Ho Chi Minh City University of Technology
MICROPROCESSORS-MICROCONTROLLERS
LAB 5 REPORT
A cooperative scheduler

Vương Quốc Anh

December 12, 2023

1 Introduction

1.1 Super Loop Architecture

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

Listing 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
    }
}
```

Listing 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.

```
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();
    }
}
```

Listing 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.

```
// ------ 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;
}
```

Listing 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).

We consider each of the required components in this section

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;
}
```

Listing 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();
}
```

Listing 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.

```

typedef struct {
    // Pointer to the task (must be a 'void (void)' function)
  void ( * pTask)(void);
  // Delay (ticks) until the function will (next) be run
  uint32_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        40
#define NO_TASK_ID           0
sTask SCH_tasks_G[SCH_MAX_TASKS];
```

Listing 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.

Note also that, if this condition is not satisfied, the scheduler should generate an error code.

2.3.3 The initialization function

Like most of the tasks we wish to schedule, the scheduler itself requires an initialization function. While this performs various important operations – such as preparing the scheduler array (discussed earlier) and the error code variable (discussed later) – the main purpose of this function is to set up a timer that will be used to generate the regular 'ticks' that will drive the scheduler.

```
void SCH_Init(void) {
    unsigned char i;
    for (i = 0; i < SCH_MAX_TASKS; i++) {
        SCH_Delete_Task(i);
    }
    // Reset the global error variable
    // - SCH_Delete_Task() will generate an error code,
    // (because the task array is empty)
    Error_code_G = 0;
    Timer_init();
    Watchdog_init();
}
```

Listing 8: Example of how

2.3.4 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.

```
void SCH_Update(void){
    unsigned char Index;
    // NOTE: calculations are in *TICKS* (not milliseconds)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++) {
        // Check if there is a task at this location
        if (SCH_tasks_G[Index].pTask){
            if (SCH_tasks_G[Index].Delay == 0) {
                // The task is due to run
                // Inc. the 'RunMe' flag
                SCH_tasks_G[Index].RunMe += 1;
                if (SCH_tasks_G[Index].Period) {
                    // Schedule periodic tasks to run again
                    SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                }
            } else {
                // Not yet ready to run: just decrement the delay
                SCH_tasks_G[Index].Delay -= 1;
            }
        }
    }
}

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

Listing 9: Example of how to write an SCH_Update function

2.3.5 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

Here are some examples.

This set of parameters causes the function Do_X() to be executed once after 1,000 scheduler ticks:

SCH_Add_Task(Do_X,1000,0);

This does the same, but saves the task ID (the position in the task array) so that the task may be subsequently deleted, if necessary (see SCH_Delete_Task() for further information about the removal of tasks from the task array):

Task_ID = SCH_Add_Task(Do_X,1000,0);

This causes the function Do_X() to be executed regularly every 1,000 scheduler ticks; the task will first be executed as soon as the scheduling is started:

SCH_Add_Task(Do_X,0,1000);

This causes the function Do_X() to be executed regularly every 1,000 scheduler ticks; task will be first executed at T = 300 ticks, then 1,300, 2,300 etc:

SCH_Add_Task(Do_X,300,1000);

```
/*--------------------------------------------------------------------------*-
SCH_Add_Task() Causes a task (function) to be executed at regular intervals
or after a user-defined delay
-*--------------------------------------------------------------------------*/
unsigned char SCH_Add_Task(void (* pFunction)(), unsigned int DELAY, unsigned int PERIOD)
{
    unsigned char Index = 0;
    // First find a gap in the array (if there is one)
    while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
    {
       Index++;
    }
    // Have we reached the end of the list?
    if (Index == SCH_MAX_TASKS)
    {
        // Task list is full
        // Set the global error variable
        Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
        // Also return an error code
        return SCH_MAX_TASKS;
    }
    // If we're here, there is a space in the task array
    SCH_tasks_G[Index].pTask = pFunction;
    SCH_tasks_G[Index].Delay = DELAY;
    SCH_tasks_G[Index].Period = PERIOD;
    SCH_tasks_G[Index].RunMe = 0;
    // return position of task (to allow later deletion)
    return Index;
}
```

Listing 10: An implementation of the scheduler 'add task' function

2.3.6 The 'Dispatcher'

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.

```
void SCH_Dispatch_Tasks(void)
{
    unsigned char Index;
    // Dispatches (runs) the next task (if one is ready)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++){
        if (SCH_tasks_G[Index].RunMe > 0) {
            (*SCH_tasks_G[Index].pTask)(); // Run the task
            SCH_tasks_G[Index].RunMe -= 1; // Reset / reduce RunMe flag
            // Periodic tasks will automatically run again
            // - if this is a 'one shot' task, remove it from the array
            if (SCH_tasks_G[Index].Period == 0)
            {
                SCH_Delete_Task(Index);
            }
        }
    }
    // Report system status
    SCH_Report_Status();
    // The scheduler enters idle mode at this point
    SCH_Go_To_Sleep();
}
```

Listing 11: An implementation of the scheduler 'dispatch task' function

The dispatcher is the only component in the Super Loop:

```
void main(void)
{
    ...
    while(1)
    {
        SCH_Dispatch_Tasks();
    }
```
Listing 12: The dispatcher in the super loop

Do we need a Dispatch function?

