VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Computer Science and Engineering

CC02 — Lab Report

Microprocessor - Microcontroller
Lab 4

Supervisors: Nguyen Thien An
Students: Nguyen Minh Dang 2252154

Ho Chi Minh City, November 26, 2024

Contents

1 Exercise

The GitHub link for the lab schematics is at here or in this link: https://github.com/dangalpha78/Workspace-for-Microprocessor---Microcontroller/tree/main/Lab4.

In this lab, I will implement the scheduler in two ways: O(n) (as guided in the file) and O(1).

The schematic for this lab is located here:

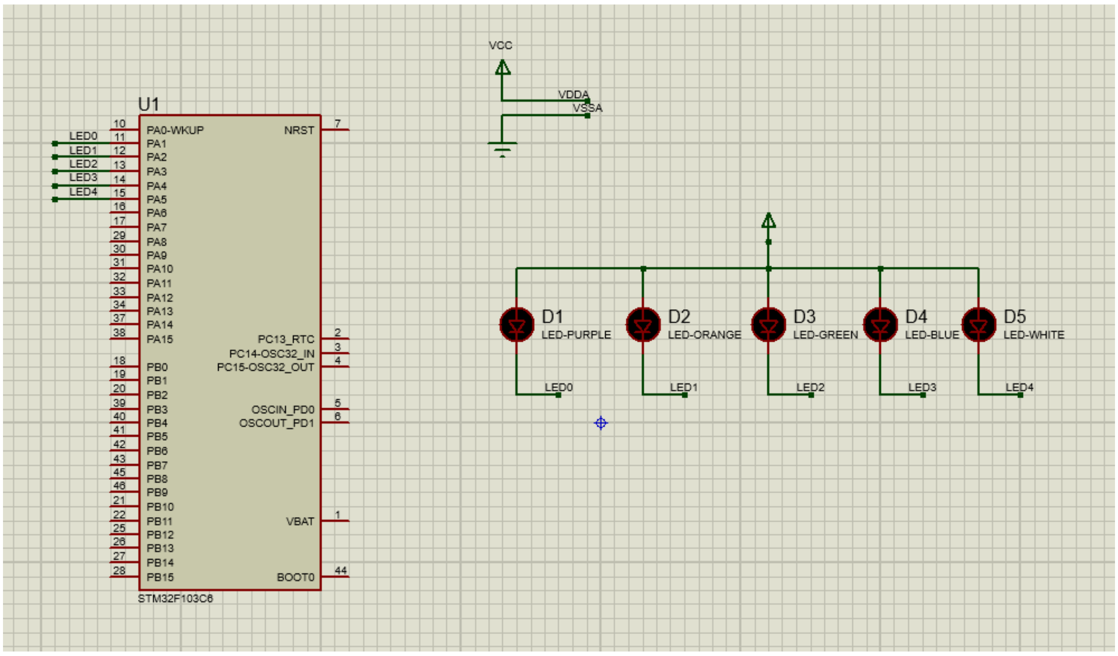

Figure 1: *The schematic for the exercises from 1 to 5.*

Explain the schematic: There are 5 LEDs, D1 to D5, which will represent 5 different tasks with different frequencies (or periods in the code).

1.1 Tasks for scheduler

These are the functions in the `led_control.c` file, with each function representing a sample task. For simplicity, these tasks are designed to make the LEDs blink.

```c
#include "main.h"
#include "timer.h"

#define LED_ON GPIO_PIN_RESET
#define LED_OFF GPIO_PIN_SET

void setUp(void){
    HAL_GPIO_WritePin(LED0_GPIO_Port, LED0_Pin, LED_OFF);
    HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, LED_OFF);
    HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, LED_OFF);
    HAL_GPIO_WritePin(LED3_GPIO_Port, LED3_Pin, LED_OFF);
    HAL_GPIO_WritePin(LED4_GPIO_Port, LED4_Pin, LED_OFF);
}

void blinky0(){
    HAL_GPIO_TogglePin(LED0_GPIO_Port, LED0_Pin);
}

void blinky1(){
    HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
}

void blinky2(){
    HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
}

void blinky3(){
    HAL_GPIO_TogglePin(LED3_GPIO_Port, LED3_Pin);
}

void blinky4(){
    HAL_GPIO_TogglePin(LED4_GPIO_Port, LED4_Pin);
}
```

