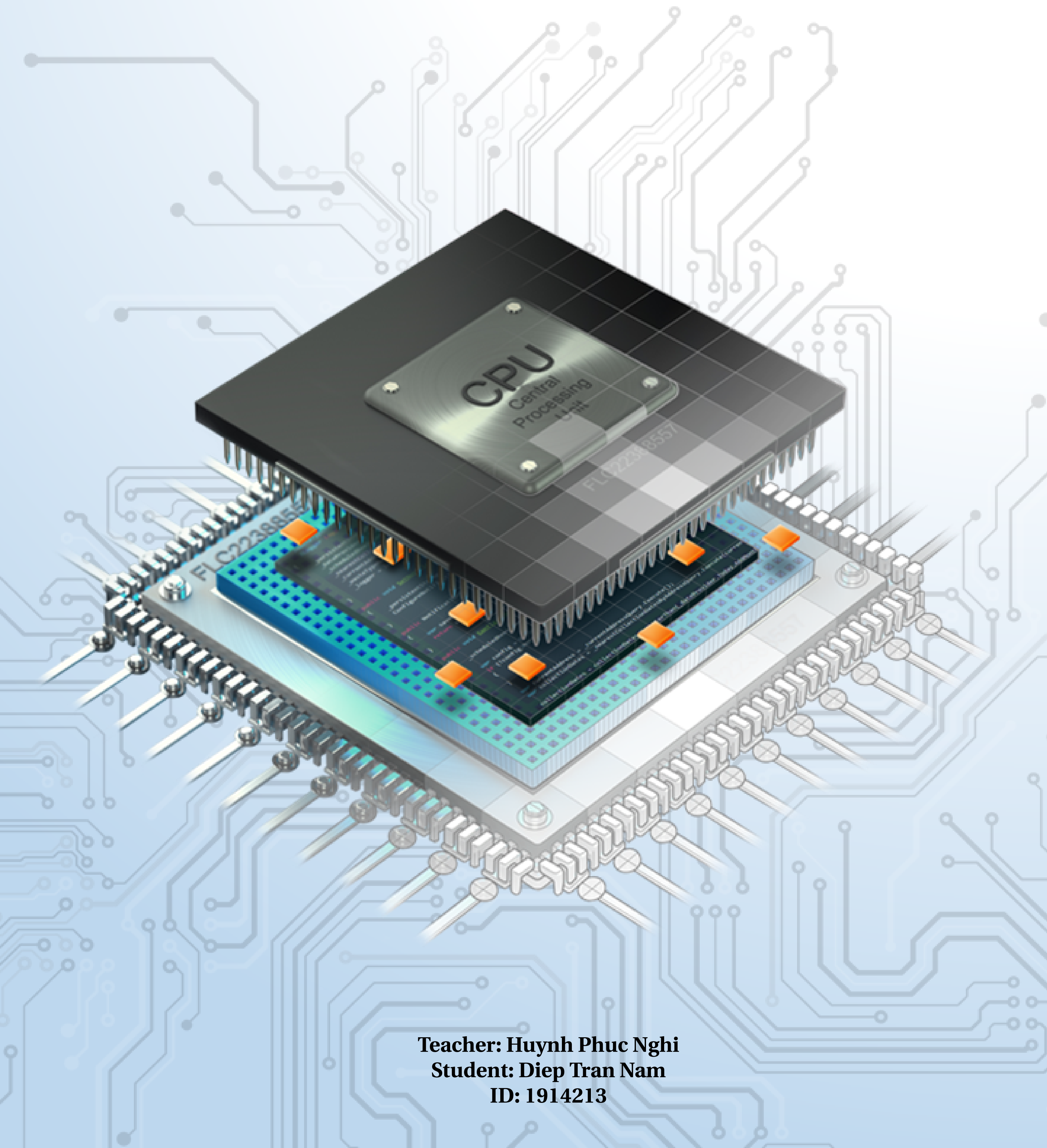
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
COMPUTER ENGINEERING

