitor_Task	3
Currently-scheduled user-defined F-Task	2
Idle user-defined F-Tasks	1

This prioritization scheme differs from our initial design: here we prioritize task release events above all else, thereby minimizing the amount of delay that gets introduced to release times.

This improves the accuracy of the timing of task release events over many hyper periods; the performance of this prioritization scheme is expanded upon in section 3.5.

2.4 Implementation details for each of the main DDS functions

2.4.1 DD_Task_Generator_Task implementation details.

The DD_Task_Generator_Task can be seen at line 178 in the code. The job of the DD_Task_Generator_Task is to trigger task release events periodically, according to the periods of each task. Task handles for the three periodic tasks are defined, but not initialized, in the DD_Task_Generator. Three timers are initialized in the main() function, having periods corresponding to the periods of these tasks; these timers are set to auto-reload upon expiry. When the scheduler starts, the DD_Task_Generator_Task starts each timer, then waits on the xTask_Generator_Queue. Whenever any of these timers expires, its callback function adds to the xTask_Generator_Queue a flag indicating the task to which the timer corresponds (ie 1, 2, or 3). When the DD_Task_Generator_Task receives such a flag from the queue, it notes the current time as the release time, calculates the absolute deadline, then calls Release_DD_Task(), passing it the following information:

- Identifier of which task is being released (ie 1, 2, or 3)
- Corresponding uninitialized task handle.
- Task type = PERIODIC
- Task ID (a unique incremental identifier for this instance of the task)
- Release time
- Absolute deadline

Release_DD_Task() then packages this information as a DDS_Message struct, sets the message type to TASK_RELEASE, and adds it to the xDDS_Queue.

After calling Release_DD_Task(), the DD_Task_Generator_Task increments the task ID counter; note that this ID field is not actually used in this implementation: it was implemented in case it became useful, but we ultimately didn’t need it. Also note that immediately after starting, the DD_Task_Generator_Task directly calls the three timer callback functions corresponding to the three task timers, causing an initial release of all tasks at time zero.

2.4.2 DDS_Task implementation details.

2.4.2.1 Overview

The DDS_Task can be seen at line 345 in the code. The job of the DDS_Task is to manage the execution of all released tasks, manage the task lists, and to provide to the Monitor_Task the current sizes of the task lists upon request. We have also made the DDS_Task capable of providing to the Monitor_Task the time of all task release, completion, and overdue events immediately as they occur, and without request. This improves our ability to observe the behaviour of the system in order to verify its correctness.

The task lists are called the active_list, completed_list, and overdue_list. The active_list contains all DD_Tasks released but not yet done executing, the completed_list contains all DD_Tasks which completed by their deadline, and the overdue_list contains all DD_Tasks which completed after their deadline. The EDF algorithm is implemented by sorting the active_list in increasing order of absolute deadline, by the Enlist_Task_By_Deadline() function, whenever a task is added to the active_list.

When the DDS_Task starts, it waits on the xDDS_Queue to receive a DDS_Message. Upon receiving a message, it checks the DDS_Message.message_type field to determine how to respond; this field indicates whether the message signifies a task release, signifies that a task has finished executing, or signifies that the Monitor_Task has request the current sizes of the task lists.

2.4.2.2 Handling task release events

If the DDS_Message signifies a task release, then it has been sent by the DD_Task_Generator_Task, and contains all information necessary for the DDS to create a DD_Task. The DDS_Task allocates on the heap a DD_Task struct storing the data received in the DDS_Message, then adds the DD_Task struct to the active list. If the task type is PERIODIC, then the DDS_Task uses Enlist_Task_By_Deadline() for this purpose, since Enlist_Task_By_Deadline() sorts items into the list in ascending order of absolute deadline, giving us an EDF ordering of active tasks. If the task type is APERIODIC, then the DDS_Task uses Enlist_Task() instead, since Enlist_Task() simply sends the task to the back of the list, thereby ensuring that APERIODIC tasks do not disrupt the execution of PERIODIC tasks. Note that the DD_Task->started field is also initialized here to *false*; this triggers the call to xTaskCreate() in the Schedule_Next_Task() function. Also note that Enlist_Task_By_Deadline() never sorts a PERIODIC task behind an APERIODIC task.

The DDS_Task stores the current sizes of the task lists in three variables: active_count, completed_count, and overdue_count. After adding a released task to the active_list, the DDS_Task increments active_count, and calls Schedule_Next_Task() to execute the F_Task corresponding to the DD_Task at the head of the active_list. Since we only insert into the active_list using the algorithm described above, then the DD_Task at the head of the active_list is always the highest-priority DD_Task, according to EDF.

The Schedule_Next_Task() function instantiates a DD_Task as its corresponding user-defined F-Task, by calling xTaskCreate() using the uninitialized task handle stored in the DD_Task. The function then raises the priority of that F-Task to 2, and lowers the priority of the F-Task corresponding to the next-highest-priority DD_Task to 1, causing the first F-Task to begin executing. The DD_Task->started field is then set to true; then if Schedule_Next_Task() is called again on the same DD_Task – as in the case of resumed execution after preemption – then we can avoid calling xTaskCreate() a second time for that DD_Task.

Note that xTaskCreate() is called for every DD_Task; a corresponding call to vTaskDelete() is later called when each DD_Task completes; ie the DDS_Task continually creates and destroys user-defined F-Tasks. This design choice is expanded upon in section 3.5.

2.4.2.3 Handling task completion and overdue events

If the DDS_Message indicates that a task has finished executing, then it has been sent by one of the user-defined F-Tasks upon that F-Tasks completion. Since this F-Task always corresponds to the DD_Task at the head of the active_list, then the DDS_Task sets the completion time of that DD_Task to the current time, and un-instantiates the F-Task using vTaskDelete(). It then checks if the task was completed by its deadline: if the task completed by its deadline, then the DDS_Task removes it from the active_list, adds it to the completed_list, and increments completed_count; if the task did not complete by its deadline, then the DDS_Task removes it from the active_list, adds it to the overdue_list, and increments overdue_count. The DDS_Task then decrements active_count.

2.4.2.4 Handling requests from the Monitor_Task

If the DDS_Message signifies that the Monitor_Task has requested the current sizes of the task lists, then DDS_Task passes these values to the Print_DD_Task_List_Sizes() monitor interface function. This function packages these values as a Monitor_Message, and sends it to the Monitor_Task on the xMonitor_Queue.

2.4.2.5 Additional functionality: sending event times to the Monitor_Task

To better observe the behaviour of the system, we made the DDS_Task capable of sending the times of task release, completion, and overdue events to the Monitor_Task immediately as they occur. This is performed using the Print_Event_Time() monitor interface function. This function accepts as input the following three parameters:

- Flag indicating the type of event (task release, task complete, or task overdue)
- Flag indicating the task involved (1, 2, or 3)
- Time at which the event occurred

The global variables at line 117 in the code can be used to quickly enable and disable this behaviour. Note that this behaviour does not require a request from the Monitor_Task: the messages are sent to the Monitor_Task immediately whenever they occur.

2.4.3 Monitor_Task implementation details.

The Monitor_Task can be seen at line 261 in the code. The job of the Monitor_Task is to periodically request from the DDS_Task the current sizes of the task lists, and print those values when received. We have also made the Monitor_Task capable of printing the times of task release, completion, and overdue events immediately as they occur, and without request.

The xMonitor_Request_Timer is initialized in the main() function and used by the Monitor_Task to periodically request the list sizes from the DDS_Task. The xMonitor_Request_Timer is configured to auto-reload upon expiry. Upon starting, the Monitor_Task starts the xMonitor_Request_Timer and waits on the xMonitor_Queue for a Monitor_Message. Upon receiving a message, the Monitor_Task checks whether the message contains a task release time, a task completion time, a task overdue time, a response to a request to the DDS_Task (ie containing the current list sizes), or a flag indicating that the xMonitor_Request_Timer has expired.

If the Monitor_Message contains a task release time, completion time, or overdue time, the Monitor_Task prints the event time, corresponding task number (ie 1, 2, or 3), and a letter indicating whether the task was released (R), completed by its deadline (C), or completed after its deadline (O).

If the Monitor_Message contains the current sizes of the task lists, then the Monitor_Task prints those values along with the current time.

If the Monitor_Message contains a flag indicating that the xMonitor_Request_Timer has expired, then the Monitor_Task requests the current task list sizes from the DDS_Task, using the Request_DD_Task_Counts() function.

2.4.4 User-defined F-Task implementation details.

The user-defined F-Tasks UD_Task_1, UD_Task_2, and UD_Task_3 can be seen at lines 544, 568, and 592 in the code, respectively. The job of these tasks is to simulate the execution of some arbitrary user-defined code, for prescribed durations, as instantiations of the DD_Task structs managed by the DDS_Task. UD_Task_1, UD_Task_2, and UD_Task_3 correspond to periodic tasks 1, 2, and 3, respectively. For example, if a DD_Task corresponding to task 1 is selected for scheduling by the DDS_Task, then the DDS_Task calls xTaskCreate() using UD_Task_1, and using the uninitialized task handle stored in the DD_Task struct. When the F-Task finishes executing, the DDS_Task calls vTaskDelete() using the same task handle. The durations for which these F-Tasks execute are controlled by the #define statements at line 104 in the code.