1.2 Scheduler with O(n)

The source code you can find here or in this link: https://github.com/dangalpha78/Workspace-for-Microprocessor---Microcontroller/tree/main/Lab4/Lab4_Source_Code/Scheduler_O(n).

1.2.1 sched.h

```
#ifndef INC_SCHED_H_
#define INC_SCHED_H_

typedef struct {
    void (*pTask)(void);
    uint32_t Delay;
    uint32_t Period;
    uint8_t RunMe;
    uint32_t TaskID;
} struct_Task;

#define SCH_MAX_TASKS 40
#define NO_TASK_ID 0

void SCH_Init(void);
void SCH_Update(void);
void SCH_Add_Task(void (* pFunction)(), unsigned int DELAY, unsigned int PERIOD);
void SCH_Dispatch_Tasks(void);
void SCH_Delete_Task(const uint8_t TASK_INDEX);
void SCH_Go_To_Sleep();
void SCH_Report_Status(void);

#endif /* INC_SCHED_H_ */
```

1.2.2 sched.c

```
#include "main.h"
#include "sched.h"

struct_Task SCH_tasks_G[SCH_MAX_TASKS];

void SCH_Init(void) {
    unsigned char i;

    for (i = 0; i < SCH_MAX_TASKS; i++) {
        SCH_Delete_Task(i);
    }
}

void SCH_Update(void) {
```

```c
    unsigned char Index;

    for (Index = 0; Index < SCH_MAX_TASKS; Index++) {
        if (SCH_tasks_G[Index].pTask) {
            if (SCH_tasks_G[Index].Delay == 0) {
                SCH_tasks_G[Index].RunMe = 1;

                if (SCH_tasks_G[Index].Period) {
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            } else {
                SCH_tasks_G[Index].Delay--;
            }
        }
    }
}


void SCH_Add_Task(void (* pFunction)(), unsigned int DELAY, unsigned int PERIOD) {
    unsigned char Index = 0;

    while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS)) {
        Index++;
    }

    if (Index == SCH_MAX_TASKS) {
        return;
    }

    SCH_tasks_G[Index].pTask = pFunction;
    SCH_tasks_G[Index].Delay = DELAY / 10;
    SCH_tasks_G[Index].Period = PERIOD / 10;
    SCH_tasks_G[Index].RunMe = 0;

}

void SCH_Dispatch_Tasks(void) {
    unsigned char Index;

    for (Index = 0; Index < SCH_MAX_TASKS; Index++) {
        if (SCH_tasks_G[Index].RunMe > 0) {
            (*SCH_tasks_G[Index].pTask)();
            SCH_tasks_G[Index].RunMe = 0 ;

            if (SCH_tasks_G[Index].Period == 0) {
                SCH_Delete_Task(Index);
            }
        }
    }
```

```
}

void SCH_Delete_Task(const uint8_t TASK_INDEX) {
    SCH_tasks_G[TASK_INDEX].pTask = 0x0000;
    SCH_tasks_G[TASK_INDEX].Delay = 0;
    SCH_tasks_G[TASK_INDEX].Period = 0;
    SCH_tasks_G[TASK_INDEX].RunMe = 0;
}
```

1.3 Scheduler with O(1)

The source code you can find here or in this link: https://github.com/dangalpha78/Workspace-for-Microprocessor---Microcontroller/tree/main/Lab4/Lab4_Source_Code/Scheduler_O(1).

