ECE 455

Project 2: Deadline-Driven Scheduler

Deadline-Driven Scheduler Implementation report

Course: Real Time Computer Systems Design Project

Term: Spring 2026

Instructor: Amirali Baniasadi

Group: 01

Project Team

John Soliman V01028377
David Emelu V01025755

Department of Electrical & Computer Engineering

March 29, 2026

Contents

1 Introduction **1**

2 Design Document **2**

3 Design Solution **3**

3.1 System Overview . . . 3
3.2 F-Task Priority Levels . . . 3
3.3 DDS Core Function Implementation Details . . . 4
3.4 Core Implementation . . . 5
3.5 Periodic DD-Task Generator Logic . . . 7
3.6 Active List Sorted Insertion Flowchart . . . 8
3.7 DDS Execution and EDF Scheduling Flowchart . . . 9
3.8 Design Justifications . . . 10

4 Discussion **11**

4.1 Test Bench 1: DDS Event Table . . . 11
4.2 Test Bench 2: Monitor Task Output . . . 12
4.3 Test Bench 3: Boundary Utilization Analysis . . . 13
4.4 DDS Scheduler Performance . . . 14
4.5 Testing Methodology and Known Issues . . . 14

5 Limitations and Possible Improvements **14**

6 Summary **14**

A Source Code **16**

List of Figures

1 Pre-implementation design document. . . . 2
2 Complete system overview: F-Tasks, queues, and DDS interface function calls. . . . 3
3 Task Generator continuous loop and release algorithm. . . . 7
4 DD-Task sorting algorithm: `insert_node()` with static memory allocation and sorted insertion. . . . 8
5 DD Scheduler flowchart: EDF main loop with deadline-driven timeout detection. . . . 9
6 DDS events for Test Bench #1 over one hyper-period (1500 ms). . . . 11
7 Monitor Task output for Test Bench #2 after one hyper-period (1500 ms). . . . 12
8 Monitor Task output for Test Bench #3 after one hyper-period (500 ms). . . . 13

List of Tables

1 F-Task list with FreeRTOS priority levels. . . . 4

1 Introduction

This report documents the design and implementation of a Deadline-Driven Scheduler (DDS) for a real-time embedded system running FreeRTOS v9.0 on the STM32F4 Discovery board. The DDS implements the Earliest Deadline First (EDF) algorithm to manage a set of periodic real-time tasks with hard execution deadlines. Because FreeRTOS does not natively support EDF scheduling, the DDS operates as a layer above the existing fixed-priority preemptive scheduler. The core mechanism is dynamic priority manipulation: the Deadline-Driven Task (DD-Task) with the nearest absolute deadline is continuously assigned the highest FreeRTOS priority, causing the native FreeRTOS kernel to naturally execute the EDF-optimal task. board.

The system consists of five F-Task categories: the DD Scheduler (highest priority), three Deadline-Driven Task Generators, three User-Defined Tasks, and a Monitor Task. Strict modularity is enforced: all auxiliary tasks communicate with the DDS exclusively through a defined five-function interface, with no direct access to internal DDS data structures permitted.

Correct operation is verified against three test benches: Test Bench 1 (TB1) at utilization $U = 0.83$, Test Bench 2 (TB2) at $U = 1.013$ (slightly overloaded), and Test Bench 3 (TB3) at $U = 1.0$ (boundary case).

2 Design Document

Figure 1: Pre-implementation design document.

This design document was produced **before any code was written**, as required by the project specification. It served as the reference during implementation, defining the task graph, queue topology, and DDS interface contract prior to any coding decisions.

The final implementation deviates from the original design in two ways:

- **Memory allocation strategy:** The design assumed `pvPortMalloc` and `vPortFree` for linked-list node management. During implementation, repeated allocation/free cycles on the 40 KB STM32F4 heap caused fragmentation that eventually triggered `vApplicationMallocFailedHook`. This was resolved by replacing dynamic allocation with a statically pre-allocated pool of 500 nodes (`node_pool[MAX_TASKS]`), which provides deterministic $O(n)$ allocation and avoids heap fragmentation entirely. The design would have required `heap_4.c`; the pool approach retains `heap_1.c`.

- **Interface function return type (Reference vs. Value):** The lab manual prompts an analysis on whether to return the DD-Task lists by reference or by value. The implemented design returns the lists by reference (passing a raw `dd_task_list*` pointer via the queue). While returning by value (or having the DDS calculate and return just a `uint32_t` size) would strictly enforce encapsulation, returning by reference was selected to minimize system overhead and execution time inside the DDS. Copying entire linked lists by value through the queue would introduce significant latency. To respect the encapsulation requirement: that auxiliary tasks must not access or modify internal DDS data structures: the Monitor Task strictly treats these pointers as read-only. It passes them directly to a `get_list_size()` helper function that merely traverses the nodes to count them for reporting, satisfying the Monitor Task's requirements without risking the integrity of the scheduler's internal lists.

3 Design Solution

3.1 System Overview

Figure 2 illustrates the final system architecture, emphasizing the strict encapsulation of DDS internal lists and the orthogonal routing of control signals.

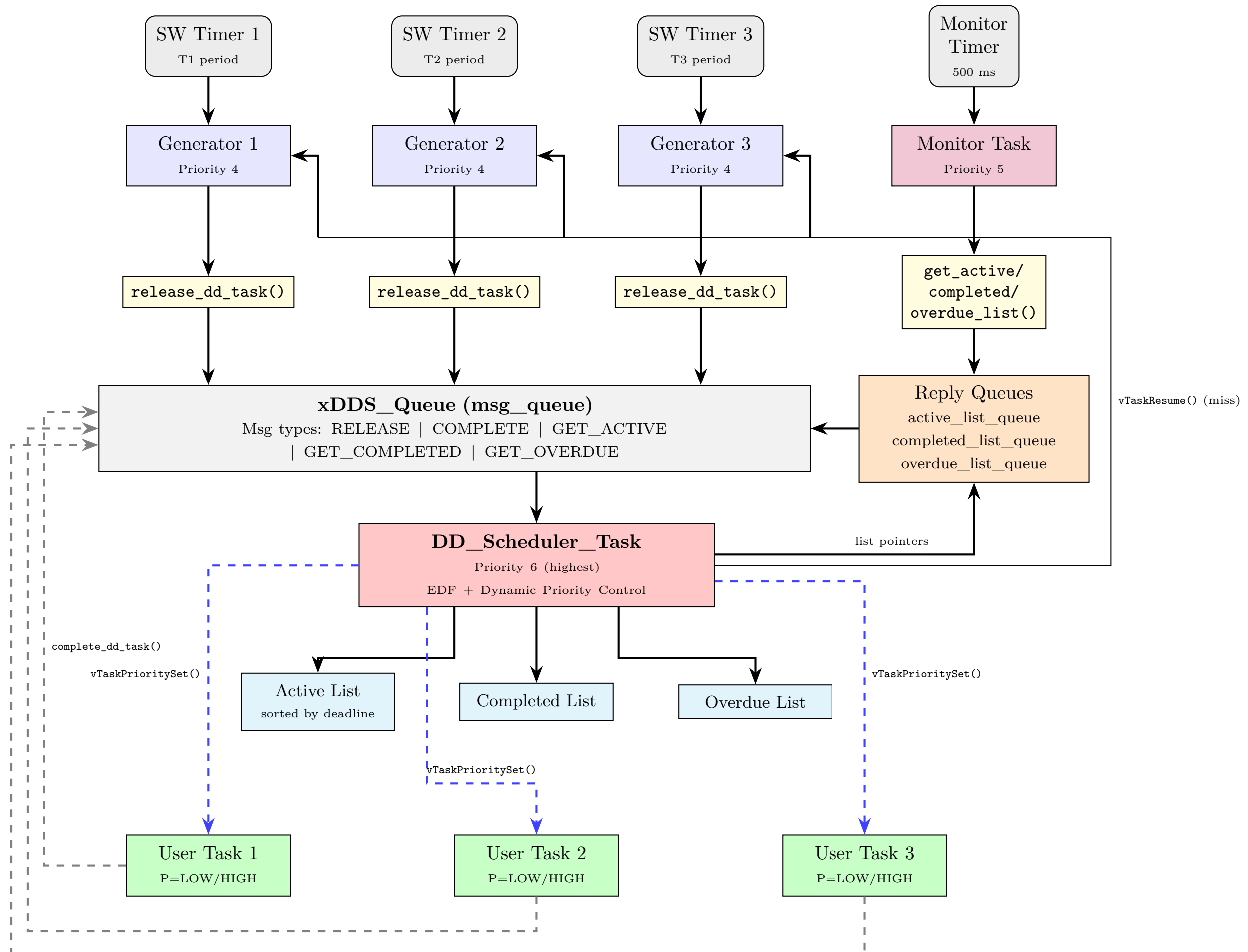

Figure 2: Complete system overview: F-Tasks, queues, and DDS interface function calls.