Since DD_Tasks can preempt each other, it was necessary to implement the user-defined F-Tasks to execute for the specified duration *not including blocking time*; for example, if UD_Task_1 executes for some time, then is preempted by UD_Task_2, then continues executing after UD_Task_2 completes, the time during which UD_Task_2 executed must not count towards the total execution time of UD_Task_1. To achieve this, we used two nested while loops: the outer loop exists only since FreeRTOS tasks must contain infinite while loops (the

F-Task completes at the end of the first iteration of the outer loop); the inner loop implements the execution-time functionality, using the following three time variables:

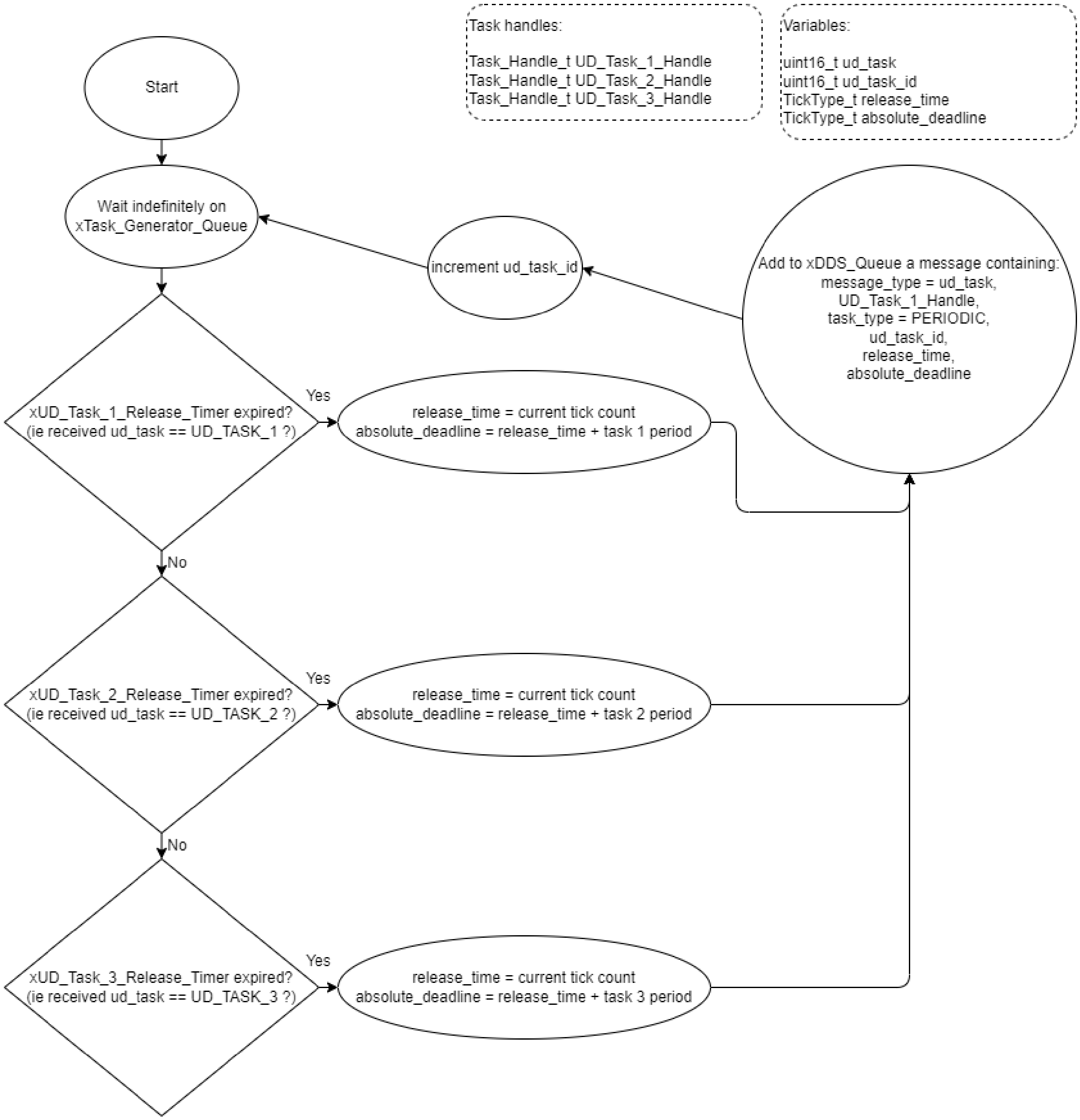
- TickType_t ticks_elapsed
- TickType_t last_time_ticks
- TickType_t current_time_ticks

The variable ticks_elapsed is initialized to zero, and last_time_ticks is initialized to the current tick count before entering the outer loop. On each iteration of the inner loop, we set current_time_ticks to the current tick count, then check if it differs from last_time_ticks; if it differs, then we increment ticks_elapsed, and set last_time_ticks to the value of current_time_ticks. When ticks_elapsed meets or exceeds the prescribed execution time, the F-Task calls Complete_DD_Task() and is subsequently uninstantiated by the DDS_Task, using vTaskDelete(). This works since the while loop can iterate many times in the duration between two tick interrupts: then if current_time_ticks differs from last_time_ticks (and the F-Task has not been preempted) we know that that it differs by one, so we increment ticks_elapsed; if preemption *has* occurred, then the ticks may differ by an arbitrary number, but we still increment ticks_elapsed only by one – in other words, we resume counting time where we left off.

2.5 Flow chart of how the Deadline-Task Generator works

A visual presentation of the deadline-task generator algorithm can be seen in Figure 4. This algorithm is implemented in the DD_Task_Generator_Task at line 178 in the code. Recall that the task timers auto-reload after expiry.

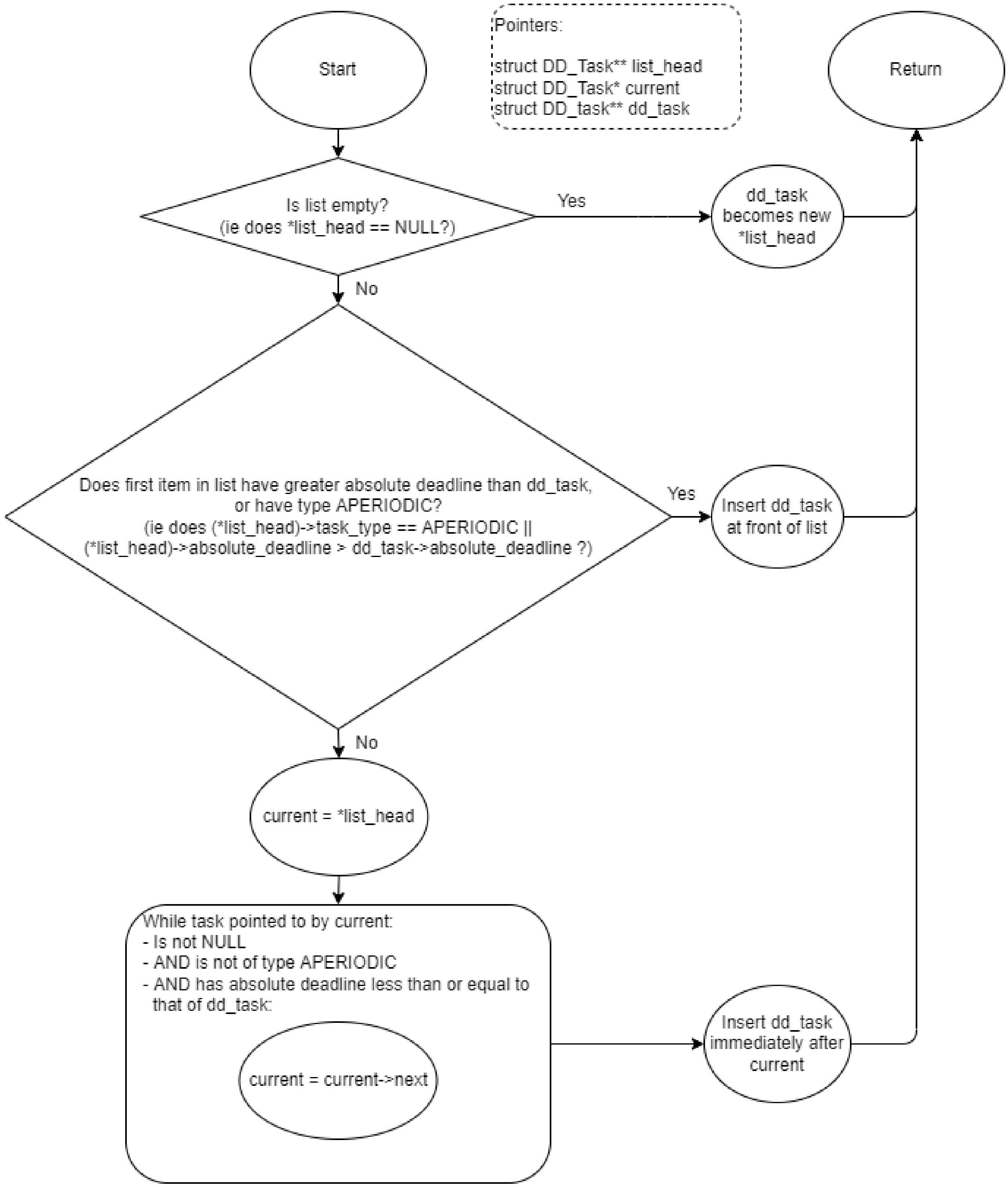
Figure 4. The task generator algorithm



2.6 Flow chart of the DD-Task sorting algorithm

A visual presentation of the DD_Task sorting algorithm implementing EDF can be seen in Figure 5. This algorithm is implemented in the Enlist_Task_By_Deadline() list-interface function at line 419 in the code; the DDS_Task calls this function whenever it adds a PERIODIC task to the active_list.

