

STM32 Microcontrollers



Tutorial #2.6 : Simple delays using timers	Rev : 0
Author(s) : Laurent Latorre	Date : 22-08-2015

Contents

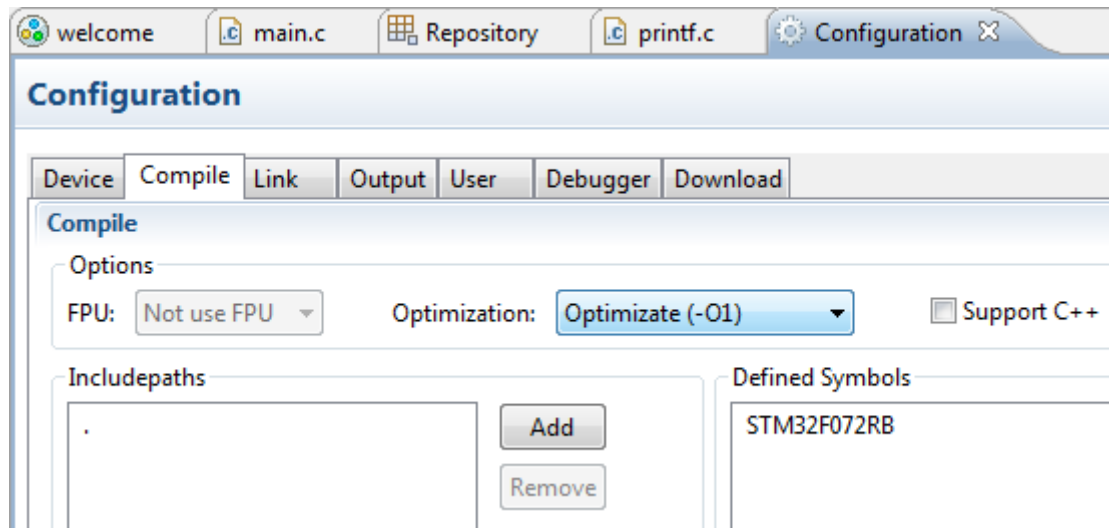
1. Why software counting is not a good practice for delays.....	2
2. Simple delay using a general purpose Timer.....	3
2.1. Start a new project	3
2.2. Time-Base timer configuration.....	3
2.3. Using the “update” event flag.....	4
2.4. Reading the Timer current value.....	5
2.5. A delay function.....	6
3. Delay function using SysTick counter	7
3.1. Add a new source files to the project	7
3.2. Writing delay.c and delay.h code	10
3.3. Prepare main.c for use with the delay function.....	11
3.4. Test the delay function.....	11
4. Summary.....	14



1. Why software counting is not a good practice for delays

Start ColIDE and open back the previous tutorial project “analog_mon”.

Build  the project and flash  your target board. Open a terminal window and make sure that the program works as before. Try to memorize the blinking period (roughly).

Open the project Configuration tab, and change Optimization from None (-O0) to Optimize (-O1)



Rebuild  the project and flash  your target board again. Look at your LED (or the console). What do you see?

Repeat the previous steps with a further optimization level (-O2, -O3). Starting from -O2 optimization, the counting loop runs so fast, that is it barely possible to see the LED blinking (although it is, but very fast).

Go back to -O0 optimization level and rebuild the code. Then flash the target a last time to recover initial behavior.

The fact is that using optimization switch changes the way your C source code translates into assembly code, making it more and more efficient as optimization level increases. Nevertheless, using optimization level different from -O0 needs special care as optimizer removes “useless” operations and variables. Code that is “useless” to the optimizer is not always useless to you. A debug message is an example, or some intermediate variables. So beware with optimization, and it is always a good idea to start writing code with no optimization. This way, you are sure to find all your stuff in the debugger.

The only purpose of the above experiment is to make clear that timing based on software counting is very dependent to external parameters such as code optimization, or clock source/frequency. It is therefore advisable to implement delays with more robust approaches.

2. Simple delay using a general purpose Timer

2.1. Start a new project

Repeating the whole procedure described in the previous tutorial, prepare a new project “**timer_delay**” by copying/cleaning/renaming the “**analog_mon**” folder project. Then open “**timer_delay**” project in ColIDE.