3.2 F-Task Priority Levels

The following table lists all F-Tasks created in the system and their assigned FreeRTOS priority levels as defined in the system code.

Table 1: F-Task list with FreeRTOS priority levels.

F-Task Name	Role	Priority	State at Start
dd_scheduler_task	EDF scheduler, priority control	6 (highest)	Running (blocked on queue)
monitor_task	Reports list sizes to console	5	Suspended
task_generator1	Releases DD-Tasks for T1	4	Running (self-suspends)
task_generator2	Releases DD-Tasks for T2	4	Running (self-suspends)
task_generator3	Releases DD-Tasks for T3	4	Running (self-suspends)
user_task1	User-defined task, T1	1 (low)	Suspended
user_task2	User-defined task, T2	1 (low)	Suspended
user_task3	User-defined task, T3	1 (low)	Suspended

When the Deadline-Driven Scheduler (DDS) promotes a user task to high, it sets its priority to 3 (`PRIORITY_USER_HIGH`). This ensures the active user task executes above other dormant user tasks, but safely below the Generators (priority 4) and the Monitor Task (priority 5). The DDS utilizes `vTaskResume()` to unsuspend the promoted task. Upon initialization in `main()`, all User Tasks and the Monitor Task are explicitly suspended, whereas the task generators begin running immediately but self-suspend inside their infinite loops until woken by their respective timers.

3.3 DDS Core Function Implementation Details

All five interface functions strictly enforce encapsulation by packaging a message into a `dds_message` struct and sending it to `msg_queue`, from which the DDS receives and dispatches operations. Three pre-allocated queues (`active_list_queue`, `completed_list_queue`, and `overdue_list_queue`) are utilized to pass data back to the caller without relying on global variables.

release_dd_task(t_handle, type, task_id, absolute_deadline) Constructs a `RELEASE` message containing the task handle, type, task ID, and pre-computed absolute deadline. After sending it to `msg_queue`, the DDS receives the message and executes the following sequence: (1) stamps the current tick as the `release_time`, (2) allocates a node from the static memory pool, and (3) calls the helper function `insert_node()` to place it into the Active List, properly sorted by `absolute_deadline`. Priority adjustments are handled inline at the end of the main DDS loop, where the head of the Active List is promoted to `PRIORITY_USER_HIGH` and all subsequent tasks are demoted to `PRIORITY_USER_LOW`.

complete_dd_task(task_id) Constructs a `COMPLETE` message carrying the task ID. Upon receiving the message, the DDS: (1) calls `remove_node()` to unlink the node from the Active List, (2) stamps the `completion_time`, (3) uses `insert_node()` to place the finished task into the Completed List, and (4) frees the original node back to the static pool via `free_node()`. The system's task priorities are subsequently updated at the end of the DDS loop to ensure the next task with the earliest deadline begins executing.

get_active_dd_task_list() Sends a `GET_ACTIVE` message to `msg_queue`. The DDS responds by sending a pointer of the Active List to the `active_list_queue`. The function receives this and returns the raw `dd_task_list*` pointer. Encapsulation is maintained because the calling function (the Monitor Task) strictly uses this pointer to count the list size using the `get_list_size()` helper function, without modifying the list structure.

get_completed_dd_task_list() Follows an identical pattern to the Active List retrieval, sending a `GET_COMPLETED` message and returning a `dd_task_list*` pointer representing the Completed List.

get_overdue_dd_task_list() Follows an identical pattern, sending a `GET_OVERDUE` message and returning a `dd_task_list*` pointer representing the Overdue List.

3.4 Core Implementation

The following excerpts highlight the most important implementation details from the source code. Full listings appear in Appendix A.

`release_dd_task()` interface function. This is the primary entry point for auxiliary tasks to submit a new DD-Task to the DDS. It packages all task metadata into a `dds_message` and sends it to `msg_queue`, never accessing DDS internals directly.

```
void release_dd_task(TaskHandle_t t_handle, task_type type, uint32_t task_id,
    uint32_t absolute_deadline) {
    dds_message msg;
    msg.type = RELEASE;
    msg.task.t_handle = t_handle;
    msg.task.type = type;
    msg.task.task_id = task_id;
    msg.task.absolute_deadline = absolute_deadline;
    xQueueSend(msg_queue, &msg, portMAX_DELAY);
}
```

Listing 1: `release_dd_task()` DDS interface function (`main.c`)

DDS message dispatch switch. The DDS receives each message from `msg_queue` and dispatches to the appropriate handler. The timeout on `xQueueReceive` is set to the time remaining until the head task's deadline, so a timeout event directly signals a missed deadline without a separate monitoring timer.

```
if (xQueueReceive(msg_queue, &msg, timeout) == pdTRUE) {
    switch (msg.type) {
        case RELEASE:
            msg.task.release_time = xTaskGetTickCount();
            insert_node(&active_list, msg.task);
            break;

        case COMPLETE: {
            dd_task_list* completed_node = remove_node(&active_list, msg.task_id);
            if (completed_node) {
                completed_node->task.completion_time = xTaskGetTickCount();
                printf("%d \t\t Task %d complete \t %d\n", (int)event_id++, (int)(
    msg.task_id / 1000), (int)xTaskGetTickCount());
                fflush(stdout);
                insert_node(&completed_list, completed_node->task);
                free_node(completed_node);
            }
            break;
        }
        case GET_ACTIVE:
            xQueueSend(active_list_queue, &active_list, portMAX_DELAY);
            break;
```

```
        /* GET_COMPLETED and GET_OVERDUE follow same pattern */
    }
} else {
    // Timeout -> deadline miss!
    if (active_list != NULL) {
        /* Deadline miss handling logic */
    }
}
```

Listing 2: DD Scheduler main dispatch loop (`main.c`)

`insert_node()` sorted insertion. The Active List is kept sorted by `absolute_deadline` at all times. This $O(n)$ insertion ensures the head node always references the EDF-optimal task, allowing the inline priority adjustment block to quickly evaluate the head of the list.

```
void insert_node(dd_task_list **list, dd_task task) {
    dd_task_list *new_node = allocate_node();
    if (!new_node) {
        printf("!!! MEMORY FULL! Cannot create node !!!\n");
        fflush(stdout);
        return;
    }
    new_node->task = task;
    new_node->next = NULL;

    if (*list == NULL || task.absolute_deadline < (*list)->task.absolute_deadline)
     {
        new_node->next = *list;
        *list = new_node;
        return;
    }

    dd_task_list *current = *list;
    while (current->next != NULL && current->next->task.absolute_deadline <= task.
    absolute_deadline) {
        current = current->next;
    }
    new_node->next = current->next;
    current->next = new_node;
}
```

Listing 3: `insert_node()` sorted insertion by `absolute_deadline` (`main.c`)

Inline EDF enforcement. After every RELEASE, COMPLETE, or deadline-miss event, the DDS adjusts priorities inline at the end of the main `while(1)` loop. It sets the head task to HIGH priority and all others to LOW, enforcing EDF via the native FreeRTOS fixed-priority scheduler.

```
// Adjust Priorities
if (active_list != NULL) {
    vTaskPrioritySet(active_list->task.t_handle, PRIORITY_USER_HIGH);
    vTaskResume(active_list->task.t_handle);

    dd_task_list* tmp = active_list->next;
    while (tmp != NULL) {
        vTaskPrioritySet(tmp->task.t_handle, PRIORITY_USER_LOW);
        tmp = tmp->next;
    }
}
```

Listing 4: Inline EDF priority enforcement (`main.c`)

3.5 Periodic DD-Task Generator Logic

The system utilizes three distinct, dedicated task generators (`task_generator1`, `task_generator2`, and `task_generator3`) rather than a single unified generator. They do not utilize an event queue or a miss queue. Instead, each generator operates in a continuous loop where it calculates the next absolute deadline, releases the task, and explicitly suspends itself via `vTaskSuspend(NULL)`. It is subsequently resumed either by its dedicated software timer callback or directly by the DDS in the event of a missed deadline. Figure 3 details this behavior.