Microcontroller

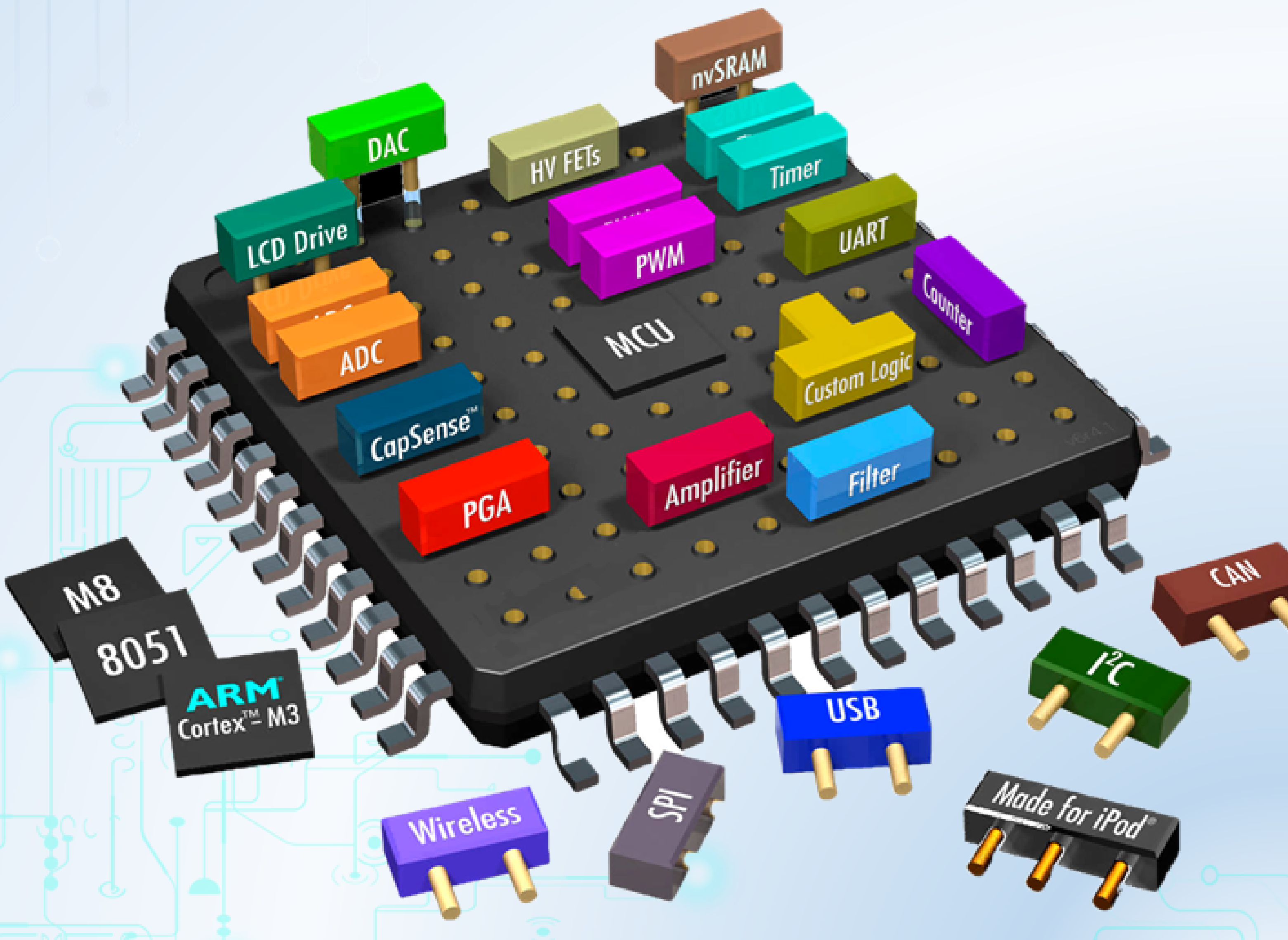


Teacher: Huynh Phuc Nghi
Student: Diep Tran Nam
ID: 1914213

Mục lục

CHƯƠNG 1

A cooperative scheduler



1 Introduction

1.1 Super Loop Architecture

```c
}
void main(void){
    // Prepare for Task X
    X_Init();
    while(1) {// 'for ever' (Super Loop)
        X(); // Perform the task
    }
}
```

Program 1.1: Super loop program

The main advantages of the Super Loop architecture illustrated above are:

- (1) that it is simple, and therefore easy to understand, and
- (2) that it consumes virtually no system memory or CPU resources.

However, we get 'nothing for nothing': Super Loops consume little memory or processor resources because they provide few facilities to the developer. A particular limitation with this architecture is that it is very difficult to execute Task X at precise intervals of time: as we will see, this is a very significant drawback.
For example, consider a collection of requirements assembled from a range of different embedded projects (in no particular order):

- The current speed of the vehicle must be measured at 0.5 second intervals.
- The display must be refreshed 40 times every second.
- The calculated new throttle setting must be applied every 0.5 seconds.
- A time-frequency transform must be performed 20 times every second.
- If the alarm sounds, it must be switched off (for legal reasons) after 20 minutes.
- If the front door is opened, the alarm must sound in 30 seconds if the correct password is not entered in this time.
- The engine vibration data must be sampled 1,000 times per second.
- The frequency-domain data must be classified 20 times every second.
- The keypad must be scanned every 200 ms.
- The master (control) node must communicate with all other nodes (sensor nodes and sounder nodes) once per second.
- The new throttle setting must be calculated every 0.5 seconds.
- The sensors must be sampled once per second.

We can summarize this list by saying that many embedded systems must carry out tasks at particular instants of time. More specifically, we have two kinds of activity to perform:

- Periodic tasks, to be performed (say) once every 100 ms
- One-shot tasks, to be performed once after a delay of (say) 50 ms

This is very difficult to achieve with the primitive architecture shown in Program above. Suppose, for example, that we need to start Task X every 200 ms, and that the task takes 10 ms to complete. Program below illustrates one way in which we might adapt the code in order to try to achieve this.

```
}
void main(void){
    // Prepare for Task X
    X_Init();
    while(1) {            // 'for ever' (Super Loop)
        X();              // Perform the task (10 ms duration)
        Delay_190ms();    // Delay for 190 ms
    }
}
```

Program 1.2: Trying to use the Super Loop architecture to execute tasks at regular intervals

The approach is not generally adequate, because it will only work if the following conditions are satisfied:

- We know the precise duration of Task X
- This duration never varies

In practical applications, determining the precise task duration is rarely straightforward. Suppose we have a very simple task that does not interact with the outside world but, instead, performs some internal calculations. Even under these rather restricted circumstances, changes to compiler optimization settings – even changes to an apparently unrelated part of the program – can alter the speed at which the task executes. This can make fine-tuning the timing very tedious and error prone.
The second condition is even more problematic. Often in an embedded system the task will be required to interact with the outside world in a complex way. In these circumstances the task duration will vary according to outside activities in a manner over which the programmer has very little control.