2.2. Time-Base timer configuration

Go to the **Repository** and add **TIM** peripheral driver to your project. Include the new driver header to your **main.c** file.

Let us add an initialization function for Timer 3 (TIM3) peripheral at the end of our **main.c** file

```
static void TIM3_Config(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

    /* TIM2 clock enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

    /* Time base configuration */
    TIM_TimeBaseStructure.TIM_Period = 5000;
    TIM_TimeBaseStructure.TIM_Prescaler = 4800-1;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

    /* Enable the Update Event */
    TIM_UpdateDisableConfig(TIM3, DISABLE);

    /* Enable TIM3 */
    TIM_Cmd(TIM3, ENABLE);
}
```

Add the **TIM3_Config()** function prototype above **main()** function.

Now, we need to remember which frequency is used to clock the Timer 3. That information was in the Excel tool we used to configure the system clock. If you don't remember, you can have a look in the **system_stm32f0xx.c** file.

With a system frequency of 48MHz, and APB1 bus prescaler set to 1, the Timer 3 is clocked by a 48MHz frequency.

According to STM32F0 Reference manual TIM3 is based on a 16-bit counter. At 48MHz, the counter register goes from 0 to 2^{16} in 1.36ms. How can we use it to measure longer period of time?

All STM32 timers have 16 bits clock prescalers before the counter register. You can therefore program any division factor from 1 to 65535 to apply to the system timer clock. In the above example, the division factor being 4800, the counter value increments at a 48MHz/4800 frequency. In other words, an increment appends every 100µs.

That being said, you can also program the maximum value that the counter must reach before restarting from zero. This is the timer period. By setting its value to 5000 in the above code, we have a counter reloading zero value every $5000 \times 100\mu s = 0.5s$. The value of the period must be between 1 and 65535 since TIM3 is 16 bits. Some timers in the STM32 are 32 bits (TIM2 for instance), providing more flexibility.

2.3. Using the “update” event flag

When the counter reloads, it produces an “update” event which internally sets an “update” flag in a specific register. Let us use this feature to implement a first simple delay in our main loop:

```
// Main program

int main(void)
{
    /* Init Peripherals */
    LED_Init();
    USART2_Init();
    ADC_Config();
    TIM3_Config();

    my_printf("Init done !\n\r");

    /* Loop forever */
    while(1)
    {
        /* Test EOC flag */
        while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);



        /* Get ADC1 converted data */
        ADC1ConvertedValue = ADC_GetConversionValue(ADC1);

        /* Print out the result of conversion */
        my_printf("ADC value is %d\r\n", ADC1ConvertedValue);

        /* Wait for TIM3 Update Event (Counter Overflow) */
        while(TIM_GetFlagStatus(TIM3, TIM_FLAG_Update) != SET);

        /* Clear the TIM3 Update Event Flag */
        TIM_ClearFlag(TIM3, TIM_FLAG_Update);

        /* Toggle the LED */
        LED_Toggle();
    }
}
```

Build  the project and flash  your target board. Experiment with various optimization levels. What do you see?

2.4. Reading the Timer current value

You can as well read the TIM3 counter value and make decision based on that value. Test the following example:

```
// Main program

int main(void)
{
    /* Init Peripherals */
    LED_Init();
    USART2_Init();
    ADC_Config();
    TIM3_Config();

    my_printf("Init done !\n\r");

    /* Loop forever */
    while(1)
    {
        /* Test EOC flag */
        while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);

        /* Get ADC1 converted data */
        ADC1ConvertedValue = ADC_GetConversionValue(ADC1);

        /* Print out the result of conversion */
        my_printf("ADC value is %d\r\n", ADC1ConvertedValue);

        /* LED Flashing for 50ms */
        while(TIM_GetCounter(TIM3)<500)
        {
            GPIO_SetBits(LED_PORT, LED_PIN);
        }

        GPIO_ResetBits(LED_PORT, LED_PIN);

        /* Wait for TIM3 Update Event (Counter Overflow) */
        while(TIM_GetFlagStatus(TIM3, TIM_FLAG_Update)!=SET);

        /* Clear the TIM3 Update Event Flag */
        TIM_ClearFlag(TIM3, TIM_FLAG_Update);
    }
}
```