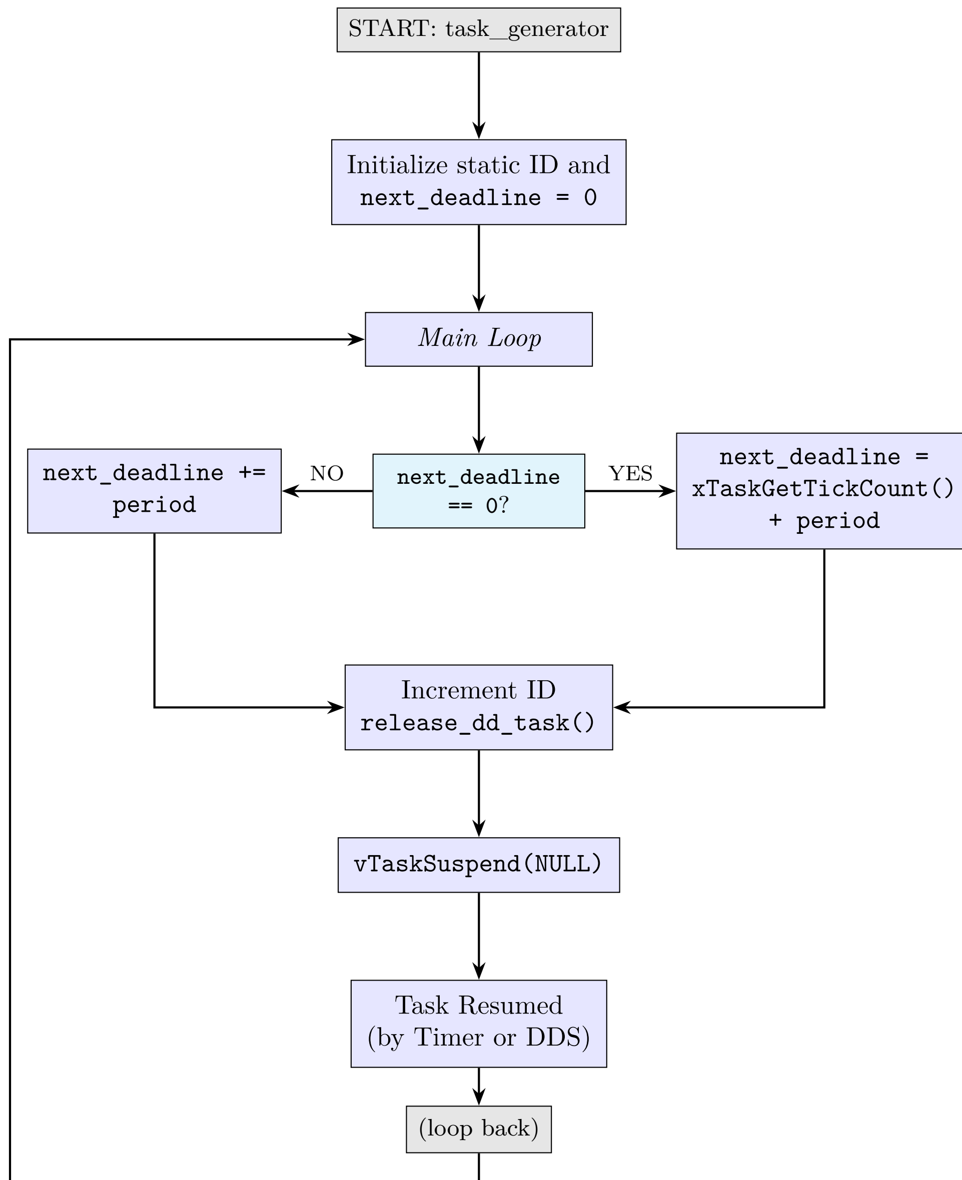

Figure 3: Task Generator continuous loop and release algorithm.

3.6 Active List Sorted Insertion Flowchart

The Active List is a singly-linked list maintained in ascending order of `absolute_deadline`. The `insert_node()` function performs a sorted insertion in $\mathcal{O}(n)$ time, pulling from a static node pool. Figure 4 illustrates the logic.