1.2 Timer-based interrupts and interrupt service routines

A better solution to the problems outlined is to use timer-based interrupts as a means of invoking functions at particular times.
An interrupt is a hardware mechanism used to notify a processor that an 'event' has taken place: such events may be internal events or external events.
When an interrupt is generated, the processor 'jumps' to an address at the bottom of the CODE memory area. These locations must contain suitable code with which the microcontroller can respond to the interrupt or, more commonly, the locations

will include another ‘jump’ instruction, giving the address of suitable ‘interrupt service routine’ located elsewhere in (CODE) memory.
Please see lab 3 for the more information of this approach.

2 What is a scheduler?

There are two ways of viewing a scheduler:

- At one level, a scheduler can be viewed as a simple operating system that allows tasks to be called periodically or (less commonly) on a one-shot basis.
- At a lower level, a scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks. As a result, only one timer needs to be initialized, and any changes to the timing generally requires only one function to be altered. Furthermore, we can generally use the same scheduler whether we need to execute one, ten or 100 different tasks.

```c
void main(void) {
    // Set up the scheduler
    SCH_Init();
    // Add the tasks (1ms tick interval)
    // Function_A will run every 2 ms
    SCH_Add_Task(Function_A, 0, 2);
    // Function_B will run every 10 ms
    SCH_Add_Task(Function_B, 1, 10);
    // Function_C will run every 15 ms
    SCH_Add_Task(Function_C, 3, 15);
    while(1) {
        SCH_Dispatch_Tasks();
    }
}
```

Program 1.3: Example of how a scheduler uses

2.1 The co-operative scheduler

A co-operative scheduler provides a single-tasking system architecture
Operation:

- Tasks are scheduled to run at specific times (either on a periodic or one-shot basis)
- When a task is scheduled to run it is added to the waiting list
- When the CPU is free, the next waiting task (if any) is executed
- The task runs to completion, then returns control to the scheduler

Implementation:

- The scheduler is simple and can be implemented in a small amount of code

- The scheduler must allocate memory for only a single task at a time
- The scheduler will generally be written entirely in a high-level language (such as ‘C’)
- The scheduler is not a separate application; it becomes part of the developer’s code

Performance:

- Obtaining rapid responses to external events requires care at the design stage Reliability and safety:

Co-operate scheduling is simple, predictable, reliable and safe
A co-operative scheduler provides a simple, highly predictable environment. The scheduler is written entirely in ‘C’ and becomes part of the application: this tends to make the operation of the whole system more transparent and eases development, maintenance and porting to different environments. Memory overheads are 17 bytes per task and CPU requirements (which vary with tick interval) are low.

2.2 Function pointers

One area of the language with which many ‘C’ programmers are unfamiliar is the function pointer. While comparatively rarely used in desktop programs, this language feature is crucial in the creation of schedulers: we therefore provide a brief introductory example here.
The key point to note is that – just as we can, for example, determine the starting address of an array of data in memory – we can also find the address in memory at which the executable code for a particular function begins. This address can be used as a ‘pointer’ to the function; most importantly, it can be used to call the function. Used with care, function pointers can make it easier to design and implement complex programs. For example, suppose we are developing a large, safety-critical, application, controlling an industrial plant. If we detect a critical situation, we may wish to shut down the system as rapidly as possible. However, the appropriate way to shut down the system will vary, depending on the system state. What we can do is create a number of different recovery functions and a function pointer. Every time the system state changes, we can alter the function pointer so that it is always pointing to the most appropriate recovery function. In this way, we know that – if there is ever an emergency situation – we can rapidly call the most appropriate function, by means of the function pointer.

```c
// ------ Private function prototypes ----------------------------
void Square_Number(int, int*);

int main(void)
{
    int a = 2, b = 3;
    /* Declares pFn to be a pointer to fn with
    int and int pointer parameters (returning void) */
    void (* pFn)(int, int*);

    int Result_a, Result_b;
    pFn = Square_Number; // pFn holds address of Square_Number
    printf("Function code starts at address: %u\n", (tWord) pFn);
    printf("Data item a starts at address: %u\n\n", (tWord) &a);
    // Call ‘Square_Number’ in the conventional way
    Square_Number(a, &Result_a);
    // Call ‘Square_Number’ using function pointer
    (*pFn)(b,&Result_b);
    printf("%d squared is %d (using normal fn call)\n", a, Result_a
   );
    printf("%d squared is %d (using fn pointer)\n", b, Result_b);
    while(1);
    return 0;
}

void Square_Number(int a, int* b)
{// Demo - calculate square of a
    *b = a * a;
}
```

Program 1.4: Example of how to use function pointers

2.3 Solution

A scheduler has the following key components:

- The scheduler data structure.
- An initialization function.
- A single interrupt service routine (ISR), used to update the scheduler at regular time intervals.
- A function for adding tasks to the scheduler.
- A dispatcher function that causes tasks to be executed when they are due to run.
- A function for removing tasks from the scheduler (not required in all applications).

2.3.1 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 1.5: 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 1.6: 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.

2.3.2 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.

```c

typedef struct {
    // Pointer to the task (must be a 'void (void)' function)
  void ( * pTask)(void);
  // Delay (ticks) until the function will (next) be run
  int32_t Delay;
  // Interval (ticks) between subsequent runs.
  uint32_t Period;
  // Incremented (by scheduler) when task is due to execute
  uint8_t RunMe;
  //This is a hint to solve the question below.
  uint32_t TaskID;
} sTask;

// MUST BE ADJUSTED FOR EACH NEW PROJECT
#define SCH_MAX_TASKS        10
#define NO_TASK_ID           0
sTask SCH_tasks_G[SCH_MAX_TASKS];
```

Program 1.7: A struct of a task

The size of the task array
You must ensure that the task array is sufficiently large to store the tasks required in your application, by adjusting the value of SCH_MAX_TASKS. For example, if you schedule three tasks as follows:

- SCH_Add_Task(Function_A, 0, 2);
- SCH_Add_Task(Function_B, 1, 10);
- SCH_Add_Task(Function_C, 3, 15);

then SCH_MAX_TASKS must have a value of three (or more) for correct operation of the scheduler.