2.5. A delay function

Let us now write a function `delay_ms(ms)` that takes a number `ms` of milliseconds as an argument. For that, we will edit the Timer 3 initialization function:

- **Prescaler** is set to 48000 so that $48\text{MHz}/48000=1\text{kHz}$ → Counting rate is 1ms
- **Period** is set to maximum value for a 16-bit counter (i.e. 65535)
- We're not using the **update** event anymore
- We're not starting the counter yet, this task is let to the **delay_ms()** function

```
// Timer 3 Init Function
static void TIM3_Config(void)
{
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;

    /* TIM2 clock enable */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

    /* Time base configuration */
    TIM_TimeBaseStructure.TIM_Period = 65535;          // Max
    TIM_TimeBaseStructure.TIM_Prescaler = 48000-1;     // Counter rate is 1ms
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
}

// Simple Delay Function in Milliseconds
void delay_ms(uint16_t ms)
{
    /* Make sure TIM3 counter start from zero */
    TIM_SetCounter(TIM3, 0);

    /* Enable TIM3 */
    TIM_Cmd(TIM3, ENABLE);

    /* Wait ms milliseconds */
    while (TIM_GetCounter(TIM3)<ms);

    /* Disable TIM3 */
    TIM_Cmd(TIM3, DISABLE);
}
```

Add the **delay_ms()** function prototype before `main()`, and use it in this basic example:

```
// Main program

int main(void)
{
    /* Init Peripherals */
    LED_Init();
    USART2_Init();
    ADC_Config();
    TIM3_Config();

    my_printf("Init done !\n\r");

    /* Loop forever */
    while(1)
    {
        /* Test EOC flag */
        while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);

        /* Get ADC1 converted data */
        ADC1ConvertedValue = ADC_GetConversionValue(ADC1);

        /* Print out the result of conversion */
        my_printf("ADC value is %d\r\n", ADC1ConvertedValue);

        /* LED Flashing for 50ms */
        GPIO_SetBits(LED_PORT, LED_PIN);
        delay_ms(50);
        GPIO_ResetBits(LED_PORT, LED_PIN);
        delay_ms(450);
    }
}
```

STM32 Timers are able to perform a lot of time-related tasks, including the multi-channel measure/generation of pulses, the management of PWM outputs or the reading of encoder wheels. These peripherals implements highly configurable operations with interrupts and DMA also.

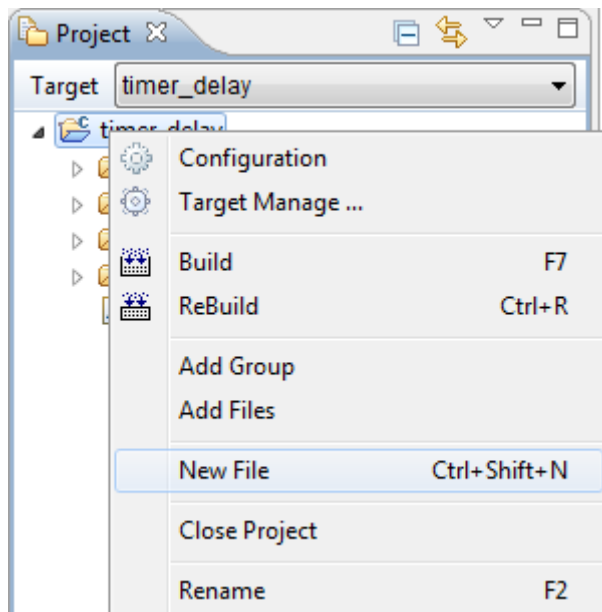
Therefore, the use of such complex peripheral for a simple delay implementation is a waste of resource, and not the best option. Besides, STM32 devices feature a specific time-running counter called **SysTick**. This is a 24-bit counter that is devoted to simple time measurement applications.