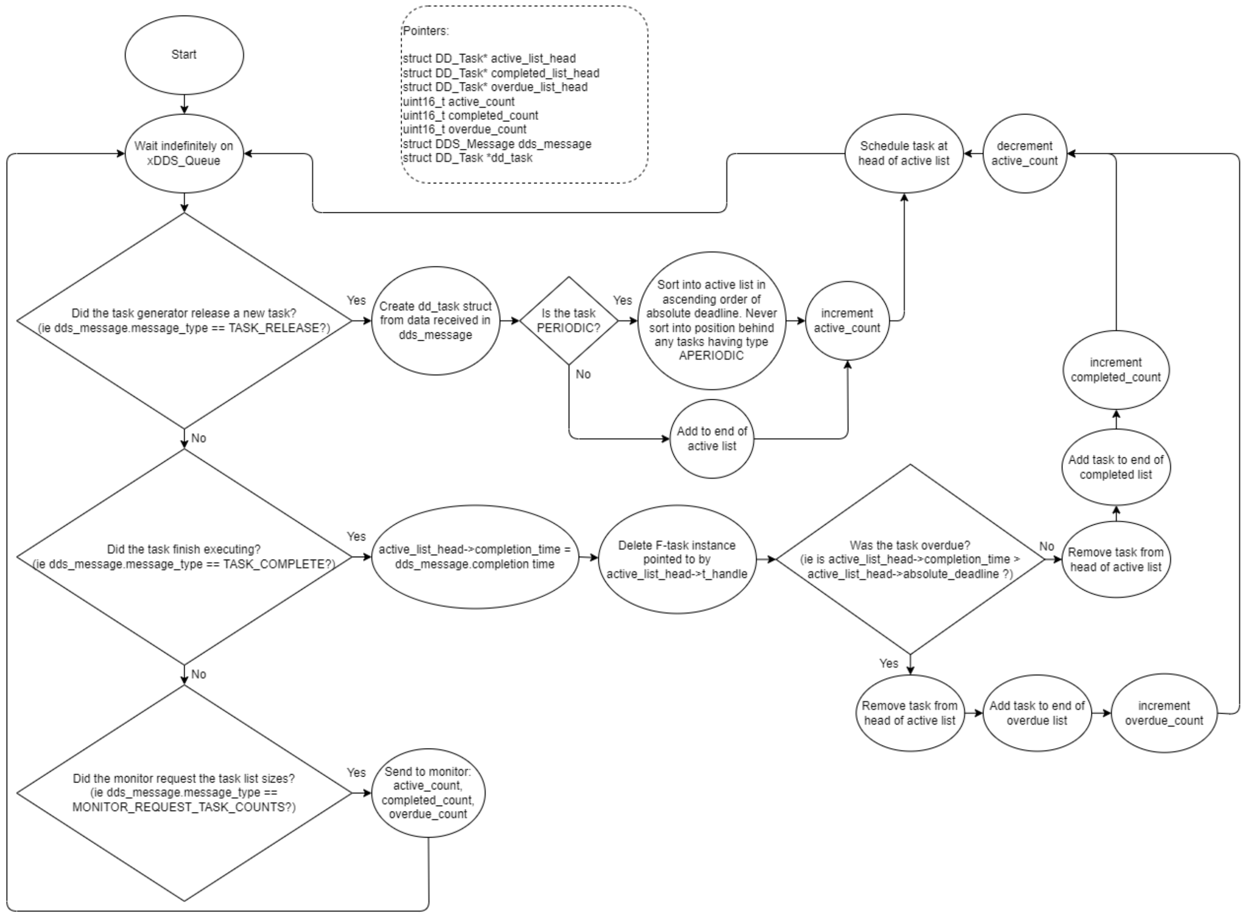
Figure 5. The task sorting algorithm for the active list



2.7 Flow charts of how the DD Scheduler works

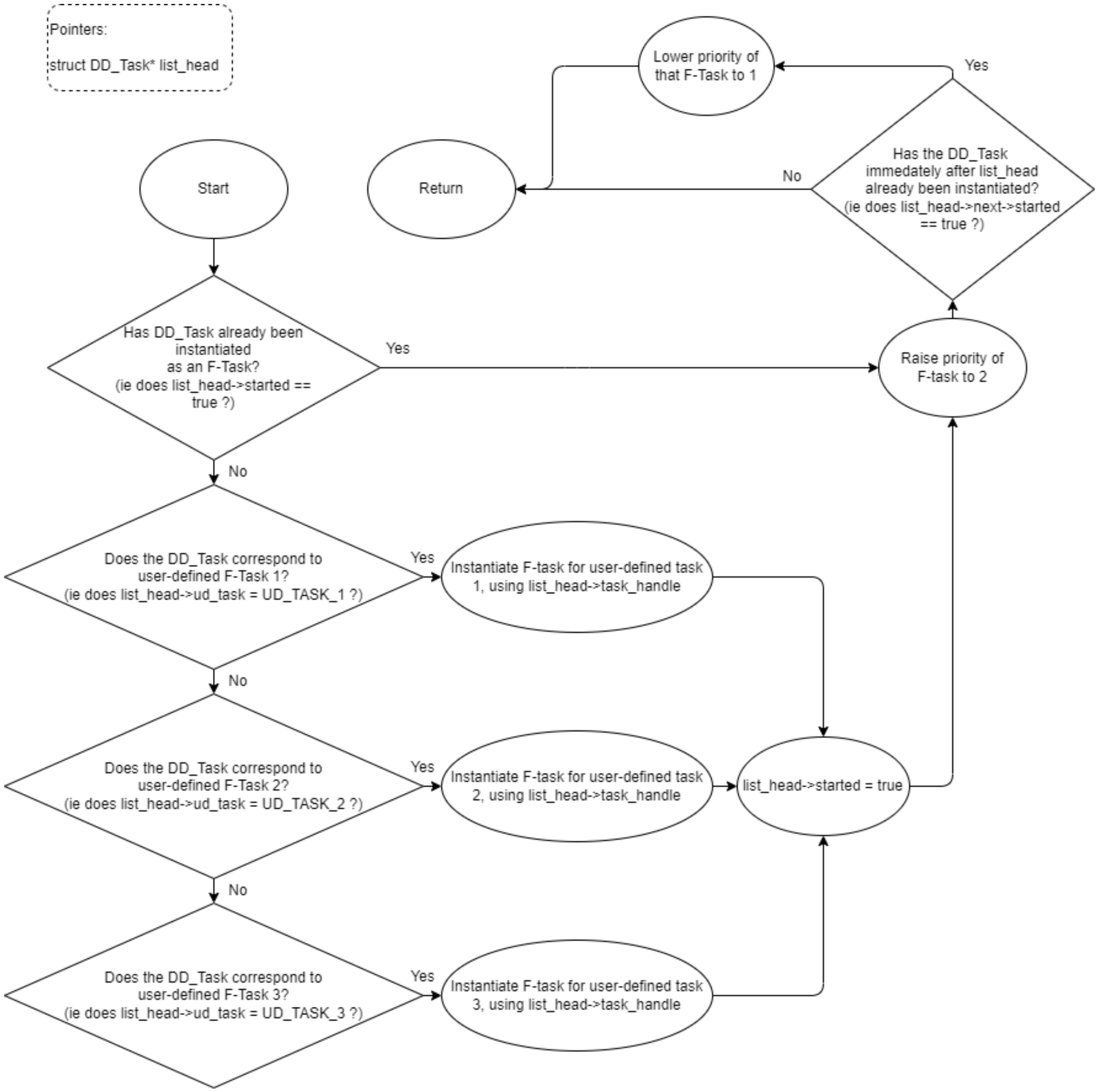
A visual overview of how the DD Scheduler works can be seen in Figure 6. This algorithm is implemented in the DDS_Task at line 345 in the code. This diagram shows the interactions of the DDS_Task with the xDDS_Queue, the tasks lists, and the xMonitor_Queue.

Figure 6. The DDS algorithm overview



A visual presentation of how DD_Tasks are instantiated as user-defined F-Tasks can be seen in Figure 7. This algorithm is implemented in the Schedule_Next_Task() function at line 472 in the code.

Figure 7. Algorithm to schedule task currently at head of active list



3. Discussion

3.1 Expected results and Gantt charts

Before implementation, we created Gantt charts to analyze the expected schedules of test benches 1, 2, and 3 under EDF scheduling. These Gantt charts can be seen in Figures 8, 9, and 10.

Figure 8. Gantt chart of test bench 1 under EDF.

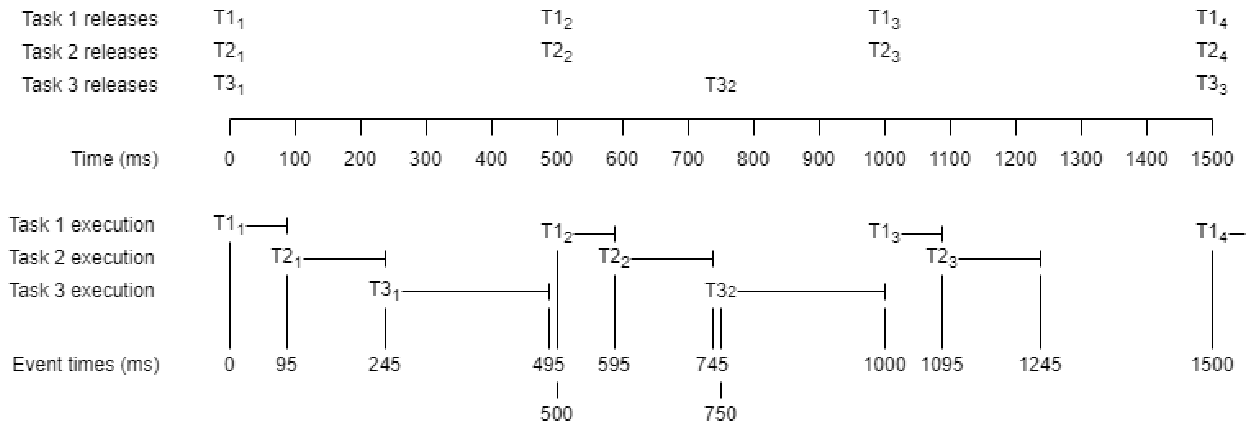


Figure 9. Gantt chart of test bench 2 under EDF.