2.3.3 The idea

To implement a *SCH_Update()* function with O(1) complexity, the idea is that we only consider the first element in the array when calling *SCH_Update()* function. So, the rest of the elements will have *Delay* value depending on the first element. Specifically, we first add tasks to the array based on their *Delay* value. After adding, we proceed to run the tasks in the order added. When a task is finished running, we will assign the task's *Delay* to its *Period* value and move it behind and rearrange it to a suitable position (based on the updated delay value), then run the next task. So, this makes the *SCH_Add_Task()* and the *SCH_Dispatch_Tasks()* function more complicated.

Details of adding a task to the array:

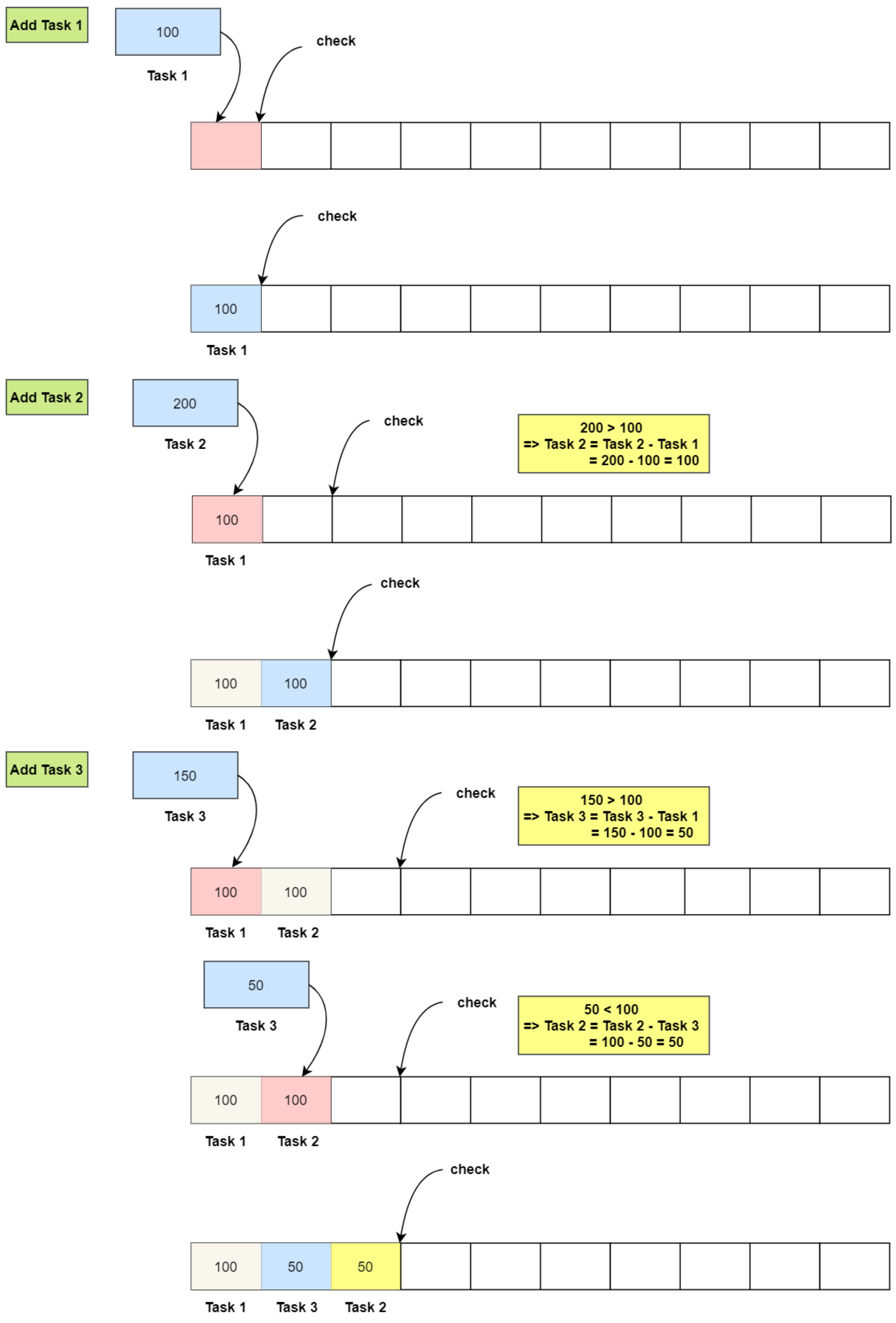
We have a *check* variable to find the first space in the array. If *check* is equal to *SCH_MAX_TASKS*, it means the array is full, no new tasks can be added.
Otherwise, suppose we have 3 tasks:

- Task1(Delay = 100, Period = 200)
- Task2(Delay = 200, Period = 150)
- Task3(Delay = 150, Period = 100)

Add these tasks into the array:

- Task 1: Because currently the array is empty, we add *task 1* at the beginning of the array.
- Task 2: First, we compare *task 2* with *task 1*, we see that the value of *task 2* is greater than *task 1* (200 > 100), so we subtract *task 2* for *task 1* (200 - 100 = 100) and then continue to compare with the following elements. Since there are no more elements behind, we add *task 2* after *task 1* (at *check* position).
- Task 3: Similar to *task 2*, we first compare *task 3* with *task 1*, we see that the value of *task 3* is greater than *task 1* (150 > 100), so we subtract *task 3* for *task 1* (150 - 100 = 50) and then continue to compare with the following elements. When compare *task 3* with *task 2*, we see that the value of *task 3* is smaller than *task 2* (50 < 100), so we add *task 3* before *task 2* and it is important to update the value of *task 2* (the value of *task 2* is now 100 - 50 = 50).