3. Delay function using SysTick counter

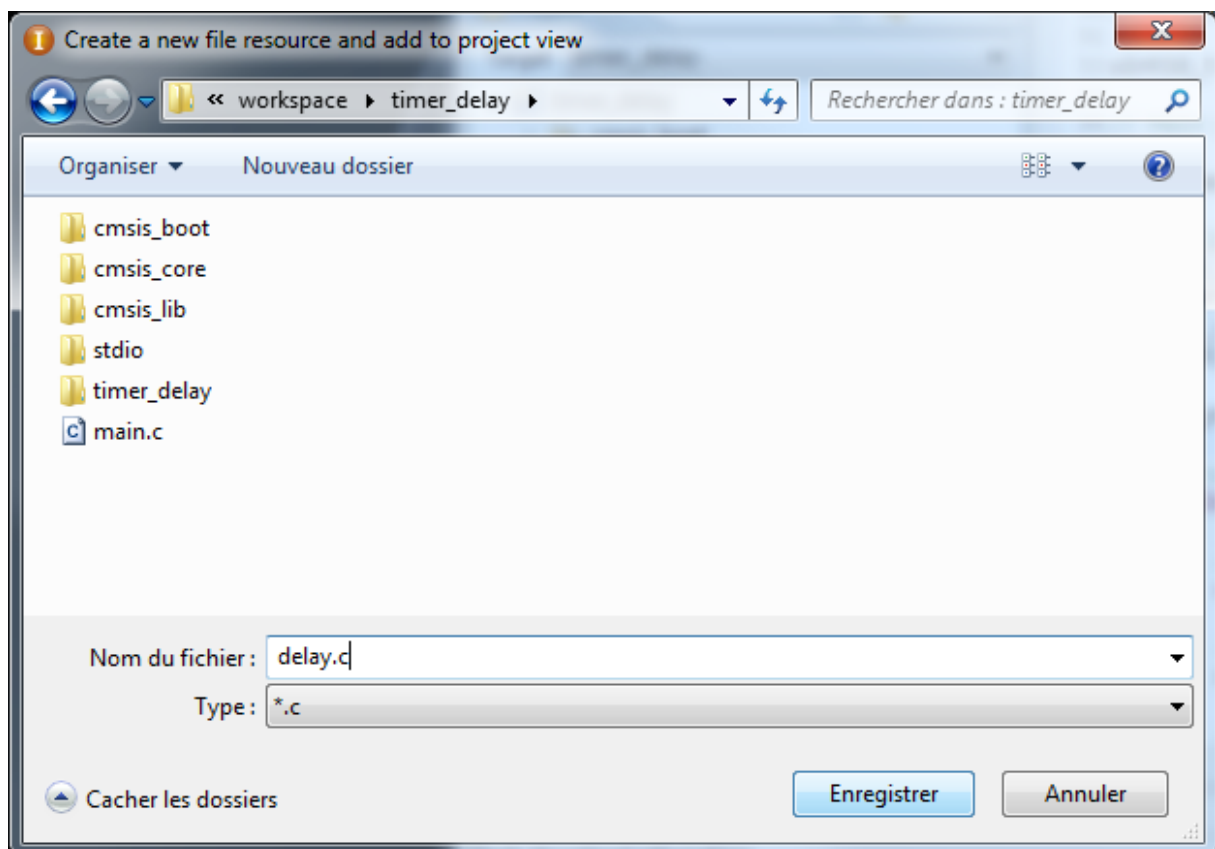
3.1. Add a new source files to the project

In order to keep the **main.c** file clear, and to make delay functions available for other projects, let us write the corresponding code in a separate file **delay.c**:

In the **Project** tab, right-click on the main folder and go to **New File**.