At first inspection, the use of both the 'Update' and 'Dispatch' functions may seem a rather complicated way of running the tasks. Specifically, it may appear that the Dispatch function in unnecessary and that the Update function could invoke the tasks directly. However, the split between the Update and Dispatch operations is necessary, to maximize the reliability of the scheduler in the presence of long tasks.

Suppose we have a scheduler with a tick interval of 1 ms and, for whatever reason, a scheduled task sometimes has a duration of 3 ms.

If the Update function runs the functions directly then – all the time the long task is being executed – the tick interrupts are effectively disabled. Specifically, two 'ticks' will be missed. This will mean that all system timing is seriously affected and may mean that two (or more) tasks are not scheduled to execute at all.

If the Update and Dispatch function are separated, system ticks can still be processed while the long task is executing. This means that we will suffer task 'jitter' (the 'missing' tasks will not be run at the correct time), but these tasks will, eventually, run.

2.3.7 The 'Delete Task' function

When tasks are added to the task array, SCH_Add_Task() returns the position in the task array at which the task has been added: Task_ID = SCH_Add_Task(Do_X,1000,0);

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)

```
/*-------------------------------------------------------------------------*/
unsigned char SCH_Delete_Task(const tByte TASK_INDEX){
    unsigned char Return_code;
    if (SCH_tasks_G[TASK_INDEX].pTask == 0) {
        // No task at this location...
        //
        // Set the global error variable
        Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK

        // ...also return an error code
        Return_code = RETURN_ERROR;
    } else {
        Return_code = RETURN_NORMAL;
    }
    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;
    return Return_code; // return status
}
```
Listing 13: An implementation of the scheduler 'delete task' function

2.3.8 Reducing power consumption

An important feature of scheduled applications is that they can lend themselves to low-power operation. This is possible because all modern MCU provide an 'idle' mode, where the CPU activity is halted, but the state of the processor is maintained. In this mode, the power required to run the processor is typically reduced by around 50

This idle mode is particularly effective in scheduled applications because it may be entered under software control, and the MCU returns to the normal operating mode when any interrupt is received. Because the scheduler generates regular timer interrupts as a matter of course, we can put the system ' to sleep' at the end of every dispatcher call: it will then wake up when the next timer tick occurs.

This is an optional feature. Students can do by yourself by looking at the reference manual of the MCU that is used.

```
void SCH_Go_To_Sleep(){
    //todo: Optional
}
```

Listing 14: An implementation of the scheduler 'go to sleep' function

2.3.9 Reporting errors

Hardware fails; software is never perfect; errors are a fact of life. To report errors at any part of the scheduled application, we can use an (8-bit) error code variable Error_code_G
unsigned char Error_code_G = 0;
To record an error we include lines such as:

- Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
- Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
- Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
- Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
- Error_code_G = ERROR_SCH_LOST_SLAVE;
- Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
- Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;

To report these error codes, the scheduler has a function **SCH_Report_Status()**, which is called from the Update function.

```
void SCH_Report_Status(void) {
#ifdef SCH_REPORT_ERRORS
    // ONLY APPLIES IF WE ARE REPORTING ERRORS
    // Check for a new error code
    if (Error_code_G != Last_error_code_G) {
        // Negative logic on LEDs assumed
        Error_port = 255 - Error_code_G;
        Last_error_code_G = Error_code_G;
        if (Error_code_G != 0){
            Error_tick_count_G = 60000;
        } else {
            Error_tick_count_G = 0;
        }
    } else {
        if (Error_tick_count_G != 0){
            if (--Error_tick_count_G == 0)   {
                Error_code_G = 0; // Reset error code
            }
        }
    }
#endif
}
```

Listing 15: An implementation of the 'report status' function

Note that error reporting may be disabled via the main.h header file:

```
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS
//Where error reporting is required, the port on which error codes will be
    displayed
//is also determined via main.h:
```

```
#ifdef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port PORTA
#endif
```

Listing 16: Define a constant to allow errors are reported

Note that, in this implementation, error codes are reported for 60,000 ticks (1 minute at a 1 ms tick rate). The simplest way of displaying these codes is to attach eight LEDs (with suitable buffers) to the error port, as discussed in IC DRIVER [page 134]: Figure 14.3 illustrates one possible approach.

