VIETNAM INTERNATIONAL UNIVERSITY
HO CHI MINH UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Microprocessors-Microcontrollers (CO3010)
LAB Report

Student Name	Student ID
Tran Chi An	2352080

Contents

1 Overview

Before discussing the scheduler components, we consider how the scheduler will typically appear to the user. To do this we will use a simple example: a scheduler used to flash a single LED on and off repeatedly: on for one second off for one second etc.

```
int main(void){
    //Init all the requirments for the system to run
  System_Initialization();
  //Init a schedule
  SCH_Init();
  //Add a task to repeatly call in every 1 second.
  SCH_Add_Task(Led_Display, 0, 1000);
  while (1){
    SCH_Dispatch_Tasks();
  }
  return 0;
}
```

Program 1: Example of how to use a scheduler

- We assume that the LED will be switched on and off by means of a 'task' Led_Display(). Thus, if the LED is initially off and we call Led_Display() twice, we assume that the LED will be switched on and then switched off again.

  To obtain the required flash rate, we therefore require that the scheduler calls Led_Display() every second ad infinitum.

- We prepare the scheduler using the function SCH_Init().

- After preparing the scheduler, we add the function Led_Display() to the scheduler task list using the SCH_Add_Task() function. At the same time we specify that the LED will be turned on and off at the required rate as follows:

  SCH_Add_Task(Led_Display, 0, 1000);

  We will shortly consider all the parameters of SCH_Add_Task(), and examine its internal structure.

- The timing of the Led_Display() function will be controlled by the function SCH_Update(), an interrupt service routine triggered by the overflow of Timer 2:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
  SCH\_Update();
}
```

Program 2: Example of how to call SCH_Update function

- The 'Update' function does not execute the task: it calculates when a task is due to run and sets a flag. The job of executing LED_Display() falls to the dispatcher function (SCH_Dispatch_Tasks()), which runs in the main ('super') loop:

```
while(1){
    SCH_Dispatch_Tasks();
}
```

Before considering these components in detail, we should acknowledge that this is, undoubtedly, a complicated way of flashing an LED: if our intention were to develop an LED flasher application that requires minimal memory and minimal code size, this would not be a good solution. However, the key point is that we will be able to use the same scheduler architecture in all our subsequent examples, including a number of substantial and complex applications and the effort required to understand the operation of this environment will be rapidly repaid.

It should also be emphasized that the scheduler is a 'low-cost' option: it consumes a small percentage of the CPU resources (we will consider precise percentages shortly). In addition, the scheduler itself requires no more than 17 bytes of memory for each task. Since a typical application will require no more than four to six tasks, the task – memory budget (around 60 bytes) is not excessive, even on an 8-bit microcontroller.

1.1 The scheduler data structure and task array

At the heart of the scheduler is the scheduler data structure: this is a user-defined data type which collects together the information required about each task.

```
typedef struct sTask {
    void (* pTask)(void);
    uint32_t Delay;
    uint32_t Period;
    uint8_t RunMe;
    uint32_t TaskID;

    struct sTask* next;
} sTask;



#define SCH_MAX_TASKS 40
#define NO_TASK_ID     (SCH_MAX_TASKS)
```

Program 3: Source code

1.2 The initialization function

```
void SCH_Init(void) {
    uint8_t i;
    for (i = 0; i < SCH_MAX_TASKS; i++) {
        SCH_tasks_Pool[i].pTask = NULL;
        SCH_tasks_Pool[i].next = NULL;
    }
    SCH_Task_List_Head = NULL;
}
```

Program 4: SCH Init

Explanation:

This is the initialization function and must be called first.

It iterates through the 40 slots of the SCH tasks Pool array and assigns pTask = NULL to each slot. This marks all slots as "empty," ready to receive new tasks.

SCH Task List Head is set to NULL, indicating that the list of waiting tasks is initially empty.

1.3 static uint32_t find_empty_slot(void)