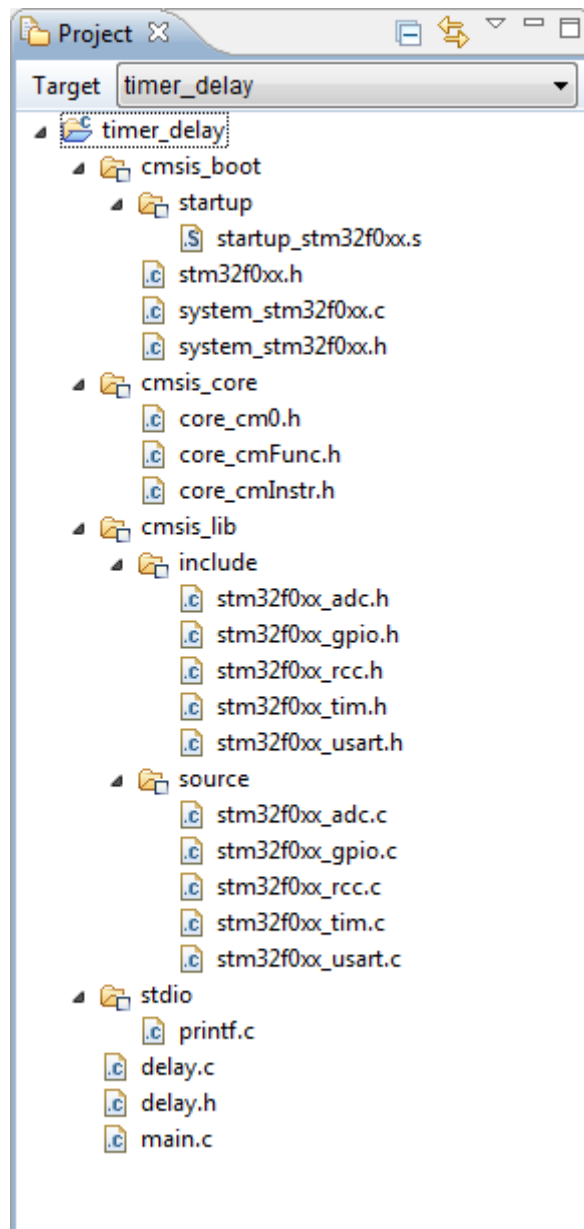


Browse to your project folder and add the file **delay.c**



Repeat the previous operation and add a header file “delay.h” to your project.

The whole project tab is now:



3.2. Writing delay.c and delay.h code

Edit the **delay.c** file and save it:

```
1
2 // Basic delay functions based on SysTick timer
3
4 #include "stm32f0xx.h"
5
6 static volatile uint32_t    timer=0;
7 extern uint32_t            SystemCoreClock;
8
9 // SysTick Initialization
10
11 void SysTick_Init()
12 {
13     SystemCoreClockUpdate();
14     while (SysTick_Config(SystemCoreClock/1000) != 0); // 1ms resolution
15 }
16
17 // SysTick Interrupt Handler
18
19 void SysTick_Handler(void)
20 {
21     if (timer !=0)
22     {
23         timer--;
24     }
25 }
26
27 // Basic delay function
28
29 void delay_ms(uint32_t ms)
30 {
31     timer = ms;
32     while(timer != 0);
33 }
34
```

The **SysTick_Init()** function starts the **SysTick** timer and configures the number of ticks between interrupt events. Here, the number of ticks is $48.10^6/1000 = 48000$. Interval between interrupts is therefore $48000/48.10^6 = 1\text{ms}$.

Because of that, the function **SysTick_Handler()** is executed every 1ms, no matter what the program is doing. In other words, your program is “interrupted” every 1ms to run this function, and then resumes from the point it was interrupted after the function returns.

The **delay_ms()** function just sets the **timer** global variable to the number of milliseconds you want to wait, and then loops until **timer** is zero, given that **timer** is decremented each time the interrupt handler is called (i.e. every millisecond).

From the **main.c** file, we will need to call the two functions

- SysTick_Init() → To start the SysTick process

- `Delay_ms()` → Everywhere we need to wait some time

We therefore need to declare those functions in the **main.c** file, before the **main()** function. Let us do that by creating a dedicated header.

Now edit **delay.h** file:

```
1
2 // Header for delay.c functions
3
4 void SysTick_Init(void);
5 void delay_ms(uint32_t);
6
```

Then save this header.