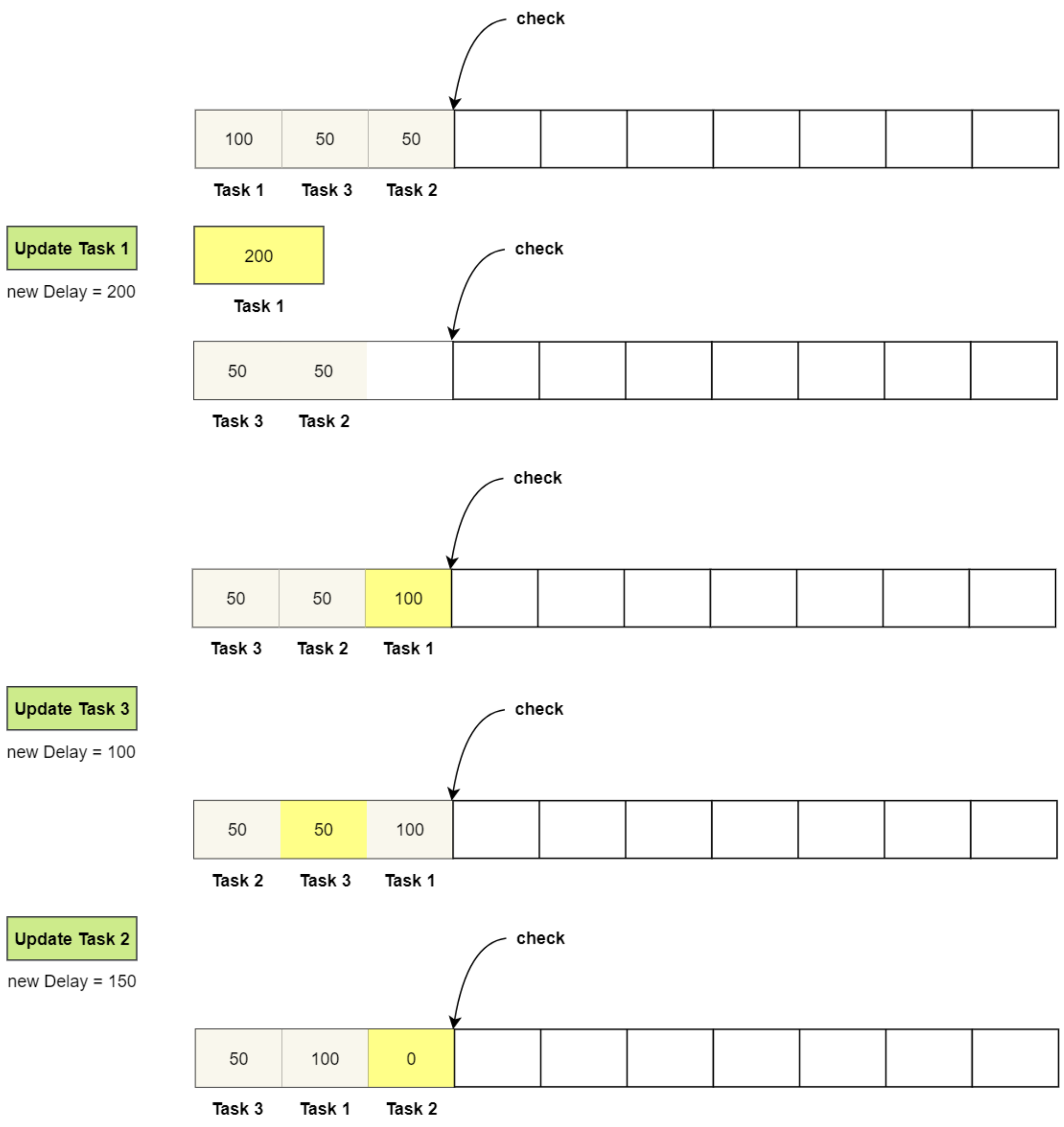
So, we have finished adding the tasks to the array with the value of the following tasks depending on the value of the previous tasks. And this is beneficial for implementing the *SCH_Update()* function with O(1) complexity.



Hình 1.1: Steps of Adding tasks.

After adding 3 tasks to the array, we get the order of *task 1*, *task 3* and then *task 2*. Next we will run and update these tasks:

- In order, *task 1* will be executed first. After *task 1* is finished running, we will assign the *Period* value of *task 1* to its *Delay*. Now, the *Delay* value of *task 1* is 200, we proceed to rearrange it similar to adding a task to the array (compare each element in the array and update the delay values and then insert the appropriate position).
- Similarly, the new *Delay* values of *task 2* and *task 3* is 150 and 100. And they will also be rearranged into the appropriate positions according to the new *Delay*.



Hình 1.2: Result of Updating and Rearranging tasks.