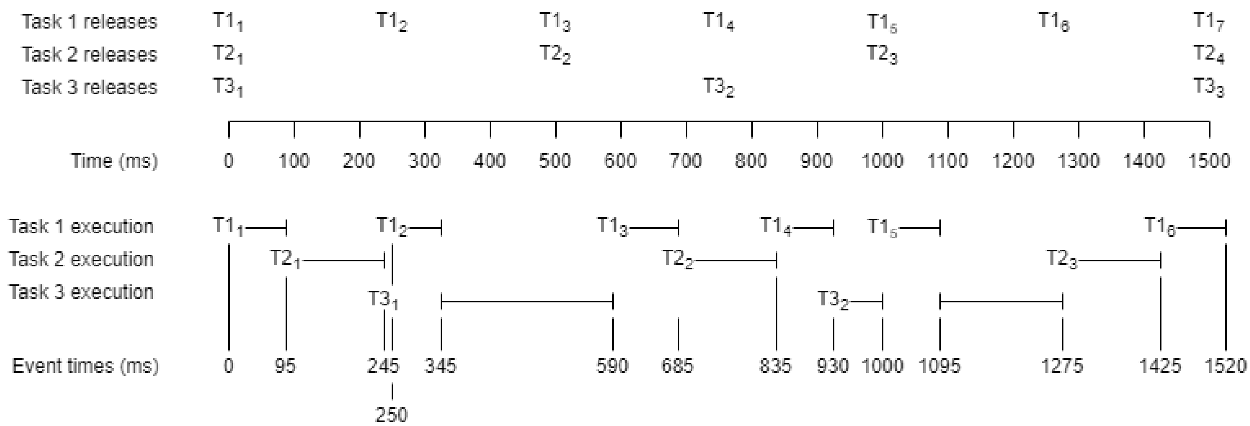
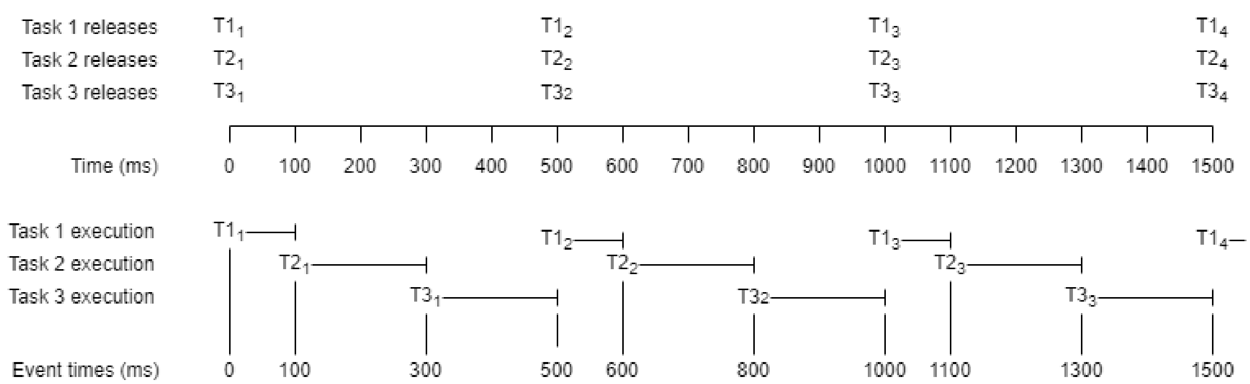


Figure 10. Gantt chart of test bench 3 under EDF.



Upon implementing our solution we observed that the system’s behaviour matched the Gantt charts above. Detailed breakdowns of these experimental results can be seen in sections 3.2, 3.3, and 3.4. All test benches were run for at least one hyperperiod; since the hyperperiod for test benches 1 and 2 is 1500 ms, and the hyperperiod for test bench 3 is 500 ms, we ran all test benches for at least 1500 ms.

Note that the Gantt charts align with the expected schedulability of each test bench, using the EDF schedulability test.

We expect test bench 1 to be schedulable, since:

$$(95 \,/\, 500) \;+\; (150 \,/\, 500) \;+\; (250 \,/\, 750) \;=\; 0.823 \;<=\; 1$$

Likewise, we expect test bench 2 not to be schedulable, since:

$$(95 \,/\, 250) \;+\; (150 \,/\, 500) \;+\; (250 \,/\, 750) \;=\; 1.013 \;>\; 1$$

This can be seen in Figure 9, where the sixth instance of task 1 completes at t = 1520, past its deadline of t = 1500.

And we expect test bench 3 to be schedulable, since:

$$(100 / 500) + (200 / 500) + (200 / 500) = 1$$

Then for test bench 3, we expect schedulability with 100% CPU utilization.

3.2 Test Bench #1 Results

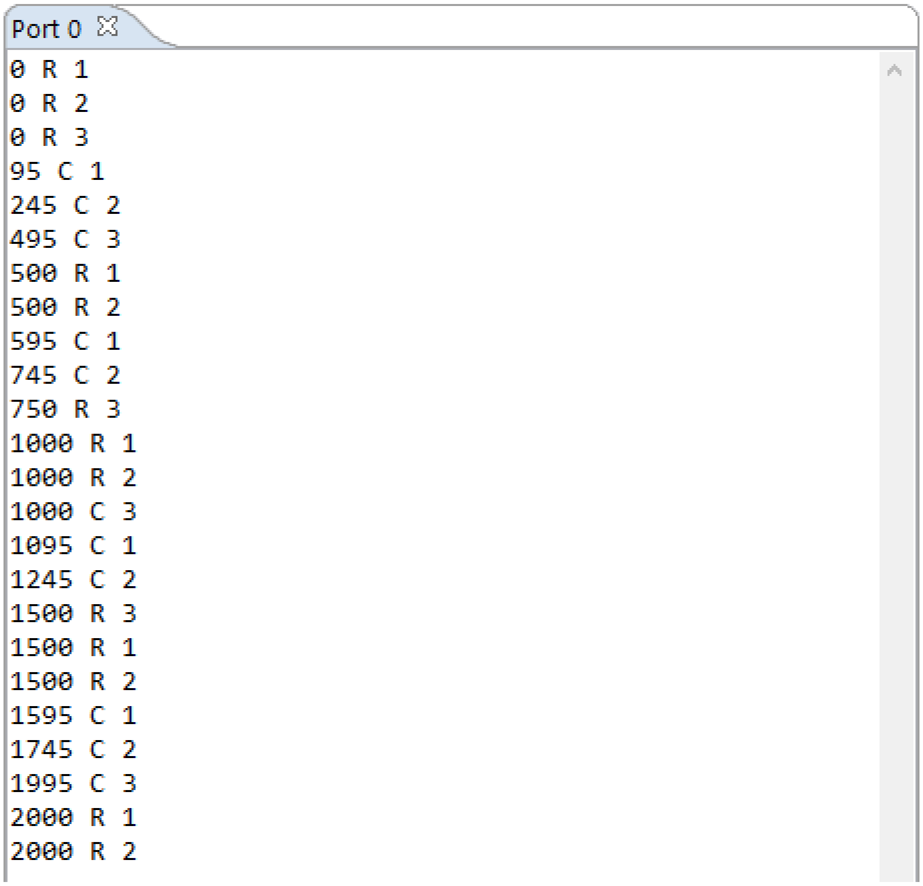
We subjected the system to each test bench twice: once with the Monitor_Task configured to print release times, completion times, and overdue times; and once with the Monitor_Task configured to periodically print the sizes of the task lists. When printing the task list sizes, the request period was set to 250 ms; this was configured using the #define statement at line 114 in the code.

The sequence of events during test bench 1 can be seen in Table 6. Expected times are taken from the Gantt chart in Figure 8. A screenshot of the console output during the execution run can be seen in Figure 11.

Table 6. Table of event times during test bench 1.

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 completed	95	95
5	Task 2 completed	245	245
6	Task 3 completed	495	495
7	Task 1 released	500	500
8	Task 2 released	500	500
9	Task 1 completed	595	595
10	Task 2 completed	745	745
11	Task 3 released	750	750
12	Task 1 released	1000	1000
13	Task 2 released	1000	1000
14	Task 3 completed	1000	1000
15	Task 1 completed	1095	1095
16	Task 2 completed	1245	1245
17	Task 3 released	1500	1500
18	Task 1 released	1500	1500
19	Task 2 released	1500	1500

Figure 11. Screenshot of console output when monitoring test bench 1 for event times.



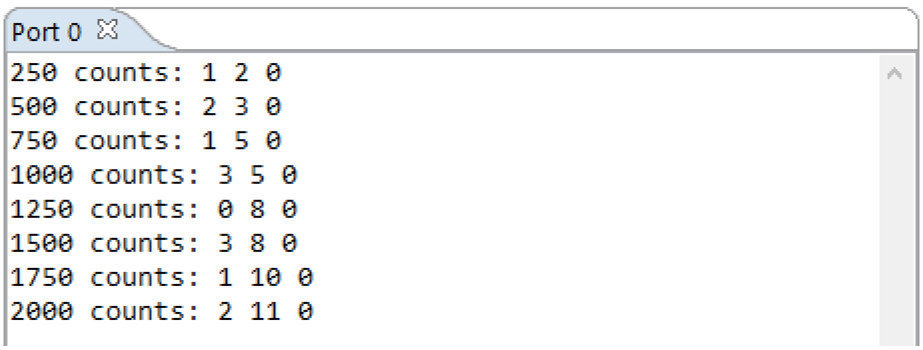
Observe in the figures above that the expected times match the measured times in all cases, indicating the correct execution of test bench 1. We can also see a phenomena indicating limitations of our system: when tasks 1, 2, and 3, are released at the same time, and after the initial release at time zero, the Monitor_Task is notified of the release of task 3 before those of tasks 1 and 2. This occurs because tasks 1 and 2 have a shorter period than task 3, and the timer callback function introduces a small amount of overhead; then tasks having a shorter period accumulate a gradual delay relative to tasks having longer periods. This limitation is expanded upon in section 4.