```
static uint32_t find_empty_slot(void) {
    uint32_t Index = 0;
    while ((SCH_tasks_Pool[Index].pTask != NULL) && (Index < SCH_MAX_TASKS))
    {
        Index++;
    }
    return Index;
}
```

Program 5: Find Empty Slot

Explanation:
This is an internal (static) "helper" function.
It iterates through the Pool array starting from Index = 0 to find the first empty slot (a slot where pTask == NULL).
If found, it returns the Index of that slot.
If it iterates through the entire array and finds no empty slots, it returns Index = 40 (which is NO_TASK_ID).

1.4 static void SCH_Insert_Task_Into_List(sTask* taskToInsert)

```
static void SCH_Insert_Task_Into_List(sTask* taskToInsert) {
    uint32_t DELAY = taskToInsert->Delay;

    // NULL list
    if (SCH_Task_List_Head == NULL) {
        taskToInsert->next = NULL;
        SCH_Task_List_Head = taskToInsert;
        return;
    }

    // Insert before head
    if (DELAY < SCH_Task_List_Head->Delay) {
        taskToInsert->next = SCH_Task_List_Head;
        SCH_Task_List_Head->Delay = SCH_Task_List_Head->Delay - DELAY;
        SCH_Task_List_Head = taskToInsert;
        return;
    }

    // Insert in the mid or tail
    sTask* current = SCH_Task_List_Head;
```

```
    uint32_t accumulatedDelay = current->Delay;
    uint32_t remainingDelay = DELAY;

    if (remainingDelay >= accumulatedDelay) {
        remainingDelay -= accumulatedDelay;
    }

    while (current->next != NULL) {
        if (remainingDelay < current->next->Delay) {
            taskToInsert->Delay = remainingDelay;
            taskToInsert->next = current->next;
            current->next->Delay = current->next->Delay - remainingDelay;
            current->next = taskToInsert;
            return;
        }

        accumulatedDelay += current->next->Delay;
        if (remainingDelay >= current->next->Delay) {
            remainingDelay -= current->next->Delay;
        }
        current = current->next;
    }

    // Insert in to tail
    taskToInsert->Delay = remainingDelay;
    taskToInsert->next = NULL;
    current->next = taskToInsert;
}
```

Program 6: Insert Task into Delta List

Explanation:
This is the most critical logic function. It inserts a task into the linked list in the correct chronological order using the "Delta Delay" technique.
Case 1 (Empty List): Sets the SCH_Task_List_Head to point to taskToInsert.
Case 2 (Insert at Head): If the new task's DELAY is less than the Head's Delay, the new task becomes the new Head. The old Head's Delay is updated (to Old_Delay - New_Delay).
Case 3 (Insert in Middle/Tail): The function iterates the list, using remainingDelay to count down. When it finds the correct insertion point (e.g., between task A and B), it updates the new task's Delay and task B's Delay (patching the Delta chain) to ensure the total time remains constant.

1.5 uint32_t SCH_Add_Task(...)

```
uint32_t SCH_Add_Task(void (*pFunction)(), uint32_t DELAY, uint32_t PERIOD)
    {
    uint32_t Index = find_empty_slot();
    if (Index == NO_TASK_ID) {
        return NO_TASK_ID;
    }
```

```
6
7       sTask* newTask = &SCH_tasks_Pool[Index];
8       newTask->pTask = pFunction;
9       newTask->Period = PERIOD;
10      newTask->RunMe = 0;
11      newTask->TaskID = Index;
12      newTask->Delay = DELAY;
13
14      SCH_Insert_Task_Into_List(newTask);
15
16      return Index;
17  }
```

Program 7: Add Task

Explanation:
This is the public function for adding a new task.
Step 1: Calls find_empty_slot() to get an available slot from the Pool.
Step 2: Fills in the task's information (pFunction, Period, TaskID...) into the newTask slot at the found Index.
Step 3: Sets newTask->Delay = DELAY (the absolute time).
Step 4: Calls SCH_Insert_Task_Into_List(newTask) to insert this task into the waiting list in the correct order.

1.6 void SCH_Update(void)