2.3.4 The initialization function

Like most of the tasks we wish to schedule, the scheduler itself requires an initialization function.

```
void SCH_Init ( void ) {
  unsigned char i ;
  for ( i = 0; i < SCH_MAX_TASKS; i++) {
    SCH_tasks_G[i].pTask = 0x0000;
    SCH_tasks_G[i].Delay = 0;
    SCH_tasks_G[i].Period = 0;
    SCH_tasks_G[i].RunMe = 0;
  }
}
```

Program 1.8: Init function

2.3.5 The 'Update' function

The 'Update' function is involved in the ISR. It is invoked when the timer is overflow.
When it determines that a task is due to run, the update function increments the RunMe field for this task: the task will then be executed by the dispatcher, as we discuss later.
Because we are implementing *SCH_Update()* function with complexity O(1). In this function, we only consider the first element - *SCH_tasks_G[0]* and the rest will have DELAY depending on this first element.

```
void SCH_Update(void){
    if (SCH_tasks_G[0].pTask) {
     if (SCH_tasks_G[0].Delay == 0) {
       // The task is due to run
       // Inc. the 'RunMe' flag
       SCH_tasks_G[0].RunMe += 1;
       if (SCH_tasks_G[0].Period) {
         // Schedule periodic tasks to run again
         SCH_tasks_G[0].Delay = SCH_tasks_G[0].Period;
       }
     }
     else {
       // Not yet ready to run : just decrement the delay
       SCH_tasks_G[0].Delay -= 1;
     }
  }
}


void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
      SCH_Update();
}
```

Program 1.9: SCH_Update function

2.3.6 The 'Add Task' function

As the name suggests, the 'Add Task' function is used to add tasks to the task array, to ensure that they are called at the required time(s). Here is the example of add task function: **unsigned char SCH_Add_Task (Task_Name , Initial_Delay, Period)**
The parameters for the 'Add Task' function are described as follows:

- **Task_Name**: the name of the function (task) that you wish to schedule
- **Initial_Delay**: the delay (in ticks) before task is first executed. If set to 0, the task is executed immediately.
- **Period**: the interval (in ticks) between repeated executions of the task. If set to 0, the task is executed only once

In this function, we first check if there is slot to add a task. If not, return the value FULL, and if so, continue to perform the following steps:

- If the array is empty, we add a new task at the beginning of the array.
- If the array already has elements, then we proceed to compare the *newDelay* value of the new task with each *Delay* value of the existing tasks in the array.
  - If *newDelay* is greater or equal then we subtract the *newDelay* value from the *Delay* value of the current task and then continue to compare with the next tasks.
  - If the *newDelay* value of the new task is smaller than the *Delay* value of the current task, we insert this new task right before and remember to update the *Delay* value for the following task.

```c
unsigned char SCH_Add_Task(void (* pFunction)(),unsigned int DELAY,
                                        unsigned int PERIOD)
{
  //check slot
  unsigned char check = 0;
  for (check = 0; check < SCH_MAX_TASKS; check++){
    if (SCH_tasks_G[check].pTask == 0)
      break;
  }
  if (check == SCH_MAX_TASKS)
  {
    // Task list is full
    // Also return an error code
    return SCH_MAX_TASKS;
  }

  uint32_t newDelay = DELAY;
  unsigned char Index = 0;
  for (Index = 0; Index < check; Index++){

    //neu lon hon hoac bang thi tru roi dung ket qua do
    //so sanh tiep voi cac ptu sau
    if (newDelay >= SCH_tasks_G[Index].Delay) {
      newDelay = newDelay - SCH_tasks_G[Index].Delay;
    }
    //neu be hon thi chen vao dang truoc ptu do va cap nhat lai
    //delay cho ptu phia sau
    else { //newDelay < SCH_tasks_G[Index].Delay
      //update value
      SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Delay -
  newDelay;
      //shift to right from (Index) position
      for (unsigned char i = check; i > Index; i--){
        SCH_tasks_G[i] = SCH_tasks_G[i - 1];
      }
      //add new element
      SCH_tasks_G[Index].pTask = pFunction;
      SCH_tasks_G[Index].Delay = newDelay;
      SCH_tasks_G[Index].Period = PERIOD;
      SCH_tasks_G[Index].RunMe = 0;

      //return position of task
      return Index;
    }
  }
  if (Index == check) {
    SCH_tasks_G[check].pTask = pFunction;
    SCH_tasks_G[check].Delay = newDelay;
    SCH_tasks_G[check].Period = PERIOD;
    SCH_tasks_G[check].RunMe = 0;

    return check;
  }
}
```

Program 1.10: Implementation of the scheduler 'add task' function

2.3.7 The 'Dispatcher' function

As we have seen, the 'Update' function does not execute any tasks: the tasks that are due to run are invoked through the 'Dispatcher' function.
In this function, in addition to the important work of executing the task, then we also need to update the *Delay* value of that task and rearrange the array. And to do this, we take advantage of the *SCH_Add_Task()* function again: we delete the task and add it back to the array with the *Delay* value now replaced by *Period* and ask the *SCH_Add_Task()* function to arrange it in a suitable position in the array.

```
void SCH_Dispatch_Tasks(void){
  unsigned char Index;
  for (Index = 0; Index < SCH_MAX_TASKS; Index++){
    if (SCH_tasks_G[Index].RunMe > 0){
      //Run the task
      (*SCH_tasks_G[Index].pTask)();
      //reset RunMe flag
      SCH_tasks_G[Index].RunMe -= 1;
```

```c

      //if this is a "oneshot" task, remove it from arr
      if (SCH_tasks_G[Index].Period == 0){
        SCH_Delete_Task(Index);
        return;
      }
      //delete from array and add back (to rearange
      //with updated DELAY)
      sTask temp = SCH_tasks_G[Index];
      SCH_Delete_Task(Index);
      //Delay o day da duoc cap nhat thanh Period trong ham Update
      //roi nhung ma do chi la cua phtu 0
      //trong khi chta dang chay loop nen se co truong hop
      //cac phtu khac chua dc cap nhat delay
      //vay nen phai cho input tai vi tri delay la period
      unsigned char i = SCH_Add_Task(temp.pTask, temp.Period,
                                     temp.Period);
      SCH_tasks_G[i].RunMe = temp.RunMe;
    }
  }
}
```