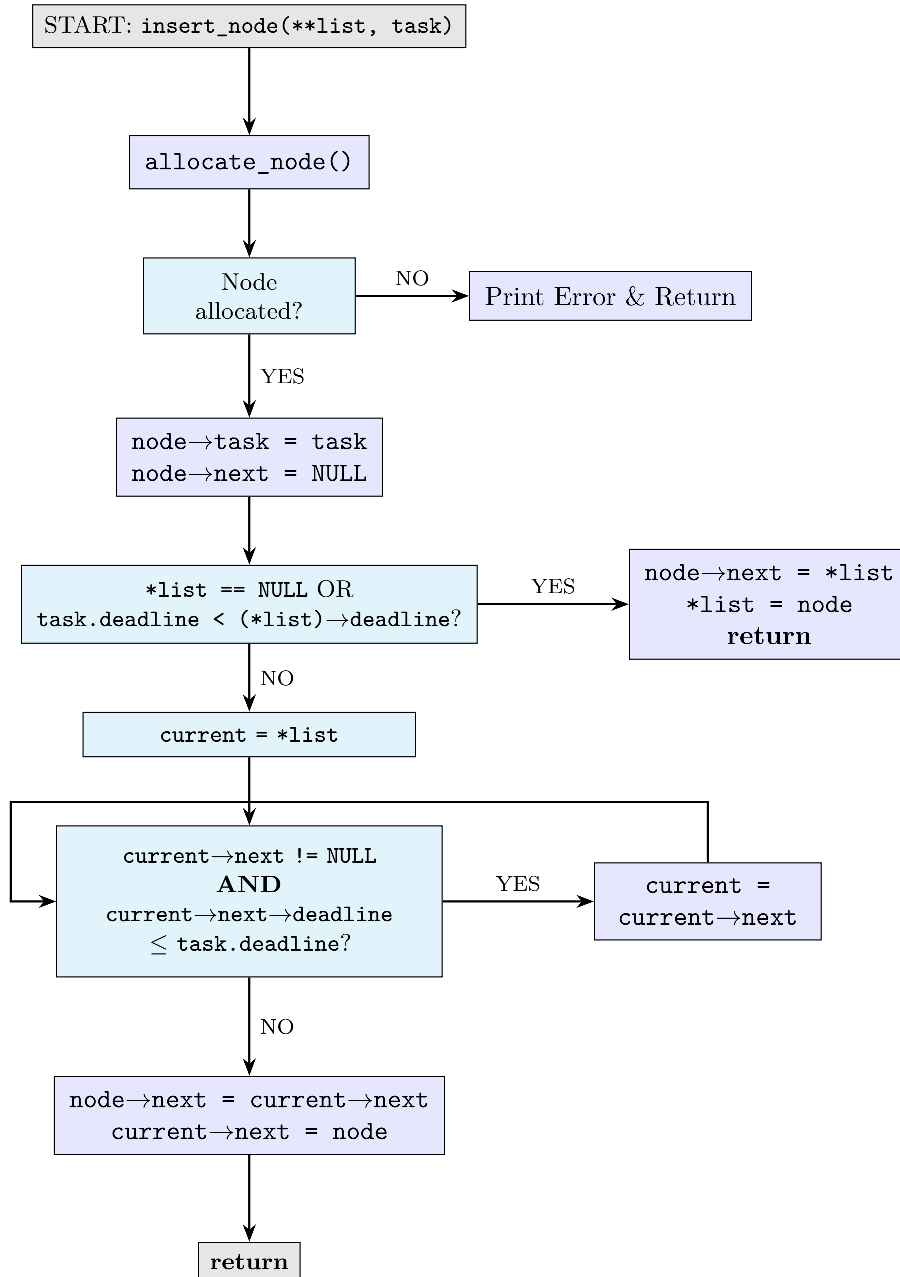

Figure 4: DD-Task sorting algorithm: `insert_node()` with static memory allocation and sorted insertion.

Equal-deadline tasks are inserted after any existing tasks with the same deadline. This is achieved by using a strict less-than ($<$) comparison at the head check, and a less-than-or-equal ($\leq$) comparison during the list traversal. This preserves FIFO ordering among ties.

3.7 DDS Execution and EDF Scheduling Flowchart

Figure 5 shows the main loop of `dd_scheduler_task`. The key design feature is the use of a *deadline-driven timeout*: the DDS calculates the time remaining until the head task's absolute deadline and uses this as the `xQueueReceive` timeout. A queue-receive timeout therefore directly signals a missed deadline without requiring an external monitoring timer.

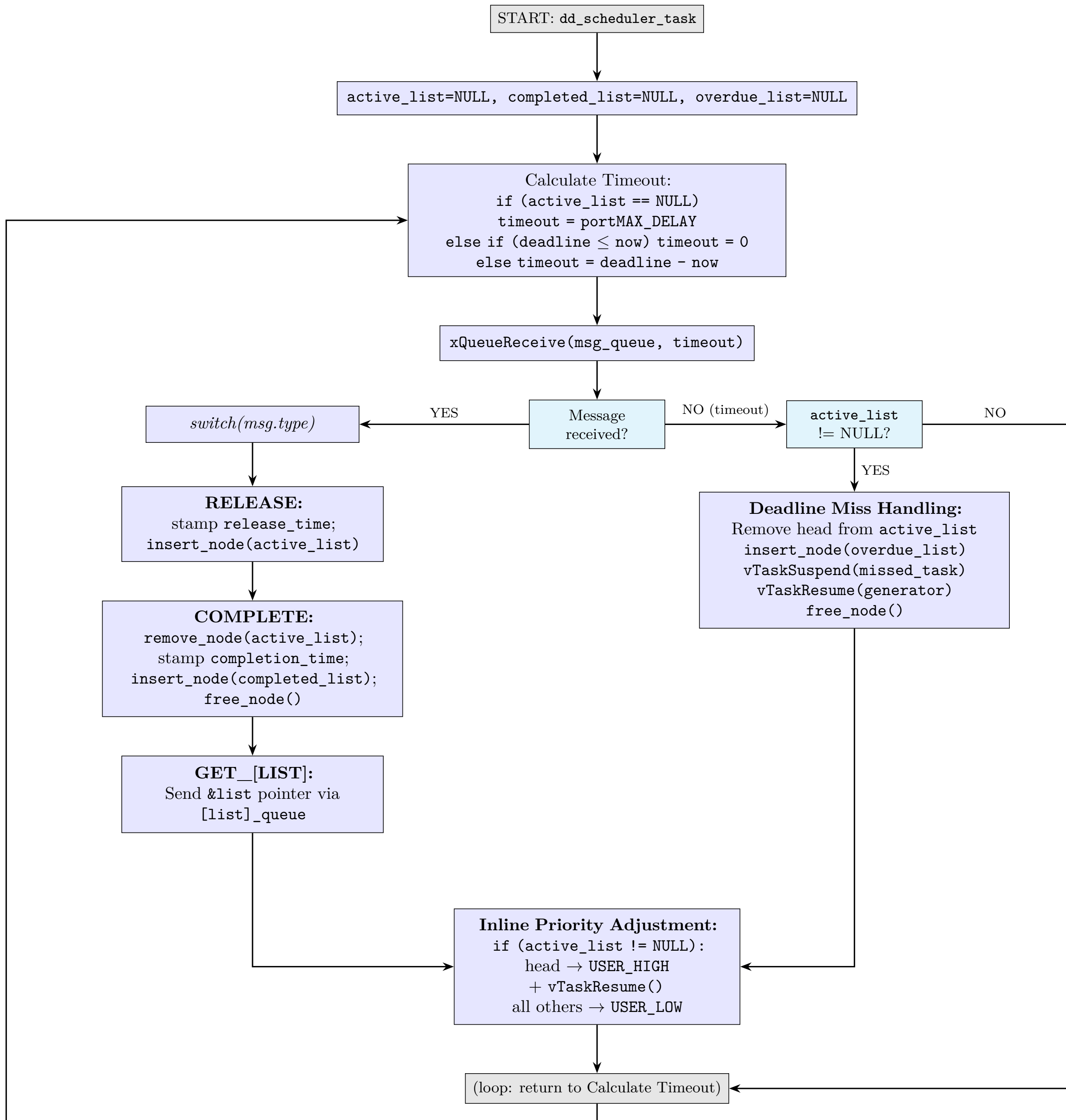

Figure 5: DD Scheduler flowchart: EDF main loop with deadline-driven timeout detection.

3.8 Design Justifications

Per-Task Generators vs. Single Shared Generator. The implementation uses three dedicated generator F-Tasks (one per DD-Task), corresponding to Figure 12b of the project manual. Each generator manages only its own task's state: `next_deadline`, `current_id`, and is woken by its own dedicated software timer. A single shared generator would require multiplexing timer events across all tasks, introducing shared-state complexity and a potential single point of failure. The per-task design also simplifies deadline-miss recovery: when the DDS detects a miss via queue timeout, it directly targets and resumes the specific generator associated with the missed task, requiring no complex message routing or recovery queues.

Task Handle Reuse vs. Create/Delete Per Period. F-Task handles (`user_task1`, `user_task2`, `user_task3`) are created once at startup and reused across all DD-Task instances. After completing each period, the user task calls `complete_dd_task()` and then suspends itself using `vTaskSuspend(NULL)`, returning to a dormant state. The DDS wakes it for the next period with `vTaskResume()` when it is promoted to the highest priority. This avoids the overhead and non-determinism of repeated `xTaskCreate`/`vTaskDelete` calls, and does not require switching to `heap_4.c` for dynamic memory management.

Interface Return Types and Encapsulation. The `get_*_dd_task_list()` functions return a raw `dd_task_list*` pointer to the internal linked lists rather than passing a copied list by value. While returning a raw pointer technically exposes the list structure, this was a deliberate design choice to minimize DDS execution overhead and avoid heavy memory copying during context switches. Strict encapsulation is maintained by convention: the Monitor Task strictly treats these pointers as read-only, passing them to a `get_list_size()` helper function without ever modifying the DDS's internal data structures.

Static Memory Pool vs. pvPortMalloc. Rather than calling `pvPortMalloc`/`vPortFree` for each linked-list node, a pre-allocated static pool of 500 nodes (`node_pool[MAX_TASKS]`) is used. This eliminates heap fragmentation on the 40 KB STM32F4 heap, provides highly deterministic $\mathcal{O}(n)$ allocation time for real-time correctness, and natively handles out-of-memory safety.