```
1   void SCH_Update(void) {
2       if (SCH_Task_List_Head == NULL) {
3           return;
4       }
5
6       if (SCH_Task_List_Head->Delay == 0) {
7           while (SCH_Task_List_Head != NULL && SCH_Task_List_Head->Delay == 0)
        {
8
9               sTask* expiredTask = SCH_Task_List_Head;
10              expiredTask->RunMe += 1;
11
12              SCH_Task_List_Head = expiredTask->next;
13              expiredTask->next = NULL;
14
15              if (expiredTask->Period > 0) {
16                  expiredTask->Delay = expiredTask->Period;
17                  SCH_Insert_Task_Into_List(expiredTask);
18              }
19          }
20      }
21      if (SCH_Task_List_Head != NULL) {
22          SCH_Task_List_Head->Delay--;
23      }
24  }
```

Program 8: Update Scheduler (O(1))

Explanation (Time Complexity O(1)):
This is the O(1) function, called every 10ms from the timer interrupt.
Step 1 (Check Delay=0): It only checks the Delay of the first task (Head).
Step 2 (Process Expiry): If Delay == 0, it enters a while loop (to handle multiple tasks expiring in the same tick).
It removes the task (expiredTask) from the Head of the list.
It increments the flag expiredTask->RunMe = 1 (signaling to Dispatch that this task is ready to run).
It updates the Head to point to the next task.
Step 3 (Re-schedule): If expiredTask is a periodic task (Period > 0), the function resets expiredTask->Delay = expiredTask->Period and calls SCH_Insert_Task_Into_List to re-insert the same task object back into the queue (this prevents the "Task Leak" bug).
Step 4 (Decrement Delay): After processing all expired tasks, if the Head is still not NULL, the function performs a single subtraction: SCH_Task_List_Head->Delay–.

1.7 void SCH_Dispatch_Tasks(void)

```
void SCH_Dispatch_Tasks(void) {
    unsigned char Index;
    for (Index = 0; Index < SCH_MAX_TASKS; Index++) {

        if (SCH_tasks_Pool[Index].RunMe > 0) {

            (*SCH_tasks_Pool[Index].pTask)();
            SCH_tasks_Pool[Index].RunMe -= 1;

            if (SCH_tasks_Pool[Index].Period == 0) {
                SCH_tasks_Pool[Index].pTask = NULL;
            }
        }
    }
}
```

Program 9: Dispatch Tasks

Explanation:
This function is called continuously in the while(1) loop of main.
It iterates through the entire 40-slot SCH_tasks_Pool array.
If it finds a task with the RunMe > 0 flag (meaning SCH_Update has cleared it to run):
1. It executes the task by calling its function pointer: (*SCH_tasks_Pool[Index].pTask)();.
2. It decrements the RunMe flag.
3. If the task is "one-shot" (Period == 0), it frees the slot by setting pTask = NULL.

1.8 uint8_t SCH_Delete_Task(uint32_t taskID)

```
uint8_t SCH_Delete_Task(uint32_t taskID) {
    if (taskID >= NO_TASK_ID || SCH_tasks_Pool[taskID].pTask == NULL) {
        return 0;
```

```
    }

    sTask* taskToDelete = &SCH_tasks_Pool[taskID];

    if (taskToDelete == SCH_Task_List_Head) {
        if (taskToDelete->next != NULL) {
            taskToDelete->next->Delay += taskToDelete->Delay;
        }
        SCH_Task_List_Head = taskToDelete->next;
    }
    else {
        sTask* current = SCH_Task_List_Head;
        while (current != NULL && current->next != taskToDelete) {
            current = current->next;
        }

        if (current != NULL) {
            sTask* nextNode = taskToDelete->next;
            if (nextNode != NULL) {
                nextNode->Delay += taskToDelete->Delay;
            }
            current->next = nextNode;
        }
    }