Program 1.11: Implementation of the scheduler ‘dispatch task’ function

The dispatcher is the only component in the Super Loop:

```c
void main(void)
{
    ...
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
```

Program 1.12: The dispatcher in the super loop

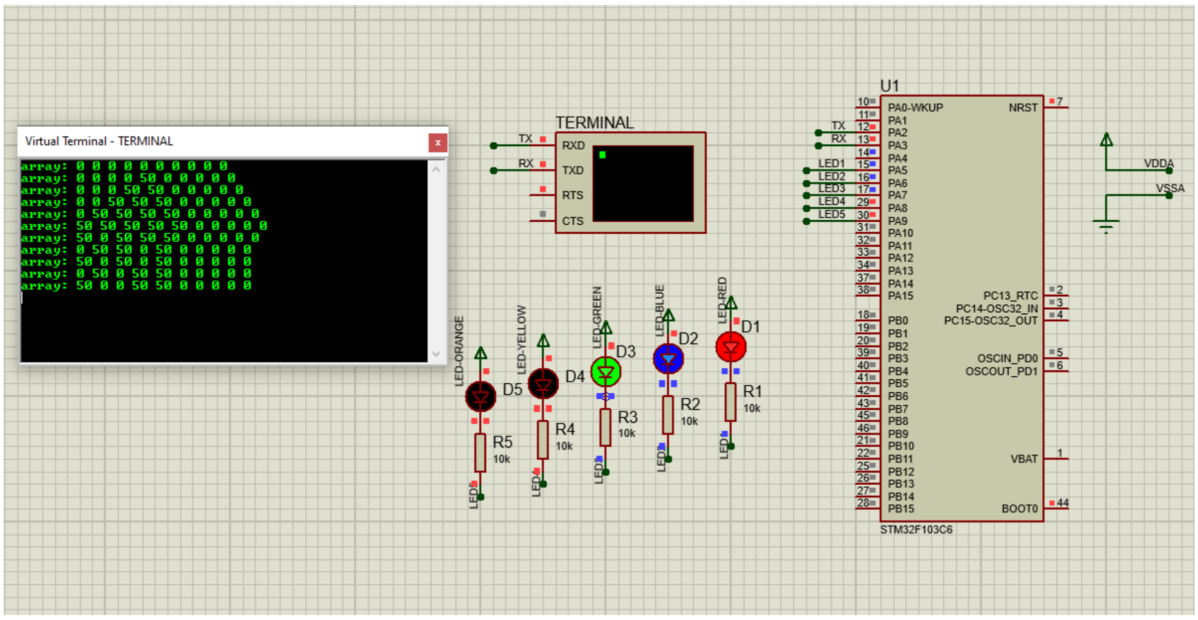
2.3.8 The ‘Delete Task’ function

Sometimes it can be necessary to delete tasks from the array. To do so, SCH_Delete_Task() can be used as follows: SCH_Delete_Task(Task_ID)
When removing an element from the array, we need to move the following tasks to the left to fill the space of the deleted task.

```c
void SCH_Delete_Task(const int TASK_INDEX){
  //shift to left and init the last element
  unsigned char Index = 0;
  for (Index = TASK_INDEX + 1; Index < SCH_MAX_TASKS; Index++) {
    SCH_tasks_G[Index - 1] = SCH_tasks_G[Index];
  }
  SCH_tasks_G[Index - 1].pTask = 0x0000;
  SCH_tasks_G[Index - 1].Delay = 0;
  SCH_tasks_G[Index - 1].Period = 0;
  SCH_tasks_G[Index - 1].RunMe = 0;
}
```

Program 1.13: Implementation of the scheduler ‘delete task’ function

2.4 Check result



```
array: 0 0 0 0 0 0 0 0 0 0
array: 0 0 0 0 50 0 0 0 0 0
array: 0 0 0 50 50 0 0 0 0 0
array: 0 0 50 50 50 0 0 0 0 0
array: 0 50 50 50 50 0 0 0 0 0
array: 50 50 50 50 50 0 0 0 0 0
array: 50 0 50 50 50 0 0 0 0 0
array: 0 50 50 0 50 0 0 0 0 0
array: 50 0 50 0 50 0 0 0 0 0
array: 0 50 0 50 50 0 0 0 0 0
array: 50 0 0 50 50 0 0 0 0 0
```

Hình 1.3: The schematic for checking

For example, we will add 5 tasks running periodically in 0.5 second, 1 second, 1.5 seconds, 2 seconds, 2.5 seconds.

```c
int main(void)
{
 ...
  //Init a schedule
  SCH_Init();
  SCH_Add_Task(LED_blinking, 0, 50);
  SCH_Add_Task(LED_blinking1, 0, 100);
  SCH_Add_Task(LED_blinking2, 0, 150);
  SCH_Add_Task(LED_blinking3, 0, 200);
  SCH_Add_Task(LED_blinking4, 0, 250);

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    SCH_Dispatch_Tasks();
  }
  /* USER CODE END 3 */
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
  SCH_Update();
}
```

Program 1.14: Code in file main.c

To check if updating the task is correct, we use a UART to print the value of the array to the terminal every time we update the task.