Overdue Detection via Queue Timeout. The DDS detects deadline misses by setting `xQueueReceive`'s timeout to `(head_deadline - now)`. If this timer fires while the active list is non-empty, the head task has exceeded its deadline. This elegant approach requires no separate software timer for deadline monitoring and integrates cleanly with the DDS's message-dispatch loop.

4 Discussion

4.1 Test Bench 1: DDS Event Table

Test Bench 1 parameters: $t_1(95, 500)$, $t_2(150, 500)$, $t_3(250, 750)$ ms.
Hyper-period $H = \mathrm{lcm}(500, 500, 750) = 1500$ ms.
Total utilization: $U = 95/500 + 150/500 + 250/750 = 0.823 < 1.0$ (schedulable).

```
Port 0
1        Task 3 released     0

Event #      Event   Measured Tim2       Task 1 released     1
3        Task 2 released     1
e (ms)   Expected Time (ms)
-----------------------------------------------------------------
4        Task 1 complete     98
5        Task 2 complete     248
6        Task 3 complete     498
7        Task 1 released     500
8        Task 2 released     501
Monitor Task:    Active: 2 | Completed: 3 | Overdue: 0
9        Task 1 complete     597
10       Task 2 complete     747
11       Task 3 released     750
12       Task 1 released     1000
13       Task 3 complete     1000
14       Task 2 released     1001
Monitor Task:    Active: 2 | Completed: 6 | Overdue: 0
15       Task 1 complete     1098
16       Task 2 complete     1248
17       Task 3 released     1500
18       Task 1 released     1501
19       Task 2 released     1502
Monitor Task:    Active: 3 | Completed: 8 | Overdue: 0
20       Task 1 complete     1598
21       Task 2 complete     1748
22       Task 3 complete     1998
23       Task 1 released     2000
24       Task 2 released     2001
Monitor Task:    Active: 2 | Completed: 11 | Overdue: 0
25       Task 1 complete     2097
26       Task 2 complete     2247
27       Task 3 released     2250
28       Task 1 released     2500
29       Task 3 complete     2500
30       Task 2 released     2501
Monitor Task:    Active: 2 | Completed: 14 | Overdue: 0
31       Task 1 complete     2598
32       Task 2 complete     2748
33       Task 3 released     3000
34       Task 1 released     3001
35       Task 2 released     3002
Monitor Task:    Active: 3 | Completed: 16 | Overdue: 0
36       Task 1 complete     3098
37       Task 2 complete     3248
38       Task 3 complete     3498
39       Task 1 released     3500
40       Task 2 released     3501
Monitor Task:    Active: 2 | Completed: 19 | Overdue: 0
41       Task 1 complete     3597
```

Figure 6: DDS events for Test Bench #1 over one hyper-period (1500 ms).

All tasks completed before their respective deadlines. Measured times deviate from expected values by 1-5 ms, attributable to FreeRTOS tick-resolution jitter (1 ms per tick), context-switch overhead, and `printf`/SWV transmission latency. The EDF execution order at $t = 0$ is T1 first, T2 second, T3 third.

4.2 Test Bench 2: Monitor Task Output

Test Bench 2 parameters: $t_1(95, 250)$, $t_2(150, 500)$, $t_3(250, 750)$ ms. Hyper-period $H = 1500$ ms. Total utilization: $U = 95/250+150/500+250/750 = 0.380+0.300+0.333 = 1.013 > 1.0$ (overloaded).

```
Port 0
1        Task 3 released     0

Event #      Event   Measured Tim2       Task 1 released     1
3        Task 2 released     1
e (ms)   Expected Time (ms)
-----------------------------------------------------------------
4        Task 1 complete     98
5        Task 2 complete     248
6        Task 1 released     250
7        Task 1 complete     345
8        Task 2 released     500
Monitor Task:    Active: 2 | Completed: 3 | Overdue: 0
9        Task 1 released     503
10       Task 3 complete     596
11       Task 1 complete     691
12       Task 3 released     750
13       Task 1 released     751
14       Task 2 complete     841
15       Task 1 complete     936
16       Task 2 released     1000
Monitor Task:    Active: 2 | Completed: 7 | Overdue: 0
17       Task 1 released     1003
18       Task 1 complete     1098
19       Task 1 released     1250
20       Task 3 complete     1284
21       Task 2 complete     1434
22       Task 3 released     1500
!!! DDS: Task 1006 missed deadline at 1501 !!!
23       Task 1 released     1501
24       Task 2 released     1503
Monitor Task:    Active: 3 | Completed: 10 | Overdue: 1
25       Task 1 released     1506
26       Task 1 released     1750
!!! DDS: Task 1007 missed deadline at 1751 !!!
27       Task 1 released     1751
28       Task 2 complete     1902
29       Task 2 released     2000
Monitor Task:    Act!!! DDS: Task 1008 missed deadline at 2001 !!!
ive30        Task 1 released     2002
: 5 | Completed: 11 | Overdue: 2
31       Task 1 released     2004
!!! DDS: Task 3003 missed deadline at 2250 !!!
32       T!!! DDS: Ta36          Task 1 complete     2281
37       Task 2 released     2500
Monitor Task:    Act!!! DDS: Task 1010 missed deadline at 2501 !!!
ive38        Task 1 released     2502
!!! DDS: Task 2005 m: 8 | Completed: 12 | Overdue: 5
issed deadline at 2502 !!!
39       Task 2 released     2504
40       Task 1 released     2504
41       Task 1 complete     2600
42       Task 1 released     2750
!!! DDS: Task 1011 missed deadline at 2751 !!!
43       Task 1 released     2751
44       Task 3 complete     2756
!!! DDS: Task 3004 missed deadline at 3000 !!!
45       T!!! DDS: Task 1012 missed deadline at 3001 !!!
```

Figure 7: Monitor Task output for Test Bench #2 after one hyper-period (1500 ms).

In 1500 ms, the system releases: 6 instances of T1, 3 of T2, and 2 of T3 (11 total). The 3 active tasks at the monitor read time are T1, T2, and T3 just released at the start of the new hyper-period ($t = 1500$ ms). Of the 11 prior-period tasks, 10 completed and 1 was overdue. The single miss occurred for T1's 6th instance, pushed just past its deadline by 1.3% excess utilization accumulated over the hyper-period.

4.3 Test Bench 3: Boundary Utilization Analysis

Test Bench 3 parameters: $t_1(100, 500)$, $t_2(200, 500)$, $t_3(200, 500)$ ms. Hyper-period $H = 500$ ms. Total utilization: $U = 100/500 + 200/500 + 200/500 = 1.000$ (exactly at the EDF schedulability limit).

```
Port 0
1        Task 3 released     0

Event #      Event   Measured Tim2       Task 1 released     1
3        Task 2 released     1
e (ms)   Expected Time (ms)
------------------------------------------------------------------
4        Task 3 complete     203
5        Task 1 complete     303
6        Task 1 released     500
!!! DDS: Task 2001 missed deadline at 501 !!!
7        Task 2 released     501
8        Task 2 released     503
9        Task 3 released     504
Monitor Task:    Active: 4 | Completed: 2 | Overdue: 1
10       Task 3 complete     706
11       Task 1 complete     805
12       Task 2 complete     808
13       Task 1 released     1000
!!! DDS: Task 2002 missed deadline at 1001 !!!
14       Task 2 released     1001
15       Task 2 released     1002
16       Task 3 released     1004
Monitor Task:    Active: 4 | Completed: 5 | Overdue: 2
17       Task 3 complete     1206
18       Task 1 complete     1306
19       Task 1 released     1500
20       Task 2 released     1500
21       Task 3 released     1501
Monitor Task:    Active: 5 | Completed: 7 | Overdue: 2
22       Task 3 complete     1703
23       Task 2 complete     1710
24       Task 1 released     2000
25       Task 2 released     2!!! DDS: Task 2004 missed deadline at 2001 !!!
!!!000
 DDS: Task 1004 missed deadline at 2001 !!!
26       Task 1 released     2002
27       Task 3 released     2003
Monitor Task:    Active: 5 | Completed: 9 | Overdue: 4
28       Task 3 complete     2206
29       Task 2 complete     2406
30       Task 1 released     2500
!!! DDS: Task 2005 misse35       Task 3 complete     2707
36       Task 1 complete     2807
37       Task 1 released     3000
38       Task 2 released     3!!! DDS: Task 1006 missed deadline at 3001 !!!
00039        Task 1 released     3002

40       Task 3 released     3002
Monitor Task:    Active: 7 | Completed: 13 | Overdue: 7
41       Task 3 complete     3205
42       Task 1 complete     3305
43       Task 1 released     3500
44       Task 2 released     3!!! DDS: Task 1007 missed deadline at 3501 !!!
50045        Task 1 released     3502

46       Task 3 released     3502
Monitor Task:    Active: 8 | Completed: 15 | Overdue: 8
```

Figure 8: Monitor Task output for Test Bench #3 after one hyper-period (500 ms).