1.3.1 sched.h

For the scheduler with $O(1)$, I reformatted the `struct_Task` into a node by adding a pointer `p_next`.

```c
#ifndef INC_SCHED_H_
#define INC_SCHED_H_

#include <stdint.h>
#include <stddef.h>
#include <stdlib.h>

typedef struct struct_Task{
    void (*pTask)(void);
    uint32_t Delay;
    uint32_t Period;
    uint8_t RunMe;
    uint32_t TaskID;
    struct struct_Task* p_next;
} struct_Task;

#define SCH_MAX_TASKS 40

void SCH_Init(void);
void SCH_Update(void);
void SCH_Add_Task(void (* pFunction)(), unsigned int DELAY, unsigned int PERIOD);
void SCH_Dispatch_Tasks(void);
struct_Task* SCH_Delete_Task(struct_Task* head);
struct_Task* create_newTask(void (*pFunction)(), unsigned int DELAY, unsigned int
    PERIOD, unsigned int ID);
uint8_t get_available_ID(void);

#endif /* INC_SCHED_H_ */
```

1.3.2 sched.c

Description:

- For the $O(1)$ scheduler, I use a singly linked list instead of an array as in $O(n)$. Therefore, I added a `create_newTask` function for support and a `p_Head` pointer to manage the linked list.
- The `SCH_Add_Task` function is modified to add new tasks by creating nodes in the linked

list. A key feature here is that tasks with higher delays compared to earlier tasks are shifted further back in the list, and their delay values are adjusted accordingly. The delay of a task now represents the time interval between itself and the preceding task. For example, consider the following linked list:

A(1) - B(2) - C(5) - D(3)

In this case:

- Task A executes after 1 tick.
- Task B executes 2 ticks after task A (or 3 ticks from the starting point).
- Task C executes 5 ticks after task B (or 8 ticks from the starting point)
- Task D executes 3 ticks after task C (or 11 ticks from the starting point)

If we add a new task $E(4)$, the following adjustments occur:

- E has a delay of 4, larger than A, so it is placed after A and set to execute 3 ticks after A: $A(1) - E(3) - \ldots$.
- Since E's delay of 3 is still larger than B's delay of 2 (relative to A), E moves further back, executing 1 tick after B: $A(1) - B(2) - E(1) - \ldots$.
- At this point, E has a delay of 1, smaller than C's delay of 5. Therefore, E is placed before C, executing 4 ticks before C. The delay of C is updated relative to its new preceding task.

The resulting linked list after adding $E(4)$ is:

A(1) - B(2) - E(1) - C(4) - D(3)

- The `get_available_ID` function is used to manage task IDs. Each task has a fixed ID that does not change with subsequent task additions.
- The `p_Update` pointer is used to point to the node containing the currently executing task. Once the task is executed, `p_Update` moves to the next node. The `SCH_Dispatch_Tasks` function checks if the `RunMe` flag of the current node is set. If the flag is set, the task in the node is executed. After execution, the task's ID is reset, and if the task has a period, it is repeated by adding it as a new node to the linked list while retaining the same ID. Finally, the `p_Update` pointer is updated to the next node, and the current node (the node containing the executed task or, in other words, the head node) is deleted using the `SCH_Delete_Task` function. This function deletes the first node and updates the head node accordingly.
- The `SCH_Update` function has a time complexity of $O(1)$ because all operations within the function are independent of the size of the data. The function performs checks and updates on a single node pointed to by the `p_Update` pointer. Specifically, it checks the condition `p_Update` and `p_Update->pTask`, then either decrements the `Delay` value or updates the `RunMe` flag for the task without any loops or computations depending on the number of