3.3. Prepare main.c for use with the delay function

Before going further, we need to do some cleaning in the **main.c** file:

- Delete or comment the whole first version of **delay_ms()** function (the one making use of Timer 3). Tip: Select all the lines you want to comment and press CTRL-SHIFT-/
- Delete or comment the first **delay_ms()** prototype declaration
- Include the **delay.h** header
- As we are not using Timer 3 anymore, you can remove the **TIM3_Config ()** call in the **main()** function. You don't have to delete the **TIM3_Config ()** function itself. Just keep it in the code for further reference.
- On the other hand, you need to start the **SysTick** process by calling **SysTick_Init()**

3.4. Test the delay function

In summary, you should have the declaration section on top of **main.c** now being:

```

1
2 // Include main Header
3
4 #include "stm32f0xx.h"
5
6 // Include Peripheral Headers
7
8 #include "stm32f0xx_rcc.h"
9 #include "stm32f0xx_gpio.h"
10 #include "stm32f0xx_usart.h"
11 #include "stm32f0xx_adc.h"
12 #include "stm32f0xx_tim.h"
13
14 // Include User Headers
15
16 #include "delay.h"
17
18 // Defines
19
20 #define LED_PIN    GPIO_Pin_5
21 #define LED_PORT   GPIOA
22
23 // Function prototypes
24
25 void LED_Init(void);
26 void LED_Toggle(void);
27 void USART2_Init(void);
28 signed int my_printf(const char *pFormat, ...);
29 static void ADC_Config(void);
30 static void TIM3_Config(void);
31
32 // Global variables
33
34 uint16_t    ADC1ConvertedValue=0;
35

```

Then the **main()** function


```

35
36 // Main program
37
38 int main(void)
39 {
40     /* Init Peripherals */
41     LED_Init();
42     USART2_Init();
43     ADC_Config();
44     SysTick_Init();
45
46     my_printf("Init done !\n\r");
47
48     /* Loop forever */
49     while(1)
50     {
51         /* Test EOC flag */
52         while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);
53
54         /* Get ADC1 converted data */
55         ADC1ConvertedValue = ADC_GetConversionValue(ADC1);
56
57         /* Print out the result of conversion */
58         my_printf("ADC value is %d\r\n", ADC1ConvertedValue);
59
60         /* LED Flashing for 50ms */
61         GPIO_SetBits(LED_PORT, LED_PIN);
62         delay_ms(50);
63         GPIO_ResetBits(LED_PORT, LED_PIN);
64         delay_ms(450);
65     }
66 }
67

```


And then all the functions we have left from previous tutorials:

- LED_Init()
- LED_Toggle()
- USART2_Init()
- ADC_Config()
- TIM3_Config()
- delay_ms() MUST have been deleted (or commented) as two versions of the same function may not co-exist in the same project.

Build  the project and look at the console. If you have left **TIM3_Config()** function code as above suggested, you will get a warning that the function is not used. This is normal, and not a problem.

```
Console
Build
GCC HOME: D:\CooCox\GNU Tools ARM Embedded\4.9 2015q2\bin
compile:
[mkdir] Created dir: D:\CooCox\workspace\timer_delay\timer_delay\Debug\bin
[mkdir] Created dir: D:\CooCox\workspace\timer_delay\timer_delay\Debug\obj
[cc] 10 total files to be compiled.
[cc] arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb -Wall -ffunction-sections -g -O0 -c -DSTM32F072RB -ID:\CooCox -ID:\CooC
[cc] 200
[cc] D:\CooCox\workspace\timer_delay\main.c: 184: 13: warning: 'TIM3_Config' defined but not used [-Wunused-function]
[cc] static void TIM3_Config(void)
[cc]      ^
[cc] Starting link
[cc] arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb -g -nostartfiles -Wl,-Map=timer_delay.map -O0 -Wl,--gc-sections -LD:\CooC
Program Size:
text    data    bss     dec     hex     filename
6556    1104      8     7668    1df4    timer_delay.elf

BUILD SUCCESSFUL
Total time: 4 seconds
```

Flash  your target board and make sure that the program works as before.

4. Summary

In this tutorial, you have seen two approaches for generating delays. Because timers are advanced peripherals, simple delay functions should be implemented using the **SysTick** timer instead.

Note that most RTOS will use the SysTick timer to produce their main tick events.