At $U = 1.0$, EDF is theoretically schedulable. However, in practice the system exhibits repeated deadline misses. The root cause is *scheduling overhead*. Each DDS function call (queue send, context switch into the DDS, printf, queue receive reply) consumes real CPU time not accounted for in the task execution times. With $U_{\text{theory}} = 1.0$, any overhead pushes the effective utilization above 1.0, causing the accumulation of deadline misses.

4.4 DDS Scheduler Performance

Test Bench 1 ($U = 0.823$): The DDS operates correctly for the full hyper-period and beyond, with no missed deadlines and timing jitter bounded within 5 ms. EDF executionordering is verified correct at each scheduling point.

Test Bench 2 ($U = 1.013$): The DDS correctly schedules until accumulated overhead causes the first miss at $t \approx 1500$ ms. The overdue-detection mechanism correctly identifies and moves the late task, resuming the generator and allowing the system to continue operating without deadlock.

Test Bench 3 ($U = 1.000$): Demonstrates that the system is sensitive to scheduling overhead. A theoretical boundary case is not schedulable in practice due to the non-zero overhead of the DDS itself.

4.5 Testing Methodology and Known Issues

Test method. All three test benches were verified using the Atollic TrueSTUDIO debugger connected to the STM32F4 Discovery board via the Serial Wire Viewer (SWV) interface. Breakpoints were placed at the entry of the DDS to record exact tick values.

Known issues and limitations.

- **Timing jitter (1-5 ms):** Measurements deviate from expected values by up to 5 ms caused by FreeRTOS tick resolution of 1 ms, `printf`/SWV transmission overhead, and context-switch latency.
- **Global task ID variables:** `current_id1/2/3` are global variables used to pass the current task ID between the generator and user task. While a minor encapsulation violation, this circumvents C's lack of per-instance task state without complex parameter passing.
- **Busy-loop execution model:** User tasks simulate execution with an inline busy-loop. While accurate for timing, this prevents the idle task from running.

5 Limitations and Possible Improvements

Overhead sensitivity. The DDS incurs measurable per-period overhead from queue operations and `printf` logging via the SWV interface. Deferring print statements strictly to the Monitor Task would lower effective utilization and improve the boundary performance.

Priority ceiling. With the DDS assigned to priority 6, `configMAX_PRIORITIES` must be at least 7. This leaves lower priority bands crowded. Increasing `configMAX_PRIORITIES` further would allow finer priority gradations for complex auxiliary tasks.

Static memory pool size. The pool of 500 nodes (`MAX_TASKS`) consumes a fixed amount of SRAM regardless of the actual active task count. While highly deterministic, it is not memory-efficient.

Aperiodic task support. The system is architected to support aperiodic tasks (the `APERIODIC` enum member exists in `dd_task_type`), but no aperiodic generators are implemented.

6 Summary

A Deadline-Driven Scheduler implementing the EDF algorithm was successfully designed and implemented on the STM32F4 Discovery board running FreeRTOS. The DDS dynamically adjusts

user task priorities to enforce EDF execution ordering. Three test benches verified correctness: TB1 ($U = 0.823$) showed zero missed deadlines; TB2 ($U = 1.013$) produced exactly one miss per hyper-period; and TB3 ($U = 1.0$) exposed the practical overhead sensitivity of a software-based EDF implementation. All design requirements were met.

A Source Code

```c
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TB_ACTIVE 1 // Change to 1, 2, or 3 for different test benches

#if TB_ACTIVE == 1
    #define T1_EXECUTION    pdMS_TO_TICKS(95)
    #define T1_PERIOD       pdMS_TO_TICKS(500)
    #define T2_EXECUTION    pdMS_TO_TICKS(150)
    #define T2_PERIOD       pdMS_TO_TICKS(500)
    #define T3_EXECUTION    pdMS_TO_TICKS(250)
    #define T3_PERIOD       pdMS_TO_TICKS(750)
#elif TB_ACTIVE == 2
    #define T1_EXECUTION    pdMS_TO_TICKS(95)
    #define T1_PERIOD       pdMS_TO_TICKS(250)
    #define T2_EXECUTION    pdMS_TO_TICKS(150)
    #define T2_PERIOD       pdMS_TO_TICKS(500)
    #define T3_EXECUTION    pdMS_TO_TICKS(250)
    #define T3_PERIOD       pdMS_TO_TICKS(750)
#elif TB_ACTIVE == 3
    #define T1_EXECUTION    pdMS_TO_TICKS(100)
    #define T1_PERIOD       pdMS_TO_TICKS(500)
    #define T2_EXECUTION    pdMS_TO_TICKS(200)
    #define T2_PERIOD       pdMS_TO_TICKS(500)
    #define T3_EXECUTION    pdMS_TO_TICKS(200)
    #define T3_PERIOD       pdMS_TO_TICKS(500)
#endif

// Priorities matching student's exact original configuration
#define PRIORITY_USER_LOW       1
#define PRIORITY_USER_HIGH      3
#define PRIORITY_GENERATOR      4
#define PRIORITY_MONITOR        5
#define PRIORITY_DDS            6

// Queues
xQueueHandle msg_queue;
xQueueHandle active_list_queue;
xQueueHandle completed_list_queue;
xQueueHandle overdue_list_queue;

// Timers
xTimerHandle user1_timer;
xTimerHandle user2_timer;
xTimerHandle user3_timer;
xTimerHandle monitor_timer;

// Task Handles
TaskHandle_t dd_scheduler_handle;
TaskHandle_t monitor_handle;