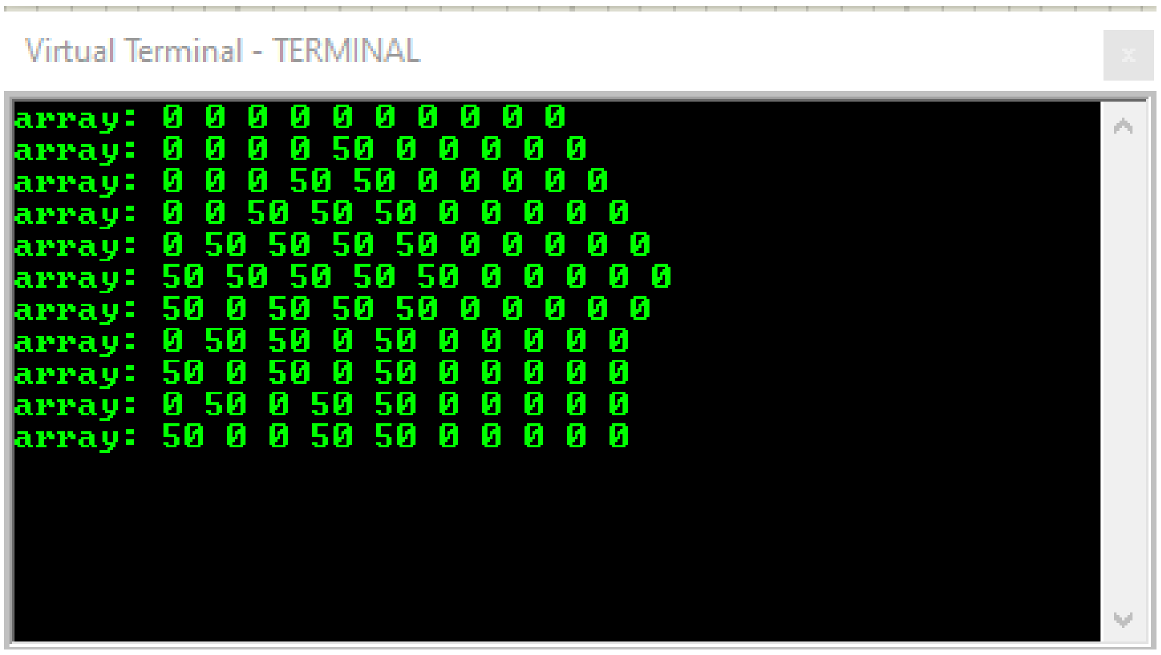
```
static char timeformat[100];

int strlength = sprintf(timeformat, "array: %ld %ld %ld %ld %ld %ld
%ld %ld %ld %ld\r\n", SCH_tasks_G[0].Delay, SCH_tasks_G[1].Delay,
SCH_tasks_G[2].Delay, SCH_tasks_G[3].Delay, SCH_tasks_G[4].Delay,
SCH_tasks_G[5].Delay,SCH_tasks_G[6].Delay, SCH_tasks_G[7].Delay,
SCH_tasks_G[8].Delay,SCH_tasks_G[9].Delay);

HAL_UART_Transmit(&huart2, (uint8_t*)timeformat, strlength,
                          10*strlength);
```

Program 1.15: Code in file main.c

And the result is:



```
array: 0 0 0 0 0 0 0 0 0 0
array: 0 0 0 0 50 0 0 0 0 0
array: 0 0 0 50 50 0 0 0 0 0
array: 0 0 50 50 50 0 0 0 0 0
array: 0 50 50 50 50 0 0 0 0 0
array: 50 50 50 50 50 0 0 0 0 0
array: 50 0 50 50 50 0 0 0 0 0
array: 0 50 50 0 50 0 0 0 0 0
array: 50 0 50 0 50 0 0 0 0 0
array: 0 50 0 50 50 0 0 0 0 0
array: 50 0 0 50 50 0 0 0 0 0
```

Hình 1.4: The result.

Since our array has 10 elements, each row will print 10 values. In this example, we only use 5 tasks, so we only care about the first 5 elements, the remaining elements will be 0 because initially we initialized all elements with the value 0.

In the first line, we see that all 5 elements are zero. This is true because all 5 of our tasks have a *Delay* value of zero - no delay.

In the second line, after *task 1* is done, we set its *Delay* value with *Period* (50ms) and rearrange it. Since this new *Delay* value is larger than all other tasks, it is appended at the end. The value in the current array is: 0 0 0 0 50.

In the third line, after *task 2* is done, we set its *Delay* value with *Period* (100ms) and rearrange it. Since this new *Delay* value is larger than all other tasks, it is appended at the end - after *task 1*. However, before adding, we subtract the *Delay* value of *task 2* for the *Delay* value of *task 1* (100 - 50 = 50). The value in the current array is:

0 0 0 50 50.

After all the tasks have been executed one turn, the current array value is exactly like line number 6: 50 50 50 50 50.

Continuing to execute and update task 1, when we compare, now the *Delay* value of *task 1* is equal to the Delay value of *task 2* (50 = 50), so we subtract it for the Delay of *task 2* (50 - 50 = 0) . Continuing to compare with *task 3*, we see that now the delay of *task 1* is smaller than *task 3* (0 < 50), so we will add *task 1* right before *task 3*. The value in the current array is: 50 0 50 50 50.

Continuing to do the same for the tasks, we can see the output on the terminal exactly as we calculated. Thus, we see that the system works well and correctly.