The observed number of active, completed, and overdue tasks after one hyperperiod of test bench 1 can be seen in Table 7; the expected values are derived from the Gantt chart in Figure 8. The observed number of active, completed, and overdue tasks every 250 ms during test bench one can be seen in Figure 12; in the console output, the three right-hand columns, from left to right, indicate the number of active, completed, and overdue tasks, respectively. The MONITOR_REQUEST_PERIOD of 250 ms intervals could be varied as required.

Table 7. Number of active, completed, and overdue tasks after one hyperperiod (1500 ms) in test bench 1.

	Measured	Expected
Number of active DD-Tasks	3	3
Number of completed DD-Tasks	8	8
Number of overdue DD-Tasks	0	0

Figure 12. Screenshot of console output of number of active, completed, and overdue tasks in test bench 1, at 250 ms intervals.


Port 0

```
250 counts: 1 2 0
500 counts: 2 3 0
750 counts: 1 5 0
1000 counts: 3 5 0
1250 counts: 0 8 0
1500 counts: 3 8 0
1750 counts: 1 10 0
2000 counts: 2 11 0
```

Observe that the monitor has reported the expected number of active, completed, and overdue tasks after 1500 ms, further indicating the correct execution of test bench 1.

3.3 Test Bench #2 Results

The sequence of events during test bench 2 can be seen in Table 8. Expected times are taken from the Gantt chart in Figure 9. A screenshot of the console output during the execution run can be seen in Figure 13.

Table 8. Table of event times during test bench 2

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 completed	95	95
5	Task 2 completed	245	245
6	Task 1 released	250	250
7	Task 1 completed	345	345
8	Task 2 released	500	500
9	Task 1 released	500	500
10	Task 3 completed	590	590
11	Task 1 completed	685	685
12	Task 3 released	750	750
13	Task 1 released	750	750
14	Task 2 completed	835	835
15	Task 1 completed	930	930
16	Task 2 released	1000	1000
17	Task 1 released	1000	1000
18	Task 1 completed	1095	1095
19	Task 1 released	1250	1250
20	Task 3 completed	1275	1275
21	Task 2 completed	1425	1425
22	Task 3 released	1500	1500
23	Task 2 released	1500	1500
24	Task 1 released	1500	1500
25	Task 1 overdue	1520	1500

Figure 13. Screenshot of console output when monitoring test bench 2 for event times.

```
Port 0
0 R 1
0 R 2
0 R 3
95 C 1
245 C 2
250 R 1
345 C 1
500 R 2
500 R 1
590 C 3
685 C 1
750 R 3
750 R 1
835 C 2
930 C 1
1000 R 2
1000 R 1
1095 C 1
1250 R 1
1275 C 3
1425 C 2
1500 R 3
1500 R 2
1500 R 1
1520 O 1
1615 C 1
1750 R 1
1765 C 2
1860 C 1
2000 R 2
2000 R 1
2110 C 3
2205 C 1
```

The measured times meet the expected times for all cases, except for the detection of the overdue task 1 at t = 1500. Notice that we observe the sixth instance of task 1 as overdue at 1520 ms, that is, after the task has fully completed its execution. This can also be seen in the Figure 14 below, and indicates a limitation of the system: that it cannot detect overdue tasks until after they have finished executing. This limitation is expanded upon in section 4.2.

As stated in section 3.2, the release times of tasks do not maintain order within a single tick due to overhead created from timer callback functions from shorter periods of tasks 1 and 2 relative to task 3. Both these limitations are discussed in section 4. Overall the above table and figure indicate correct execution of test bench 2.

The observed number of active, completed, and overdue tasks after one hyperperiod of test bench 2 can be seen in Table 9; the expected values are derived from the Gantt chart in Figure 9. The observed number of active, completed, and overdue tasks every 250 ms during test bench one can be seen in Figure 14.

Table 9. Number of active, completed, and overdue tasks after one hyperperiod (1500 ms) in test bench 2.

	Measured	Expected
Number of active DD-Tasks	4	4
Number of completed DD-Tasks	10	10
Number of overdue DD-Tasks	0	1

Figure 14. Screenshot of console output of number of active, completed, and overdue tasks in test bench 2, at 250 ms intervals.

```
Port 0
250 counts: 2 2 0
500 counts: 3 3 0
750 counts: 3 5 0
1000 counts: 3 7 0
1250 counts: 3 8 0
1500 counts: 4 10 0
1750 counts: 3 11 1
2000 counts: 3 13 1
2250 counts: 3 15 1
```

We can observe that the supposedly overdue task 1 at 1500 ms is not accounted overdue at 1500 ms; however, this is the expected behaviour given our design, since the system cannot detect overdue tasks until after they finish executing. The DDS_Task deems this task as overdue at 1520 ms, so at the next MONITOR_REQUEST_PERIOD, i.e., at 1750 ms, we can see the task as overdue in the updated DD-Task lists in Figure 14.

3.4 Test Bench #3 Results

Table 10. Table of event times during test bench 3

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 completed	100	100
5	Task 2 completed	300	300
6	Task 1 released	500	500
7	Task 2 released	500	500
8	Task 3 released	500	500
9	Task 3 completed	500	500
10	Task 1 completed	600	600
11	Task 2 completed	800	800
12	Task 1 released	1000	1000
13	Task 2 released	1000	1000
14	Task 3 released	1000	1000
15	Task 3 completed	1000	1000
16	Task 1 completed	1100	1100
17	Task 2 completed	1300	1300
18	Task 1 released	1500	1500
19	Task 2 released	1500	1500
20	Task 3 released	1500	1500
21	Task 3 completed	1500	1500

Figure 15. Screenshot of console output when monitoring test bench 3 for event times.

```
Port 0
0 R 1
0 R 2
0 R 3
100 C 1
300 C 2
500 R 1
500 R 2
500 R 3
500 C 3
600 C 1
800 C 2
1000 R 1
1000 R 2
1000 R 3
1000 C 3
1100 C 1
1300 C 2
1500 R 1
1500 R 2
1500 R 3
1500 C 3
1600 C 1
1800 C 2
2000 R 1
2000 R 2
2000 R 3
2000 C 3
2100 C 1
```

The expected times match the measured times in all cases, indicating the correct execution of test bench 3. The completion times of task 3 appears to be outputted after the release of all tasks in the same tick. This is due to DD_Task_Generator_Task having higher priority than the user-defined F-Tasks. At multiples of 500 ms, instances of task 3 finish execution, but at the same time the timer callback for release times of all the tasks also kicks in, which takes over the FreeRTOS scheduler.

The observed number of active, completed, and overdue tasks after three hyper periods of test bench 3 can be seen in Table 11; the expected values are derived from the Gantt chart in Figure

10. The observed number of active, completed, and overdue tasks every 250 ms during test bench one can be seen in Figure 16.

Table 11. Number of active, completed, and overdue tasks after three hyper periods (1500 ms) in test bench 3.

	Measured	Expected
Number of active DD-Tasks	4	3
Number of completed DD-Tasks	9	8
Number of overdue DD-Tasks	0	0

Figure 16. Screenshot of console output of number of active, completed, and overdue tasks in test bench 3

```
Port 0
250 counts: 2 1 0
500 counts: 4 2 0
750 counts: 2 4 0
1000 counts: 4 5 0
1250 counts: 2 7 0
1500 counts: 4 8 0
1750 counts: 2 10 0
2000 counts: 4 11 0
2250 counts: 2 13 0
2500 counts: 4 14 0
```

Observe that at time t = 1500, the monitor observes 4 active tasks, and 8 completed tasks, where we would expect 3 active tasks, and 9 completed tasks; this is because the Monitor_Task has a higher priority than the user-defined F-Tasks, so the Monitor_Task's request for the list sizes is processed by the DDS_Task before the F-Task's message indicating task completion. This demonstrates the limitation of the system that if a task completes in the same tick that the Monitor_Task requests the list sizes, the task will be observed as active; this limitation is expanded upon in section 4. Test bench 3 has a 100% CPU utilization efficiency, so it has no overdue tasks, as is suggested by the fifth column in the figure above.

3.5 Discussion of DDS performance and Design

The DD Scheduler performs quite well for the duration of 1500 ms: all test benches executed as expected with no observed delays in release times or completion times. The ticks are computed as milliseconds since tick rate = 1000 Hz, so 1 tick = 1 millisecond.

Priority of the running user-defined F-Task is kept higher than other F-tasks, so the FreeRTOS scheduler can execute the high priority F-task only. Our final prioritization scheme differed from the prioritization scheme in the initial design. We found that by assigning the same priority to the DD_Task_Generator_Task and the Monitor_Task, it was possible that the DD_Task_Generator_Task could be forced to wait for the Monitor_Task, since tasks of the same priority execute in a round-robin manner. Then to prevent this, we raised the priority of the DD_Task_Generator_Task above that of the Monitor_Task, thereby minimizing the amount of delay that gets introduced into task release events. We also raised the priority of the DD_Task_Generator_Task above that of the DDS_Task, to further minimize the amount of delay introduced to task release events; this required raising the priority of the timers as well, in order that their priority would be still higher than that of the DD_Task_Generator_Task.

To enhance code readability, interface functions for Monitor_Task, such as Print_Event_Time and Print_DD_Task_List_Sizes were implemented. These functions are used to direct the Monitor_Task to print a release, completion, or overdue times and list sizes respectively. Define directives (#define) were fairly used for many parameters, such as task periods and execution times. This also helps to improve readability and makes it easier to quickly change parameters for testing. These #define statements can be seen in the code block starting at line 96. The introduction of the list interface functions also improved the readability and modularity of the code.