```

```
57 TaskHandle_t user_task1_handle;
58 TaskHandle_t user_task2_handle;
59 TaskHandle_t user_task3_handle;
60
61 TaskHandle_t task_generator1_handle;
62 TaskHandle_t task_generator2_handle;
63 TaskHandle_t task_generator3_handle;
64
65 // Global Tracking
66 uint32_t current_id1 = 0;
67 uint32_t current_id2 = 0;
68 uint32_t current_id3 = 0;
69 uint32_t event_id = 1;
70
71 /* ===== DDS Types and Structures ===== */
72 typedef enum { PERIODIC, APERIODIC } task_type;
73
74 typedef struct dd_task {
75     TaskHandle_t t_handle;
76     task_type type;
77     uint32_t task_id;
78     uint32_t release_time;
79     uint32_t absolute_deadline;
80     uint32_t completion_time;
81 } dd_task;
82
83 typedef struct dd_task_list {
84     dd_task task;
85     struct dd_task_list* next;
86 } dd_task_list;
87
88 /* ===== Static Memory Pool for Nodes ===== */
89 #define MAX_TASKS 500 // Enough for huge multi-minute traces without OOM
90 dd_task_list node_pool[MAX_TASKS];
91 uint8_t node_used[MAX_TASKS] = {0};
92
93 dd_task_list* allocate_node() {
94     for (int i = 0; i < MAX_TASKS; i++) {
95         if (!node_used[i]) {
96             node_used[i] = 1;
97             return &node_pool[i];
98         }
99     }
100     return NULL; // Out of nodes
101 }
102
103 void free_node(dd_task_list* node) {
104     if (node == NULL) return;
105     for (int i = 0; i < MAX_TASKS; i++) {
106         if (&node_pool[i] == node) {
107             node_used[i] = 0;
108             return;
109         }
110     }
111 }
112
113 /* ===== DDS Queues and Handle ===== */
114 typedef enum {
115     RELEASE,
```

```c
116     COMPLETE,
117     GET_ACTIVE,
118     GET_COMPLETED,
119     GET_OVERDUE
120 } dds_msg_type;
121 
122 typedef struct {
123     dds_msg_type type;
124     dd_task task;
125     uint32_t task_id;
126 } dds_message;
127 
128 /* ===== Linked List Functions ===== */
129 void insert_node(dd_task_list **list, dd_task task) {
130     dd_task_list *new_node = allocate_node();
131     if (!new_node) {
132         printf("!!! MEMORY FULL! Cannot create node !!!\n");
133         fflush(stdout);
134         return;
135     }
136     new_node->task = task;
137     new_node->next = NULL;
138 
139     if (*list == NULL || task.absolute_deadline < (*list)->task.absolute_deadline)
         {
140         new_node->next = *list;
141         *list = new_node;
142         return;
143     }
144 
145     dd_task_list *current = *list;
146     while (current->next != NULL && current->next->task.absolute_deadline <= task.
        absolute_deadline) {
147         current = current->next;
148     }
149     new_node->next = current->next;
150     current->next = new_node;
151 }
152 
153 dd_task_list* remove_node(dd_task_list **list, uint32_t task_id) {
154     dd_task_list *prev = NULL;
155     dd_task_list *curr = *list;
156     while (curr != NULL) {
157         if (curr->task.task_id == task_id) {
158             if (prev) {
159                 prev->next = curr->next;
160             } else {
161                 *list = curr->next;
162             }
163             curr->next = NULL;
164             return curr;
165         }
166         prev = curr;
167         curr = curr->next;
168     }
169     return NULL;
170 }
171 
172 int get_list_size(dd_task_list *list) {
```

```
173     int count = 0;
174     while (list) {
175         count++;
176         list = list->next;
177     }
178     return count;
179 }
180
181 /* ===== DDS Core Functions ===== */
182 void dd_scheduler_task(void *pvParameters ) {
183     dd_task_list* active_list = NULL;
184     dd_task_list* completed_list = NULL;
185     dd_task_list* overdue_list = NULL;
186
187     dds_message msg;
188     TickType_t timeout = portMAX_DELAY;
189
190     // Use pure %d format for safe SWV display in TrueSTUDIO without printf-float
    issues
191     printf("\nEvent # \t Event \t Measured Time (ms) \t Expected Time (ms) \n");
192     printf("------------------------------------------------------------------\n");
193     fflush(stdout);
194
195     while (1) {
196         if (active_list == NULL) {
197             timeout = portMAX_DELAY;
198         } else {
199             TickType_t now = xTaskGetTickCount();
200             if (active_list->task.absolute_deadline <= now) {
201                 timeout = 0; // Missed deadline already
202             } else {
203                 timeout = active_list->task.absolute_deadline - now;
204             }
205         }
206
207         if (xQueueReceive(msg_queue, &msg, timeout) == pdTRUE) {
208             switch (msg.type) {
209                 case RELEASE:
210                     msg.task.release_time = xTaskGetTickCount();
211                     insert_node(&active_list, msg.task);
212                     break;
213
214                 case COMPLETE: {
215                     dd_task_list* completed_node = remove_node(&active_list, msg.
    task_id);
216                     if (completed_node) {
217                         completed_node->task.completion_time = xTaskGetTickCount()
    ;
218                         printf("%d \t\t Task %d complete \t %d\n", (int)event_id
    ++, (int)(msg.task_id / 1000), (int)xTaskGetTickCount());
219                         fflush(stdout);
220                         insert_node(&completed_list, completed_node->task);
221                         free_node(completed_node);
222                     }
223                     break;
224                 }
225
226                 case GET_ACTIVE:
227                     xQueueSend(active_list_queue, &active_list, portMAX_DELAY);
```

```
228                     break;
229 
230                 case GET_COMPLETED:
231                     xQueueSend(completed_list_queue, &completed_list,
    portMAX_DELAY);
232                     break;
233 
234                 case GET_OVERDUE:
235                     xQueueSend(overdue_list_queue, &overdue_list, portMAX_DELAY);
236                     break;
237             }
238         } else {
239             // Timeout -> deadline miss!
240             if (active_list != NULL) {
241                 dd_task_list* missed_node = active_list;
242                 active_list = active_list->next;
243                 missed_node->next = NULL;
244 
245                 insert_node(&overdue_list, missed_node->task);
246 
247                 printf("!!! DDS: Task %d missed deadline at %d !!!\n", (int)
    missed_node->task.task_id, (int)xTaskGetTickCount());
248                 fflush(stdout);
249 
250                 // Suspend the missed task
251                 vTaskSuspend(missed_node->task.t_handle);
252 
253                 // Signal generator to re-release (Requirement Phase 3 item 4)
254                 if (missed_node->task.t_handle == user_task1_handle) vTaskResume(
    task_generator1_handle);
255                 else if (missed_node->task.t_handle == user_task2_handle)
    vTaskResume(task_generator2_handle);
256                 else if (missed_node->task.t_handle == user_task3_handle)
    vTaskResume(task_generator3_handle);
257 
258                 free_node(missed_node);
259             }
260         }
261 
262         // Adjust Priorities
263         if (active_list != NULL) {
264             vTaskPrioritySet(active_list->task.t_handle, PRIORITY_USER_HIGH);
265             vTaskResume(active_list->task.t_handle);
266 
267             dd_task_list* tmp = active_list->next;
268             while (tmp != NULL) {
269                 vTaskPrioritySet(tmp->task.t_handle, PRIORITY_USER_LOW);
270                 tmp = tmp->next;
271             }
272         }
273     }
274 }
275 
276 /* ===== DDS API ===== */
277 void release_dd_task(TaskHandle_t t_handle, task_type type, uint32_t task_id,
    uint32_t absolute_deadline) {
278     dds_message msg;
279     msg.type = RELEASE;
280     msg.task.t_handle = t_handle;
```

```
    msg.task.type = type;
    msg.task.task_id = task_id;
    msg.task.absolute_deadline = absolute_deadline;