nodes or other factors. Therefore, the execution time of the function is constant and does not change with the number of tasks in the system, ensuring a time complexity of $O(1)$.

```
#include "sched.h"

uint8_t a_taskID[SCH_MAX_TASKS];
struct_Task* p_Head;
struct_Task* p_Update;

uint8_t get_available_ID() {
    for (uint8_t i = 0; i < SCH_MAX_TASKS; i++) {
        if (a_taskID[i] == 0) {
          a_taskID[i] = 1;
            return i;
        }
    }
    return -1;
}

struct_Task* create_newTask(void (*pFunction)(), unsigned int DELAY, unsigned int
    PERIOD, unsigned int ID){
  struct_Task* newTask = (struct_Task*)malloc(sizeof(struct_Task));
  if (newTask){
    newTask->pTask = pFunction;
    newTask->Delay = DELAY / 10;
    newTask->Period = PERIOD;
    newTask->RunMe = 0;
    newTask->TaskID = ID;
    newTask->p_next = NULL;
  }
  return newTask;
}

void SCH_Init(void) {
  unsigned char i;

  for (i = 0; i < SCH_MAX_TASKS; i++){
    a_taskID[i] = 0;
  }
    p_Head = NULL;
    p_Update = NULL;

}

void SCH_Update(void) {
    if (p_Update && p_Update->pTask){
        if (p_Update->Delay <= 0){
            p_Update->RunMe += 1;
        }
        else {
```

```
          p_Update->Delay--;
        }
    }
}

void SCH_Add_Task(void (* pFunction)(), unsigned int DELAY, unsigned int PERIOD) {
  uint8_t ID = get_available_ID();
      struct_Task* newTask = create_newTask(pFunction, DELAY, PERIOD, ID);
      if (!newTask) return;

      if (p_Head == NULL){
        p_Head = newTask;
        p_Update = newTask;
      }
      else {
        struct_Task* current = p_Head;
        struct_Task* prev = NULL;

        while (current && newTask->Delay >= current->Delay){
          newTask->Delay -= current->Delay;
          prev = current;
          current = current->p_next;
        }

        if (prev == NULL){
          newTask->p_next = p_Head;
          p_Head = newTask;
          current->Delay -= newTask->Delay;
        }
        else if (current == NULL){
          prev->p_next = newTask;
        }
        else {
          newTask->p_next = current;
          prev->p_next = newTask;
          current->Delay -= newTask->Delay;
        }
      }
}

struct_Task* SCH_Delete_Task(struct_Task* head) {
    if (head == NULL) return NULL;

    struct_Task* temp = head;

    head = head->p_next;
    free(temp);

    return head;
```

```
}

void SCH_Dispatch_Tasks(void) {
    if (p_Update && p_Update->RunMe > 0){
      (*p_Update->pTask)();

      a_taskID[p_Update->TaskID] = 0;
      if(p_Update->Period > 0) SCH_Add_Task(p_Update->pTask, p_Update->Period,
    p_Update->Period);
      p_Update = p_Update->p_next;
      p_Head = SCH_Delete_Task(p_Head);
    }
}
```

1.4 Timer and Main

1.4.1 timer.c

`SCH_Update` will be call every 10 ms.

```
#include "main.h"
#include "sched.h"

void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef *htim ) {
    if(htim->Instance == TIM2){
        SCH_Update();
    }
}
```

1.4.2 main.c

This code initializes the scheduler with `SCH_Init()` and adds several tasks using `SCH_Add_Task()`. The `setUp` task is added for initial configuration, while the `blinkyLed0` task will execute after 1 second but only once, as it is a one-shot task (it does not have a period). The `blinkyLed1` to `blinkyLed4` tasks will blink at intervals of 500ms, 1000ms, 1500ms, and 2000ms, respectively. The infinite `while` loop continuously calls `SCH_Dispatch_Tasks()`, which checks and executes tasks based on their scheduled times.

```
int main(void)
{
    HAL_TIM_Base_Start_IT(&htim2);

    SCH_Init();
    SCH_Add_Task(setUp, 0, 0);

    SCH_Add_Task(blinkyLed0, 1000, 0);
    SCH_Add_Task(blinkyLed1, 0, 500);
    SCH_Add_Task(blinkyLed2, 0, 1000);
    SCH_Add_Task(blinkyLed3, 0, 1500);
    SCH_Add_Task(blinkyLed4, 0, 2000);
    while (1)
    {
        SCH_Dispatch_Tasks();
    }
}
```

References