In our design, the user-defined F-Tasks are continually created and destroyed by the DDS_Task, by calling xTaskCreate() and vTaskDelete() for every DD_Task. We chose this design since it simplifies the logic of the user-defined F-Tasks: if we instead used the same instance of a user-defined F-Task for every corresponding DD_Task (ie by calling each of xTaskCreate() and vTaskDelete() only once for that F-Task), we would have to implement logic to reset the execution time of that F-Task for every corresponding DD_Task. Our approach introduces the additional overhead of repeatedly creating and destroying F-Task instances; however, since it was already difficult to implement the correct execution time logic for the user-defined F-Tasks, we decided to go ahead with our approach, and modify it only if necessary. Ultimately our approach delivered the necessary performance.

To accommodate all tasks, queues, and structs, the default heap size was increased 14 folds. Although the timer callback and print statements overhead causes delays in task executions, this delay is suppressed by reducing the length of output strings, as shown in figures from Test Bench Results. This allows to postpone the delay well beyond the hyper period of 1500 ms.

We observed that if the system is left to run for several hyper periods, delay is gradually introduced into the task completion times, in increments of 1 ms. This limitation is expanded upon in section 4. Note, however, that the system is capable of executing a single complete hyperperiod of each test bench while simultaneously printing all event times as well as the list sizes (at intervals of 250 ms), without introducing any observable delay. This is the result of a number of optimizations made in our design: first, the minimization of delay in release times by the prioritization scheme discussed above; second, the inclusion of the active_count, completed_count, and overdue_count variables in the DDS_Task (which reduce the overhead of computing and obtaining these values); and third, by minimizing the lengths of the strings printed by the Monitor_Task.

3.6 Handling aperiodic tasks

As per the project specification, the system is designed to handle aperiodic tasks. The Enlist_Task() function can be used as-is to add DD_Tasks of type APERIODIC to the active_list; this function always sends aperiodic tasks to the back of the list, so no periodic task ever as to wait for an aperiodic task. This is appropriate, since aperiodic tasks have soft deadlines, while periodic tasks have hard deadlines. Furthermore, the Enlist_Task_By_Deadline() function never sorts any periodic task behind and aperiodic task, ensuring the desired behavior. To introduce an aperiodic task into the system, the DD_Task_Generator_Task could be extended to use an additional one-shot timer to release an aperiodic task to the DDS_Task after some duration, by calling Release_DD_Task() with task_type = APERIODIC.

4. Limitations and Possible Improvements

4.1 Known Bug

Printing all of release times, completion times and overdue times eventually causes delay in completion times by increments of 1 ms, which starts a domino effect on the proceeding times. Reducing the use of printf statements delays the start of this domino effect at a later time period. This bug appears to be prevalent when more print statements are used, i.e., when testing and/or demonstrating the program. However, since the monitor task only ever needs to print the number of active, completed and overdue DD-Tasks, print statements can be kept to a minimum and this overhead problem no longer persists over a hyper period of 1500 ms.

4.2 Limitations

- Overdue tasks are not detected until the task completes its execution. For instance, if a task has a period of 100 ms, execution time of 40 ms and the task starts executing at 80 ms. This task would be overdue the instant it goes past 100 ms. In such cases, our DDS scheduler would detect this overdue only after it has fully completed its execution, i.e., at 120 ms assuming no interruption. This limitation could be overcome by comparing task period against current period at every clock tick. Although our current design approach has this limitation, it saves us a lot of time and memory by reducing these time comparisons when tasks are not overdue, thereby allowing the monitor to print the list sizes more frequently. Our design approach was implemented considering this tradeoff.

- If the monitor obtains the task list counts in the same tick as a user-defined F- task completes at, the monitor will detect that completed task as still being active. This happens since the monitor has a higher priority than the user-defined F- tasks, so the DDS responds to the monitor first with the lists and then updates the number of active and completed task lists. Setting a higher priority for the monitor task is important, so the monitor never has to wait for user-defined F-tasks to finish before sending requests to the DDS. Note that the finished tasks will always be detected as complete or overdue in the next report by the monitor.

- The release times of the user-defined F-tasks having the shortest period gradually get delayed relative to the others, due to the overhead introduced by the timer callback functions. Since tasks with shortest periods are executed more often, hence the timer callback functions, for these tasks, are called more often which results in this delay.

4.3 Possible Improvements

Currently, the DDS scheduler creates 3 separate user defined F-tasks for their execution. This means 3 separate functions that all have similar functionality. This could lead to a bottleneck if more user defined F-Tasks are to be scheduled; it also introduces unnecessary code duplication. Instead, a single F-task can be defined with the execution time passed as a parameter to the task when it is instantiated. This can enhance the compactness of the program as the same F-task definition could be used for all user-defined tasks.

We could improve the error handling of the system. Currently, the codebase has no implementations to handle run-time errors. For any errors during data transmission through queues, the program prints a predetermined or static message to the console. In future, built-in functions like *perror()* and *strerror()* could be used to produce error messages, if any, thereby facilitating in taking appropriate actions. These functions could be implemented in several events in the program. For example, ensuring correct hardware configuration, ensuring data transmission between queues, etc.

We could experiment with possible approaches to eliminate the additional overhead introduced for tasks having shorter periods (due to the timer callback functions); this could involve the use of a single timer with a variable period to trigger the release of all periodic tasks.

5. Summary

The objective of the project was successfully completed. The Deadline Driven Scheduler was built on top of the existing FreeRTOS task scheduler. The design solution discussed meets all the functional and technical requirements. A few limitations are identified, but these limitations are preferred over their trade-offs.

The project was completed over a period of 3 weeks. From our experience, we have understood that setting apart some time in the beginning to discuss, plan and architect a tentative design can help deepen understanding of the problem space and speed up the implementation stage. We highly recommend having a solid design overview before moving into coding for this project.

6. References

[1]. STMicroelectronics. *STM32F4xx advanced Arm-based 32-bit MCUs Reference Manual*, 2019. [Online].

[2]. Department of Electrical and Computer Engineering, University of Victoria. *Lab Manual: ECE 455: Real Time Computer Systems Design Project*, Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, 2019. [Online].

[3] Amazon Web Services. “FreeRTOS API categories”. [Online] Available: https://www.freertos.org/a00106.html. [accessed Apr 5., 2022].

Appendix (with source code)

```c
//// ECE 455
// Lab project 2
// Spencer Davis V00759537
// Mustafa Wasif V00890184

// Standard includes.
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "stm32f4_discovery.h"
#include <inttypes.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"

// Struct for packaging messages on DDS_Queue.
struct DDS_Message
{
    uint16_t message_type;
    uint16_t ud_task;
    TaskHandle_t t_handle;
    uint16_t task_type;
    uint32_t task_id;
    TickType_t release_time;
    TickType_t absolute_deadline;
    TickType_t completion_time;
};

// Struct for packaging message on Monitor_Queue.
struct Monitor_Message
{
    uint16_t message_type;
    uint16_t ud_task;
    TickType_t time;
    uint16_t active_count;
    uint16_t completed_count;
    uint16_t overdue_count;
};

// Struct representing a DD task. One node in a DD task list.
struct DD_Task
{
    uint16_t ud_task;
    TaskHandle_t t_handle;
    uint16_t task_type;
    uint32_t task_id;
    uint32_t release_time;
    uint32_t absolute_deadline;
    uint32_t completion_time;
    bool started; // Used to determine whether to call xTaskCreate.
    struct DD_Task *next;
};

// Task functions.
static void DD_Task_Generator_Task(void *pvParameters);
static void Monitor_Task(void *pvParameters);
static void DDS_Task(void *pvParameters);
static void Print_Event_Time(uint16_t event_type, uint16_t ud_task, TickType_t time);
static void Print_DD_Task_List_Sizes(uint16_t active_count, uint16_t completed_count, uint16_t overdue_count);
static void Release_DD_Task(uint16_t ud_task, TaskHandle_t t_handle, uint16_t task_type, uint32_t task_id, TickType_t
release_time, TickType_t absolute_deadline);
static void Complete_DD_Task(TickType_t completion_time);
static void Request_DD_Task_Counts();
static void Enlist_Task_By_Deadline(struct DD_Task** list_head, struct DD_Task* dd_task);
static void Enlist_Task(struct DD_Task** list_head, struct DD_Task* task);
static struct DD_Task* Delist_Task(struct DD_Task** list_head);
static void Schedule_Next_Task(struct DD_Task* list_head);
static void UD_Task_1( void *pvParameters );
static void UD_Task_2( void *pvParameters );
static void UD_Task_3( void *pvParameters );

// Queue handles.
```

```c
xQueueHandle xTask_Generator_Queue = 0;
xQueueHandle xDDS_Queue = 0;
xQueueHandle xMonitor_Queue = 0;

// Timer handles.
xTimerHandle xUD_Task_1_Release_Timer = 0;
xTimerHandle xUD_Task_2_Release_Timer = 0;
xTimerHandle xUD_Task_3_Release_Timer = 0;
xTimerHandle xMonitor_Request_Timer = 0;

// Timer callback functions.
static void Enqueue_UD_Task_1_Release_Timer_Flag(xTimerHandle pxTimer);
static void Enqueue_UD_Task_2_Release_Timer_Flag(xTimerHandle pxTimer);
static void Enqueue_UD_Task_3_Release_Timer_Flag(xTimerHandle pxTimer);
static void Enqueue_Monitor_Request(xTimerHandle pxTimer);
static void Enqueue_Monitor_Request(xTimerHandle pxTimer);

// Defines.
#define mainQUEUE_LENGTH 100
#define QUEUE_WAIT_TIME 100
#define UD_TASK_1 1
#define UD_TASK_2 2
#define UD_TASK_3 3
#define UD_TASK_1_PERIOD 500
#define UD_TASK_2_PERIOD 500
#define UD_TASK_3_PERIOD 500
#define UD_TASK_1_EXECUTION_TIME 100
#define UD_TASK_2_EXECUTION_TIME 200
#define UD_TASK_3_EXECUTION_TIME 200
#define TASK_RELEASE 0
#define TASK_COMPLETE 1
#define TASK_OVERDUE 2
#define MONITOR_REQUEST_TASK_COUNTS 3
#define MONITOR_REQUEST_TIMER_EXPIRED 4
#define PERIODIC 0
#define APERIODIC 1
#define MONITOR_REQUEST_PERIOD 250

// Globals for output control.
bool print_release_times = false;
bool print_completion_times = false;
bool print_overdue_times = false;
bool print_dd_task_list_sizes = true;

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

int main(void)
{
    // Do we need this?:
    // NVIC_SetPriorityGrouping( 0 );

    // Create tasks and priorities.
    xTaskCreate( DD_Task_Generator_Task, "DD_Task_Generator_Task", configMINIMAL_STACK_SIZE, NULL, 5, NULL);
    xTaskCreate( DDS_Task, "DDS_Task", configMINIMAL_STACK_SIZE, NULL, 4, NULL);
    xTaskCreate( Monitor_Task, "Monitor_Task", configMINIMAL_STACK_SIZE, NULL, 3, NULL);

    // Create queues.
    xTask_Generator_Queue = xQueueCreate(mainQUEUE_LENGTH, sizeof( uint16_t ) );
    xDDS_Queue = xQueueCreate( mainQUEUE_LENGTH, sizeof( struct DDS_Message ) );
    xMonitor_Queue = xQueueCreate( mainQUEUE_LENGTH, sizeof( struct Monitor_Message ));

    // Add queues to the registry, for the benefit of kernel aware debugging.
    vQueueAddToRegistry( xTask_Generator_Queue, "Task_Generator_Queue" );
    vQueueAddToRegistry( xDDS_Queue, "DDS_Queue" );
    vQueueAddToRegistry( xMonitor_Queue, "Monitor_Queue" );

    // Create timers.
    xUD_Task_1_Release_Timer = xTimerCreate( "UD_Task_1_Release_Timer",
                                             pdMS_TO_TICKS(UD_TASK_1_PERIOD),
                                             pdTRUE,
                                             (void *) 0,
                                             Enqueue_UD_Task_1_Release_Timer_Flag);

    xUD_Task_2_Release_Timer = xTimerCreate( "UD_Task_2_Release_Timer",
                                             pdMS_TO_TICKS(UD_TASK_2_PERIOD),
                                             pdTRUE,
                                             (void *) 0,
                                             Enqueue_UD_Task_2_Release_Timer_Flag);