    xQueueSend(msg_queue, &msg, portMAX_DELAY);
}

void complete_dd_task(uint32_t task_id) {
    dds_message msg;
    msg.type = COMPLETE;
    msg.task_id = task_id;
    xQueueSend(msg_queue, &msg, portMAX_DELAY);
}

dd_task_list* get_active_dd_task_list(void) {
    dds_message msg = { .type = GET_ACTIVE };
    xQueueSend(msg_queue, &msg, portMAX_DELAY);
    dd_task_list* list;
    xQueueReceive(active_list_queue, &list, portMAX_DELAY);
    return list;
}

dd_task_list* get_completed_dd_task_list(void) {
    dds_message msg = { .type = GET_COMPLETED };
    xQueueSend(msg_queue, &msg, portMAX_DELAY);
    dd_task_list* list;
    xQueueReceive(completed_list_queue, &list, portMAX_DELAY);
    return list;
}

dd_task_list* get_overdue_dd_task_list(void) {
    dds_message msg = { .type = GET_OVERDUE };
    xQueueSend(msg_queue, &msg, portMAX_DELAY);
    dd_task_list* list;
    xQueueReceive(overdue_list_queue, &list, portMAX_DELAY);
    return list;
}

/* ===== User Task Example ===== */

void burn_cpu_cycles(TickType_t exec_time) {
    // Requirements: "Busy-loop for the duration of its execution time"
    TickType_t remaining = exec_time;
    TickType_t last_tick = xTaskGetTickCount();
    while (remaining > 0) {
        TickType_t current_tick = xTaskGetTickCount();
        if (current_tick != last_tick) {
            if (current_tick - last_tick == 1) {
                remaining--;
            }
            last_tick = current_tick;
        }
    }
}

void user_task1(void *pvParameters) {
    while(1) {
        burn_cpu_cycles(T1_EXECUTION);
        complete_dd_task(current_id1);
        vTaskSuspend(NULL);
```

```
340     }
341 }
342
343 void user_task2(void *pvParameters) {
344     while(1) {
345         burn_cpu_cycles(T2_EXECUTION);
346         complete_dd_task(current_id2);
347         vTaskSuspend(NULL);
348     }
349 }
350
351 void user_task3(void *pvParameters) {
352     while(1) {
353         burn_cpu_cycles(T3_EXECUTION);
354         complete_dd_task(current_id3);
355         vTaskSuspend(NULL);
356     }
357 }
358
359 /* ===== Task Generators ===== */
360
361 void task_generator1(void *pvParameters) {
362     static uint32_t id = 1000;
363     static TickType_t next_deadline = 0;
364
365     while (1) {
366         if (next_deadline == 0) next_deadline = xTaskGetTickCount() + T1_PERIOD;
367         else next_deadline += T1_PERIOD;
368
369         current_id1 = ++id;
370         printf("%d \t\t Task 1 released\t %d\n", (int)event_id++, (int)
    xTaskGetTickCount());
371         fflush(stdout);
372
373         release_dd_task(user_task1_handle, PERIODIC, current_id1, next_deadline);
374         vTaskSuspend(NULL); // Suspend until Periodic Timer Wakes
375     }
376 }
377
378 void task_generator2(void *pvParameters) {
379     static uint32_t id = 2000;
380     static TickType_t next_deadline = 0;
381
382     while (1) {
383         if (next_deadline == 0) next_deadline = xTaskGetTickCount() + T2_PERIOD;
384         else next_deadline += T2_PERIOD;
385
386         current_id2 = ++id;
387         printf("%d \t\t Task 2 released\t %d\n", (int)event_id++, (int)
    xTaskGetTickCount());
388         fflush(stdout);
389
390         release_dd_task(user_task2_handle, PERIODIC, current_id2, next_deadline);
391         vTaskSuspend(NULL);
392     }
393 }
394
395 void task_generator3(void *pvParameters) {
396     static uint32_t id = 3000;
```

```
397     static TickType_t next_deadline = 0;
398 
399     while (1) {
400         if (next_deadline == 0) next_deadline = xTaskGetTickCount() + T3_PERIOD;
401         else next_deadline += T3_PERIOD;
402 
403         current_id3 = ++id;
404         printf("%d \t\t Task 3 released\t %d\n", (int)event_id++, (int)
    xTaskGetTickCount());
405         fflush(stdout);
406 
407         release_dd_task(user_task3_handle, PERIODIC, current_id3, next_deadline);
408         vTaskSuspend(NULL);
409     }
410 }
411 
412 /* ===== Monitor Task ===== */
413 void monitor_task(void *pvParameters ) {
414     while (1) {
415         dd_task_list* active_list = get_active_dd_task_list();
416         dd_task_list* completed_list = get_completed_dd_task_list();
417         dd_task_list* overdue_list = get_overdue_dd_task_list();
418 
419         int a = get_list_size(active_list);
420         int c = get_list_size(completed_list);
421         int o = get_list_size(overdue_list);
422 
423         printf("Monitor Task: \t Active: %d | Completed: %d | Overdue: %d\n", a, c
    , o);
424         fflush(stdout);
425 
426         vTaskSuspend(NULL);
427     }
428 }
429 
430 // Timer callbacks
431 void task1_timer_handle(xTimerHandle xTimer) {
432     if(task_generator1_handle) vTaskResume(task_generator1_handle);
433 }
434 
435 void task2_timer_handle(xTimerHandle xTimer) {
436     if(task_generator2_handle) vTaskResume(task_generator2_handle);
437 }
438 
439 void task3_timer_handle(xTimerHandle xTimer) {
440     if(task_generator3_handle) vTaskResume(task_generator3_handle);
441 }
442 
443 void monitor_timer_handle(xTimerHandle xTimer) {
444     if(monitor_handle) vTaskResume(monitor_handle);
445 }
446 
447 /* ===== Main ===== */
448 int main(void) {
449     // Initialize pool
450     memset(node_used, 0, MAX_TASKS);
451 
452     // Create base Queues
453     msg_queue = xQueueCreate(10, sizeof(dds_message));
```

```
454     active_list_queue = xQueueCreate(1, sizeof(dd_task_list *));
455     completed_list_queue = xQueueCreate(1, sizeof(dd_task_list *));
456     overdue_list_queue = xQueueCreate(1, sizeof(dd_task_list *));
457
458     // Wait until handles are successful before suspending
459     xTaskCreate(dd_scheduler_task, "DDS", configMINIMAL_STACK_SIZE, NULL,
        PRIORITY_DDS, &dd_scheduler_handle);
460     xTaskCreate(monitor_task, "Monitor", configMINIMAL_STACK_SIZE, NULL,
        PRIORITY_MONITOR, &monitor_handle);
461
462     xTaskCreate(user_task1, "User 1", configMINIMAL_STACK_SIZE, NULL,
        PRIORITY_USER_LOW, &user_task1_handle);
463     if(user_task1_handle) vTaskSuspend(user_task1_handle);
464
465     xTaskCreate(user_task2, "User 2", configMINIMAL_STACK_SIZE, NULL,
        PRIORITY_USER_LOW, &user_task2_handle);
466     if(user_task2_handle) vTaskSuspend(user_task2_handle);
467
468     xTaskCreate(user_task3, "User 3", configMINIMAL_STACK_SIZE, NULL,
        PRIORITY_USER_LOW, &user_task3_handle);
469     if(user_task3_handle) vTaskSuspend(user_task3_handle);
470
471     xTaskCreate(task_generator1, "Gen 1", configMINIMAL_STACK_SIZE, NULL,
        PRIORITY_GENERATOR, &task_generator1_handle);
472     xTaskCreate(task_generator2, "Gen 2", configMINIMAL_STACK_SIZE, NULL,
        PRIORITY_GENERATOR, &task_generator2_handle);
473     xTaskCreate(task_generator3, "Gen 3", configMINIMAL_STACK_SIZE, NULL,
        PRIORITY_GENERATOR, &task_generator3_handle);
474
475     if(monitor_handle) vTaskSuspend(monitor_handle); // Started via timer shortly
476
477     // Timers
478     user1_timer = xTimerCreate("Timer 1", T1_PERIOD, pdTRUE, 0, task1_timer_handle
        );
479     user2_timer = xTimerCreate("Timer 2", T2_PERIOD, pdTRUE, 0, task2_timer_handle
        );
480     user3_timer = xTimerCreate("Timer 3", T3_PERIOD, pdTRUE, 0, task3_timer_handle
        );
481     monitor_timer = xTimerCreate("Monitor Timer", pdMS_TO_TICKS(500), pdTRUE, 0,
        monitor_timer_handle);
482
483     if(user1_timer) xTimerStart(user1_timer, 0);
484     if(user2_timer) xTimerStart(user2_timer, 0);
485     if(user3_timer) xTimerStart(user3_timer, 0);
486     if(monitor_timer) xTimerStart(monitor_timer, 0);
487
488     vTaskStartScheduler();
489
490     return 0;
491 }
492
493 /*-----------------------------------------------------------*/
494 void vApplicationMallocFailedHook( void ) { for( ;; ); }
495 void vApplicationStackOverflowHook( xTaskHandle pxTask, signed char *pcTaskName )
      { for( ;; ); }
496 void vApplicationIdleHook( void ) {}
497 /*-----------------------------------------------------------*/
```

Listing 5: project 2 code