    taskToDelete->pTask = NULL;
    taskToDelete->Delay = 0;
    taskToDelete->Period = 0;
    taskToDelete->RunMe = 0;
    taskToDelete->next = NULL;

    return 1;
}
```

Program 10: Delete Task

Explanation:
This function deletes a task based on its taskID (which is also its Index in the Pool).
Step 1 (Remove from List): The most complex part is iterating through the Linked List to find taskToDelete. When found, it must "patch" the Delta chain by adding the deleted task's Delay to the Delay of the task immediately following it.
Step 2 (Remove from Pool): It frees the slot by setting taskToDelete->pTask = NULL.

2 Analysis of main.c (Added Sections)

This analysis covers the code added to main.c to use the scheduler library.

2.1 Declarations and Task Functions

```c
/* USER CODE BEGIN PV */
volatile uint32_t g_system_time_ms = 0;
/* USER CODE END PV */
...
/* USER CODE BEGIN 0 */

uint32_t get_time(void) {
    return g_system_time_ms;
}

void Task_Print_Time(void) {
    uint32_t time_ms = get_time();
    char tx_buffer[50];
    sprintf(tx_buffer, "Time: %lu ms\r\n", time_ms);
    HAL_UART_Transmit(&huart1, (uint8_t*)tx_buffer, strlen(tx_buffer), 100);
}

void Task1_Run(void) { HAL_GPIO_TogglePin(LED_RED_1_GPIO_Port, LED_RED_1_Pin
    ); }
void Task2_Run(void) { HAL_GPIO_TogglePin(LED_RED_2_GPIO_Port, LED_RED_2_Pin
    ); }
void Task3_Run(void) { HAL_GPIO_TogglePin(LED_RED_3_GPIO_Port, LED_RED_3_Pin
    ); }
void Task4_Run(void) { HAL_GPIO_TogglePin(LED_RED_4_GPIO_Port, LED_RED_4_Pin
    ); }
void Task5_Run(void) { HAL_GPIO_TogglePin(LED_RED_5_GPIO_Port, LED_RED_5_Pin
    ); }
void Task6_Run(void) { HAL_GPIO_TogglePin(LED_RED_6_GPIO_Port, LED_RED_6_Pin
    ); }
/* USER CODE END 0 */
```

Program 11: main.c - Declarations and Tasks

Explanation:
g_system_time_ms: A global (volatile) variable used to count system time, incremented by 10ms in each timer interrupt.
get_time(): A function that returns the value of g_system_time_ms
Task1_Run -> Task5_Run: 5 periodic tasks that toggle 5 LEDs (PA1-PA5) at different intervals to demonstrate multitasking.
Task6_Run: A "one-shot" task that toggles LED 6 (PA7).
Task_Print_Time: A periodic task that uses sprintf to create a time string and HAL_UART_Transmit (with a 100ms timeout to prevent blocking) to send the time over UART1 to the Virtual Terminal.

2.2 Initialization and Main Loop (in main())

```c
  /* USER CODE BEGIN 2 */
  SCH_Init();

  SCH_Add_Task(Task1_Run, 0, 50);
```

```
  SCH_Add_Task(Task2_Run, 1, 100);
  SCH_Add_Task(Task3_Run, 2, 150);
  SCH_Add_Task(Task4_Run, 3, 200);
  SCH_Add_Task(Task5_Run, 4, 250);
  //one-shot
  SCH_Add_Task(Task6_Run, 500, 0);