```

```c
    xUD_Task_3_Release_Timer = xTimerCreate( "UD_Task_3_Release_Timer",
                                              pdMS_TO_TICKS(UD_TASK_3_PERIOD),
                                              pdTRUE,
                                              (void *) 0,
                                              Enqueue_UD_Task_3_Release_Timer_Flag);

    xMonitor_Request_Timer = xTimerCreate(   "Monitor_Request_timer",
                                              pdMS_TO_TICKS(MONITOR_REQUEST_PERIOD),
                                              pdTRUE,
                                              (void *) 0,
                                              Enqueue_Monitor_Request);

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

    return 0;
}

/*-----------------------------------------------------------*/
// DD Task Generator Task.

static void DD_Task_Generator_Task( void *pvParameters )
{
    // Prepare task handles.
    TaskHandle_t UD_Task_1_Handle = NULL;
    TaskHandle_t UD_Task_2_Handle = NULL;
    TaskHandle_t UD_Task_3_Handle = NULL;

    // Prepare data items.
    uint16_t ud_task = -1;
    uint16_t ud_task_id = 0;
    TickType_t release_time;
    TickType_t absolute_deadline;

    // Start task timers.
    xTimerStart(xUD_Task_1_Release_Timer, UD_TASK_1_PERIOD);
    xTimerStart(xUD_Task_2_Release_Timer, UD_TASK_2_PERIOD);
    xTimerStart(xUD_Task_3_Release_Timer, UD_TASK_3_PERIOD);

    // Perform initial release of all tasks.
    Enqueue_UD_Task_1_Release_Timer_Flag(xUD_Task_1_Release_Timer);
    Enqueue_UD_Task_2_Release_Timer_Flag(xUD_Task_2_Release_Timer);
    Enqueue_UD_Task_3_Release_Timer_Flag(xUD_Task_3_Release_Timer);

    while(1)
    {
        // Wait for flag from any task timer.
        xQueueReceive(xTask_Generator_Queue, &ud_task, portMAX_DELAY);

        switch(ud_task)
        {
                case UD_TASK_1:
                    // Inform DDS of task 1 release.
                    release_time = xTaskGetTickCount();
                    absolute_deadline = release_time + pdMS_TO_TICKS(UD_TASK_1_PERIOD);
                    Release_DD_Task(UD_TASK_1, UD_Task_1_Handle, PERIODIC, ud_task_id, release_time, absolute_deadline);
                    break;
                case UD_TASK_2:
                    // Inform DDS of task 2 release.
                    release_time = xTaskGetTickCount();
                    absolute_deadline = release_time + pdMS_TO_TICKS(UD_TASK_2_PERIOD);
                    Release_DD_Task(UD_TASK_2, UD_Task_2_Handle, PERIODIC, ud_task_id, release_time, absolute_deadline);
                    break;
                case UD_TASK_3:
                    // Inform DDS of task 3 release.
                    release_time = xTaskGetTickCount();
                    absolute_deadline = release_time + pdMS_TO_TICKS(UD_TASK_3_PERIOD);
                    Release_DD_Task(UD_TASK_3, UD_Task_3_Handle, PERIODIC, ud_task_id, release_time, absolute_deadline);
                    break;
        }

        // Increment id.
        ud_task_id++;
    }
}

// Task generator timer callback functions.
static void Enqueue_UD_Task_1_Release_Timer_Flag(xTimerHandle pxTimer)
{
```

```
    // Task 1 period timer restarts by autoreload.

    uint16_t flag = UD_TASK_1;
    xQueueSend(xTask_Generator_Queue, &flag, UD_TASK_1_PERIOD);
}

static void Enqueue_UD_Task_2_Release_Timer_Flag(xTimerHandle pxTimer)
{
    // Task 2 period timer restarts by autoreload.

    uint16_t flag = UD_TASK_2;
    xQueueSend(xTask_Generator_Queue, &flag, UD_TASK_2_PERIOD);
}

static void Enqueue_UD_Task_3_Release_Timer_Flag(xTimerHandle pxTimer)
{
    // Task 2 period timer restarts by autoreload.

    uint16_t flag = UD_TASK_3;
    xQueueSend(xTask_Generator_Queue, &flag, UD_TASK_3_PERIOD);
}

/*-----------------------------------------------------------*/
// Monitor Task.

static void Monitor_Task(void *pvParameters)
{
    struct Monitor_Message monitor_message;
    TickType_t current_time;

    // Start monitor request timer.
    xTimerStart(xMonitor_Request_Timer, MONITOR_REQUEST_PERIOD);

    while(1)
    {
        // Wait for message on monitor queue.
        xQueueReceive(xMonitor_Queue, &monitor_message, portMAX_DELAY);

        switch(monitor_message.message_type)
        {
            case TASK_RELEASE: // We have a release time to print.
                if (print_release_times)
                {
                    printf("%d %s %d\n", (int) monitor_message.time, "R", monitor_message.ud_task);

                }
                break;
            case TASK_COMPLETE: // We have a completion time to print.
                if (print_completion_times)
                {
                    printf("%d %s %d\n", (int) monitor_message.time, "C", monitor_message.ud_task);
                }
                break;
            case TASK_OVERDUE: // We have an overdue time to print.
                if (print_overdue_times)
                {
                    printf("%d %s %d\n", (int) monitor_message.time, "O", monitor_message.ud_task);
                }
                break;
            case MONITOR_REQUEST_TASK_COUNTS: // We have a received the requested list counts.
                if (print_dd_task_list_sizes)
                {
                    current_time = xTaskGetTickCount();
                    printf("%d %s %d %d %d\n", (int) current_time, "counts:", monitor_message.active_count, monitor_message.completed_count, monitor_message.overdue_count);
                }
                break;
            case MONITOR_REQUEST_TIMER_EXPIRED: // We need to request list counts.
                Request_DD_Task_Counts();
                break;
        }
    }
}

// Monitor task interface functions.
// Direct monitor to print a release, completion, or overdue time.
static void Print_Event_Time(uint16_t event_type, uint16_t ud_task, TickType_t time)
{
    struct Monitor_Message monitor_message;
```

```
    monitor_message.message_type = event_type;
    monitor_message.ud_task = ud_task;
    monitor_message.time = time;

    xQueueSend(xMonitor_Queue, &monitor_message, pdMS_TO_TICKS(QUEUE_WAIT_TIME));
}

// Direct monitor to print list sizes.
static void Print_DD_Task_List_Sizes(uint16_t active_count, uint16_t completed_count, uint16_t overdue_count)
{
    struct Monitor_Message monitor_message;
    monitor_message.message_type = MONITOR_REQUEST_TASK_COUNTS;
    monitor_message.active_count = active_count;
    monitor_message.completed_count = completed_count;
    monitor_message.overdue_count = overdue_count;

    xQueueSend(xMonitor_Queue, &monitor_message, pdMS_TO_TICKS(QUEUE_WAIT_TIME));
}

// Monitor request timer callback function.
static void Enqueue_Monitor_Request(xTimerHandle pxTimer)
{
    // Monitor request timer restarts by autoreload.

    uint16_t flag = MONITOR_REQUEST_TIMER_EXPIRED;
    xQueueSend(xMonitor_Queue, &flag, pdMS_TO_TICKS(QUEUE_WAIT_TIME));
}

/*-----------------------------------------------------------*/
// DDS Task.

static void DDS_Task( void *pvParameters )
{
    // Prepare task lists.
    struct DD_Task *active_list_head = NULL;
    struct DD_Task *completed_list_head = NULL;
    struct DD_Task *overdue_list_head = NULL;
    uint16_t active_count = 0;
    uint16_t completed_count = 0;
    uint16_t overdue_count = 0;

    struct DDS_Message dds_message;

    while(1)
    {
        // Wait for message on dds queue.
        xQueueReceive(xDDS_Queue, &dds_message, portMAX_DELAY);

        if (dds_message.message_type == TASK_RELEASE)
        {
            // Prepare struct defining task.
            struct DD_Task *dd_task = (struct DD_Task*) pvPortMalloc(sizeof(struct DD_Task));
            dd_task->ud_task = dds_message.ud_task;
            dd_task->t_handle = dds_message.t_handle;
            dd_task->task_type = dds_message.task_type;
            dd_task->task_id = dds_message.task_id;
            dd_task->release_time = dds_message.release_time;
            dd_task->absolute_deadline = dds_message.absolute_deadline;
            dd_task->started = false;
            dd_task->next = NULL;

            // Add task to active list.
            if (dd_task->task_type == PERIODIC) Enlist_Task_By_Deadline(&active_list_head, dd_task); // If periodic, sort by deadline.