What does that error code mean? The forms of error reporting discussed here are low-level in nature and are primarily intended to assist the developer of the application or a qualified service engineer performing system maintenance. An additional user interface may also be required in your application to notify the user of errors, in a more user-friendly manner.

2.3.10 Adding a watchdog

The basic scheduler presented here does not provide support for a watchdog timer. Such support can be useful and is easily added, as follows:

- Start the watchdog in the scheduler Start function.
- Refresh the watchdog in the scheduler Update function.

```
IWDG_HandleTypeDef hiwdg;
static uint32_t counter_for_watchdog = 0;

void MX_IWDG_Init(void){
  hiwdg.Instance = IWDG;
  hiwdg.Init.Prescaler = IWDG_PRESCALER_32;
  hiwdg.Init.Reload = 4095;
  if (HAL_IWDG_Init(&hiwdg) != HAL_OK) {
    Error_Handler();
  }
}
void Watchdog_Refresh(void){
  HAL_IWDG_Refresh(&hiwdg);
}
unsigned char Is_Watchdog_Reset(void){
  if(counter_for_watchdog > 3){
    return 1;
  }
  return 0;
}
void Watchdog_Counting(void){
  counter_for_watchdog++;
}

void Reset_Watchdog_Counting(void){
  counter_for_watchdog = 0;
}
```

Listing 17: An implementation of the 'watchdog' functions

2.3.11 Reliability and safety implications

- Make sure the task array is large enough
- Take care with function pointers

- Dealing with task overlap

  Suppose we have two tasks in our application (Task A, Task B). We further assume that Task A is to run every second and Task B every three seconds. We assume also that each task has a duration of around 0.5 ms.

  Suppose we schedule the tasks as follows (assuming a 1ms tick interval):

  SCH_Add_Task(TaskA, 0, 1000);

  SCH_Add_Task(TaskB, 0, 3000);

  In this case, the two tasks will sometimes be due to execute at the same time. On these occasions, both tasks will run, but Task B will always execute after Task A. This will mean that if Task A varies in duration, then Task B will suffer from 'jitter': it will not be called at the correct time when the tasks overlap.

  Alternatively, suppose we schedule the tasks as follows:

  SCH_Add_Task(TaskA, 0, 1000);

  SCH_Add_Task(TaskB, 5, 3000);

  Now, both tasks still run every 1,000 ms and 3,000 ms (respectively), as required. However, Task A is explicitly scheduled always to run 5 ms before Task B. As a result,Task B will always run on time.

  In many cases, we can avoid all (or most) task overlaps simply by the judicious use of the initial task delays.

2.3.12 Portability

3 Objectives

The aim of this lab is to design and implement a cooperate scheduler to accurately provide timeouts and trigger activities. You should add a file for the scheduler implementation and modify the main system call loop to handle timer interrupts.

4 Problem

- Your system should have at least four functions:

- **void SCH_Update(void)**:This function will be updated the remaining time of each tasks that are added to a queue. It will be called in the interrupt timer, for example 10 ms.
- **void SCH_Dispatch_Tasks(void)**: This function will get the task in the queue to run.
- **uint32_t SCH_Add_Task(void (* pFunction)(), uint32_t DELAY, uint32_t PERIOD)**: This function is used to add a task to the queue. It should return an ID that is corresponding with the added task.
- **uint8_t SCH_Delete_Task(uint32_t taskID)**: This function is used to delete the task based on its ID.

You should add more functions if you think it will help you to solve this problem. Your main program must have 5 tasks running periodically in 0.5 second, 1 second, 1.5 seconds, 2 seconds, 2.5 seconds.

5 Demonstration

You should be able to show some test code that uses all the functions specified in the driver interface.

Specifically set up and demonstrate:

- A regular 10ms timer tick.

- Register a timeout to fire a callback every 10ms.
- Then, print the value returned by get_time every time this callback is received.
- Note: Your timestamps must be at least accurate to the nearest 10ms.
- Register another timeout at a different interval in addition to the 500ms running concurrently (i.e. demo more than one timeout registered at a time).
- Before entering the main loop, set up a few calls to SCH_Add_Task. Make sure the delay used is long enough such that the loop is entered before these wake up. These callbacks should just print out the current timestamp as each delay expires.

Note this is not a complete list. The following designs are considered unsatisfactory:

- Only supporting a single timeout registered at a time.
- Delivering callbacks in the wrong order
- O(n) searches in the SCH_Update function.
- Interrupt frequencies greater than 10Hz, if your timer ticks regularly.

6 Submission

You need to

- Demonstrate your work in the lab class and then
- Submit your source code to the BKeL.

https://github.com/VgQ8Auk/LAB5

7 References