  SCH_Add_Task(Task_Print_Time, 10, 100);
  HAL_TIM_Base_Start_IT(&htim2);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    SCH_Dispatch_Tasks();
    /* USER CODE END WHILE */
...
```

Program 12: main.c - Initialization

Explanation:
SCH_Init(): Calls the scheduler's initialization function (mandatory).
SCH_Add_Task(...): Adds 7 tasks to the system:
5 periodic LED tasks (0.5s, 1s, 1.5s, 2s, 2.5s) with small initial delays (0, 1, 2, 3, 4 ticks)
to avoid simultaneous execution at startup.
1 "one-shot" LED task to run once after 500 ticks (5 seconds).
1 time-printing task, to run after 10 ticks (100ms) and repeat every 100 ticks (1 second).
HAL_TIM_Base_Start_IT(&htim2): Activates the 10ms timer interrupt (TIM2).
while (1): The Super Loop, which only calls SCH_Dispatch_Tasks() to dispatch any ready-to-run tasks.

2.3 Timer Interrupt Function (ISR)

```
/* USER CODE BEGIN 4 */
extern volatile uint32_t g_system_time_ms;

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
  SCH_Update();
  g_system_time_ms += 10;
}
/* USER CODE END 4 */
```

Program 13: main.c - Timer ISR

Explanation:
This function is automatically called every time TIM2 interrupts (every 10ms).
SCH_Update(): Calls the O(1) "heartbeat" function of the scheduler to update the task queue.
g_system_time_ms += 10: Increments the global time-keeping variable by 10ms.

3 GITHUB LINK:

- Github link to WHOLE Project

- Github link to scheduler_o1.c

- Github link to main.c

4 Proteus:

Report: Schematic from Proteus

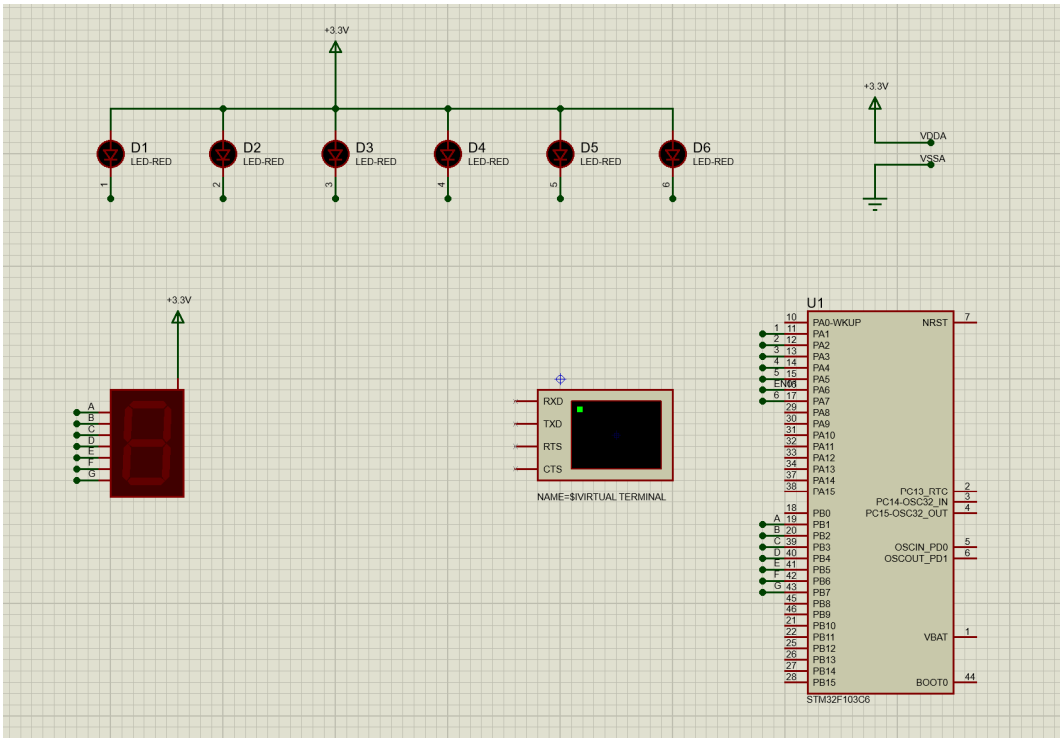

Figure 1: Simulation schematic in Proteus

- Github link to Proteus