            else Enlist_Task(&active_list_head, dd_task); // If aperiodic, send to back of list.
            active_count++;

            // Start next task.
            Schedule_Next_Task(active_list_head);

            if (print_release_times) Print_Event_Time(TASK_RELEASE, dds_message.ud_task, dds_message.release_time);
        }
        else if (dds_message.message_type == TASK_COMPLETE)
        {
            // Set completion_time of head.
            active_list_head->completion_time = dds_message.completion_time;

            // Delete the F-task instance corresponding to head.
            vTaskDelete(active_list_head->t_handle);
```

```c

            // Check if overdue, move to appropriate list, and delete from active list.
            if (active_list_head->completion_time <= active_list_head->absolute_deadline)
            {
                if (print_completion_times) Print_Event_Time(TASK_COMPLETE, active_list_head->ud_task, active_list_head->completion_time);
                Enlist_Task(&completed_list_head, Delist_Task(&active_list_head));
                completed_count++;
            }
            else
            {
                if (print_overdue_times) Print_Event_Time(TASK_OVERDUE, active_list_head->ud_task, active_list_head->completion_time);
                Enlist_Task(&overdue_list_head, Delist_Task(&active_list_head));
                overdue_count++;
            }
            active_count--;

            // Schedule next.
            Schedule_Next_Task(active_list_head);
        }
        else if (dds_message.message_type == MONITOR_REQUEST_TASK_COUNTS)
        {
            Print_DD_Task_List_Sizes(active_count, completed_count, overdue_count);
        }
    }
}

// DDS helper functions.
static void Enlist_Task_By_Deadline(struct DD_Task** list_head, struct DD_Task* dd_task)
{
    // Check if list empty.
    if (*list_head == NULL)
    {
        *list_head = dd_task;
    }
    else
    {
        // Check if must add to front.
        if ((*list_head)->task_type == APERIODIC || (*list_head)->absolute_deadline > dd_task->absolute_deadline)
        {
            dd_task->next = (*list_head);
            *list_head = dd_task;
        }
        // Else sort into list by ascending absolute_deadline.
        else
        {
            struct DD_Task *current = *list_head;
            while (current->next != NULL && current->next->task_type != APERIODIC && current->next->absolute_deadline <= dd_task->absolute_deadline)
            {
                current = current->next;
            }
            dd_task->next = current->next;
            current->next = dd_task;
        }
    }
}

static void Enlist_Task(struct DD_Task** list_head, struct DD_Task* dd_task)
{
    if (*list_head == NULL)
    {
        *list_head = dd_task;
    }
    else
    {
        struct DD_Task *current = *list_head;
        while (current->next != NULL) current = current->next;
        current->next = dd_task;
    }
}

static struct DD_Task* Delist_Task(struct DD_Task** list_head)
{
    if (*list_head == NULL) return NULL;

    struct DD_Task* temp = *list_head;
    *list_head = (*list_head)->next;
```

```c
    temp->next = NULL;
    return temp;
}

static void Schedule_Next_Task(struct DD_Task* list_head)
{
    if (list_head == NULL) return;
    if (!list_head->started)
    {
        switch(list_head->ud_task)
        {
            case UD_TASK_1:
                xTaskCreate(UD_Task_1, "UD_Task_1", configMINIMAL_STACK_SIZE, NULL, 1, &list_head->t_handle);
                break;
            case UD_TASK_2:
                xTaskCreate(UD_Task_2, "UD_Task_2", configMINIMAL_STACK_SIZE, NULL, 1, &list_head->t_handle);
                break;
            case UD_TASK_3:
                xTaskCreate(UD_Task_3, "UD_Task_3", configMINIMAL_STACK_SIZE, NULL, 1, &list_head->t_handle);
                break;
        }
        list_head->started = true;
    }

    // Raise priority of head task.
    vTaskPrioritySet(list_head->t_handle, 2);

    // Lower priority of head task in the list, in case it had been running.
    if (list_head->next != NULL && list_head->next->started)
    {
        vTaskPrioritySet(list_head->next->t_handle, 1);
    }
}

// DDS interface functions.

static void Release_DD_Task(uint16_t ud_task, TaskHandle_t t_handle, uint16_t task_type, uint32_t task_id, TickType_t release_time, TickType_t absolute_deadline)
{
    // Create task release message struct.
    struct DDS_Message task_release_message;
    task_release_message.message_type = TASK_RELEASE;
    task_release_message.ud_task = ud_task;
    task_release_message.t_handle = t_handle;
    task_release_message.task_type = task_type;
    task_release_message.task_id = task_id;
    task_release_message.release_time = release_time;
    task_release_message.absolute_deadline = absolute_deadline;

    // Add message to DDS queue.
    xQueueSend(xDDS_Queue, &task_release_message, pdMS_TO_TICKS(QUEUE_WAIT_TIME));
}


static void Complete_DD_Task(TickType_t completion_time)
{
    // Prepare task complete message struct.
    struct DDS_Message task_complete_message;
    task_complete_message.message_type = TASK_COMPLETE;
    task_complete_message.completion_time = completion_time;

    // Add message to DDS queue.
    xQueueSend(xDDS_Queue, &task_complete_message, pdMS_TO_TICKS(QUEUE_WAIT_TIME));
}

static void Request_DD_Task_Counts()
{
    // Prepare task count message struct.
    struct DDS_Message task_counts_request_message;
    task_counts_request_message.message_type = MONITOR_REQUEST_TASK_COUNTS;
    // Add message to DDS queue.
    xQueueSend(xDDS_Queue, &task_counts_request_message, pdMS_TO_TICKS(QUEUE_WAIT_TIME));
}

/*-----------------------------------------------------------*/
// User-defined f-tasks.

static void UD_Task_1( void *pvParameters )
{
```

```c
    TickType_t execution_time_ticks = pdMS_TO_TICKS(UD_TASK_1_EXECUTION_TIME);
    TickType_t ticks_elapsed = 0;
    TickType_t last_time_ticks = xTaskGetTickCount();
    TickType_t current_time_ticks;

    while(1)
    {
        // Execute "user-defined code" for duration specified by UD_TASK_1_EXECUTION_TIME.
        while(1)
        {
            current_time_ticks = xTaskGetTickCount();
            if (current_time_ticks != last_time_ticks)
            {
                ticks_elapsed++;
                last_time_ticks = current_time_ticks;
            }
            if (ticks_elapsed >= execution_time_ticks) break;
        }
        Complete_DD_Task(current_time_ticks);
    }
}

static void UD_Task_2( void *pvParameters )
{
    TickType_t execution_time_ticks = pdMS_TO_TICKS(UD_TASK_2_EXECUTION_TIME);
    TickType_t ticks_elapsed = 0;
    TickType_t last_time_ticks = xTaskGetTickCount();
    TickType_t current_time_ticks;

    while(1)
    {
        // Execute "user-defined code" for duration specified by UD_TASK_2_EXECUTION_TIME.
        while(1)
        {
            current_time_ticks = xTaskGetTickCount();
            if (current_time_ticks != last_time_ticks)
            {
                ticks_elapsed++;
                last_time_ticks = current_time_ticks;
            }
            if (ticks_elapsed >= execution_time_ticks) break;
        }
        Complete_DD_Task(current_time_ticks);
    }
}

static void UD_Task_3( void *pvParameters )
{
    TickType_t execution_time_ticks = pdMS_TO_TICKS(UD_TASK_3_EXECUTION_TIME);
    TickType_t ticks_elapsed = 0;
    TickType_t last_time_ticks = xTaskGetTickCount();
    TickType_t current_time_ticks;

    while(1)
    {
        // Execute "user-defined code" for duration specified by UD_TASK_3_EXECUTION_TIME.
        while(1)
        {
            current_time_ticks = xTaskGetTickCount();
            if (current_time_ticks != last_time_ticks)
            {
                ticks_elapsed++;
                last_time_ticks = current_time_ticks;
            }
            if (ticks_elapsed >= execution_time_ticks) break;
        }
        Complete_DD_Task(current_time_ticks);
    }
}


/*-----------------------------------------------------------*/
// Built-in functions.

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( ;; );
    printf("%s\n", "malloc failed");
}
/*-----------------------------------------------------------*/

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. */
    }
}
```